Unit - 1

Introduction & overview

- Data may be organised in different ways:
- The logical or mathematical model for placing data & the particular format or organization is called DS
- The choice of particular data model depends on 2 considerations:

(i) It must be rich enough in structure to mirror the actual relationship b/w data in real world.

(ii) The structure should be simple, that one can process effectively whenever necessary.

- It can be defined as _____ which provides an efficient way of storing & organising data in computer. so that it can be use effectively.

Eg:- Arrays, linked list

- Data structure widely used in almost every aspect of compiter science i.e, used in OS, AI, Graphics, compiler design & many more.

→ Basic terminologies: Elementary data organization

- Data structure are the building blocks of any program of the s/w.

(i) Data is an elementary value or collection of value.

Eg:- Student name, id, are the data about the students.

(ii) Group items :- Data items which has subordinate data items.

Eg:- Name of the student has firstname, middle name & last name.

(iii) Elementary items :- Data items that are not able to divide in to sub data item.

Eg :- record, SSN.

(iv) Record :- Is a collection of various data items.

Eg:- consider a student entity & its name, course, address & marks can be grouped together to form record for student.

- Record may classified according to length:

1) Fixed length record :- All records contain the same data item with same amount of space assigned to each data items.

2) Variable length record :- Record which contains different length.

- Variable length record have a minimum & maximum length.

Eg:- Student records have variable length, since different students take different no of courses,

→ classification of data structure :-

(i) Primitive data structure :- are fundamental rights supported by programing languages.

- Basic datatype such as real, integer, char & boolean are known as primitive data structure.
- These datatypes consist of characters that cannot be divided & hence called as simple datatype.

(ii) Non-Primitive data structure :-

are those data structure which are created using primitive data structure.

- Based on structure & arrangement of data, non-primitive data structure is further classified into linear & non-linear.

1) Linear data structure :-

- The elements are stored in the form of sequence / linear list.
- 2 ways of representing such linear structure in memory are:
  (a) To have linear relationship b/w elements represented by means of sequential memory location, these linear structure are called arrays.
  (b) To have linear relationship b/w elements represented by means of pointers / links, these linear structure are called linked list.

2) Non-linear data structure :-

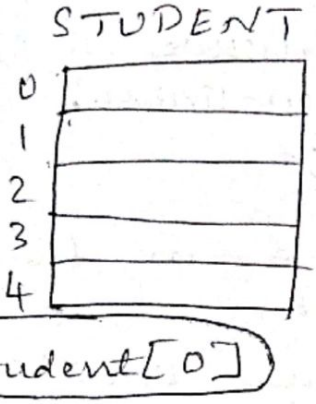
- If data are not arranged in sequence or linear, it is called non-linear data structure.
- This structur is mainly used to represent data containing hierarchical relationship b/w elements.

Eg:- Trees, Graphs.

→ Arrays :-

- Simplest type of data structure is linear / one-dimension array.

Eg:-



- Linear array is also called one dimensional array; each element in an array is reference by 'one' sub-script.

- Two-dimension array is a collection of similar data element, where each element is referend by using '2' sub-scripts.

Eg:-

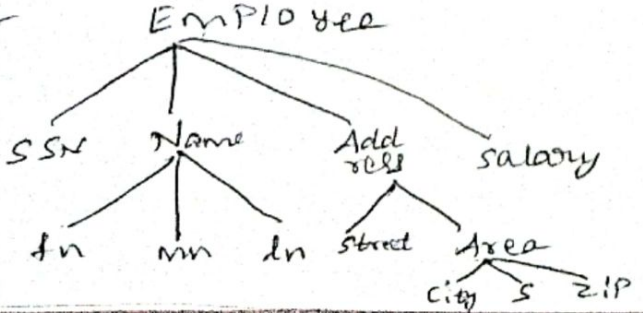
Student	1	2	3	4
1	2001	2002	2003	2004
2	1990	1991	1992	1993
3		1981		

Student [3, 2] = 1981

→ Trees :-

- Data frequently contains a hierarchical relationship b/w various elements.
- The data structure which reflects this relationship called a rooted tree ~~graph~~ map / tree.

Eg:-



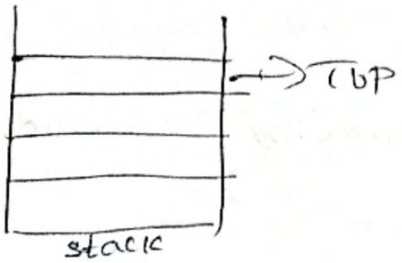
- Another way of representing a tree:

→ 01 : Employee → Root Tree.
02 : SSN
02 : Name
03 : LN
03 : MN
03 : FN
→ 02 : Address
03 : Street
03 : Area
04 : State
04 : zip code
04 : City
02 : salary
02 : ~~Marital~~ ~~Sex~~ Dependents
02 : Age

→ Stack :-

- It follows LIFO principles.
- stack is a linear list in which insertion & deletions can take place only at one end called TOP

eg:- consider 4 dishes that are inserted only at the top of stack & dishes can be deleted only top of stack.



→ Queue :-

- It follows FIFO principles.
- Queue is a linear list in which deletion can take place only at one end of queue called Front (Dequeue)

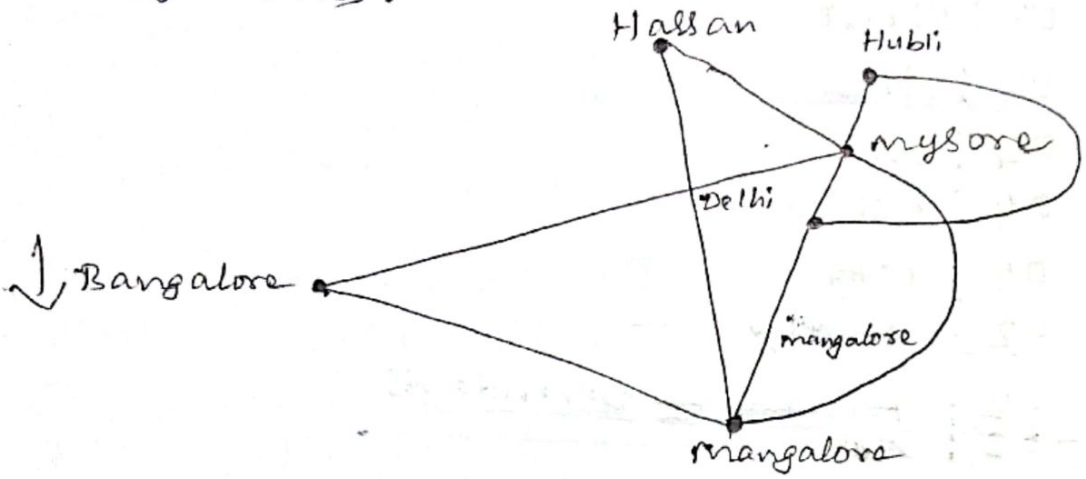
- An insertion can take place only at another end of list called Rear (EnQueue).

eg:- Boarding a Bus.

→ Graph :-

- Data sometimes contains a relationship b/w pair of elements which is not necessarily hierarchical in nature.

eg:- Supose airlines flies b/w city connected by lines.



→ Operations of datastructure :-

(i) Traversing :- Accessing each records or nodes exactly once, so that certain itemes in the records may be processed

(ii) Searching :- Finding the location of a desired record with a given key value.

(iii) Inserting :- Adding a new node to the structure.

(iv) Deleting :- Removing a node / record from structure.

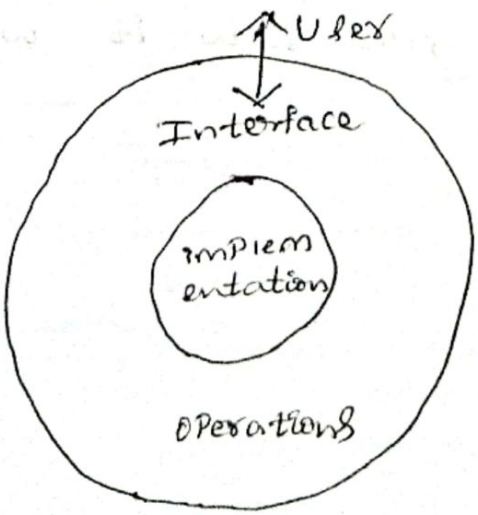
(v) Sorting :- Arranging the record in some logical order.

eg:- Alphabetically, According to some name key / neumerical order according to some no key such as SSN or account no

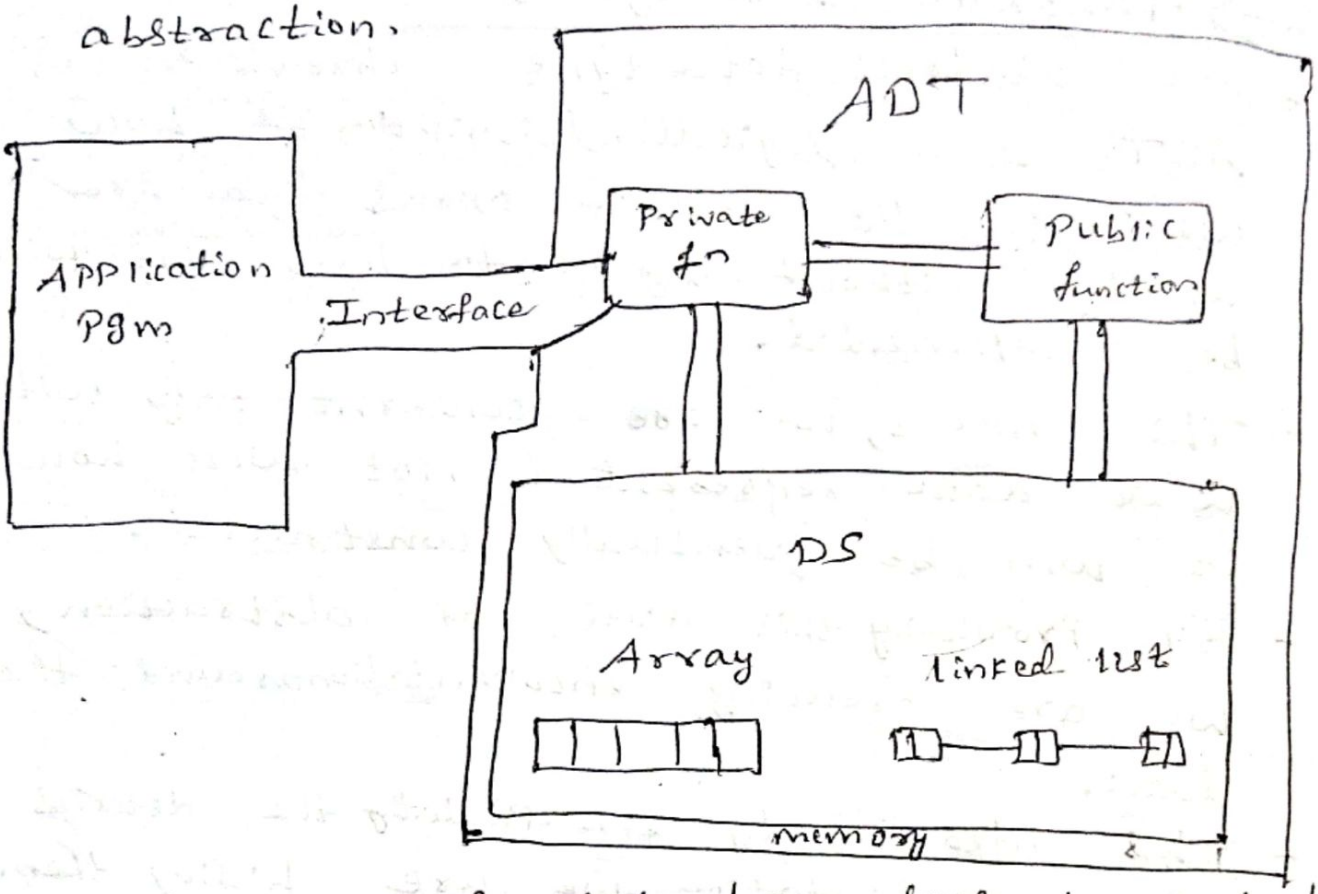
(vi) merging :- combining the records in 2 different sorted files into a single sorted files.

⇒ Abstract data type :-

- An abstract data type abreviated as ADT is a logical description of how we view the data & oprns that are allowed without regard to how they will be implemented.
- This means, we are consernt only with what data represent & not with how it will be eventually constructed.
- By Providing this level of abstraction, we are creating encapsulation around the data.
- This idea is by encapsulating the details of implementation we are hiding them from user's view, this is called info hiding.
- The implementation of ADT often refered as data structure will require that we provide a physical view of data using some collection of programing constraints & primitive data - type.



- This called abstract because it gives a implement^n independent view.
- The process of providing only essentials & hiding the details is known as abstraction.



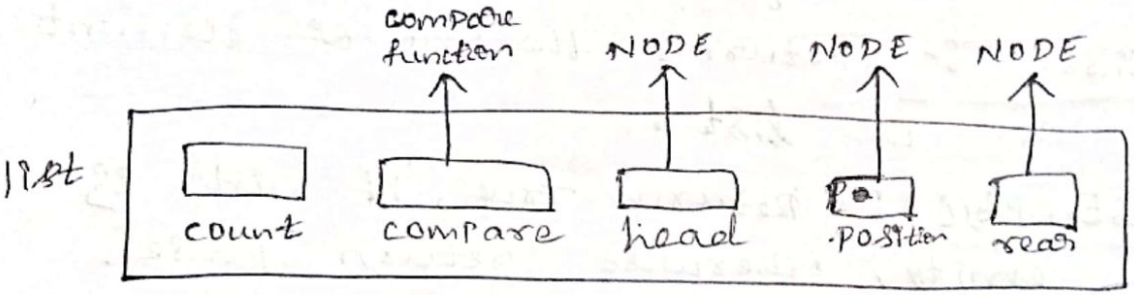
- The user of data type does not need to know how data type is implemented.

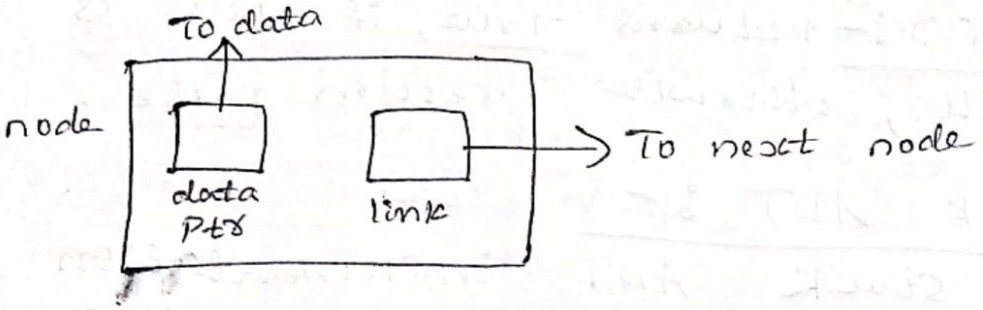
Eg:- we having using primitive values like int, float, char data type with only a knowledge, this data type can operate & be performed without any idea of how they are implemented, so a user only needs to know what a datatype can do but not how it will be implemented.

↳ 3 Types of ADT :-

(1) List ADT :-

- The data is generally stored in key sequentially in a list which has a head structure consisting of count, pointers & address of compare function need a to compare data in a list.





- The data node contain the pointer to the data structure & a cell Preferencial Pointer which Points to the next node of the list.
- A list contains an element of same type arranged in sequential order & following operations can be performed on the list

a) get() :- Returns an element of any given position of the list.

b) insert() :- Insert an element at any position in the list.

c) remove( ): Removes an element from a non-empty list.

d) removeAt( ): Removes an element at a specified location from a non-empty list.

e) replace( ): - Replace an element at any position by another element.

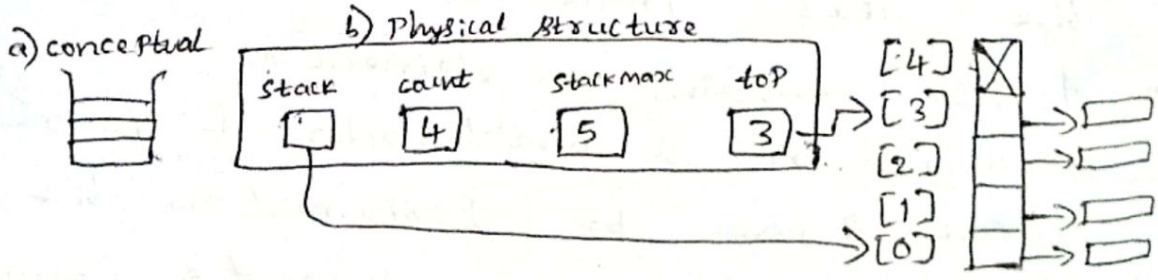
f) size( ): - Returns the no of element in the list.

g) isEmpty( ): - Returns True, if list is empty, otherwise return False.

h) isFull( ): - Returns True, if list is full, otherwise return False.

(2) Stack ADT :-

- In stack ADT implementation instead of data being stored in each node, the pointer to data is stored.
- The program allocates memory for the data & address is passed to the stack ADT.



- The stack head structure also contains a pointer to top & count of no of entries currently in stack.

- A stack contains elements of a same type arranged in sequential order.
- All oprns takes place in single end i.e TOP of the stack.

operations:-

a) Push( ) :- Insert an element at one end of stack called top.

b) POP( ) :- Remove & returns the element at the top of the stack if it is not empty.

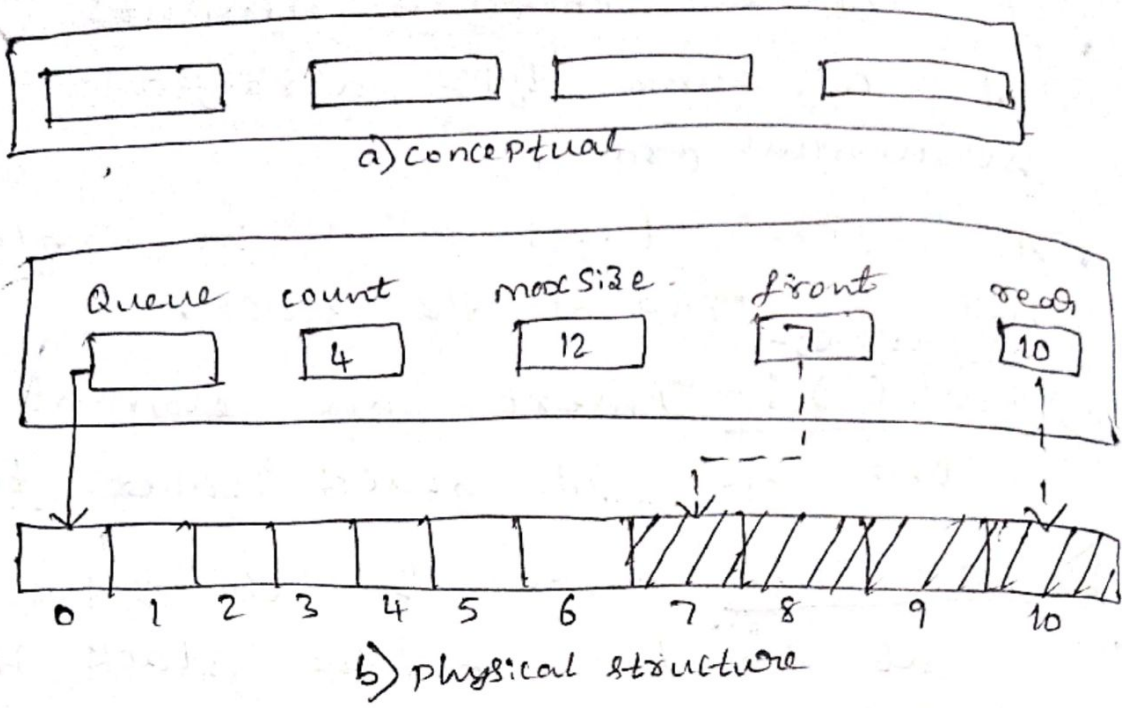
c) Peek( ) :- Returns the element at the top of the stack without removing it, if the stack is not empty.

d) Size( ) :- Returns the no of elements in the stack

e) isEmpty( ) :- Returns 'True', if the stack is empty, otherwise 'false'.

f) isFull( ) :- Returns 'True', if the stack is full, otherwise 'false'

3) Queue ADT :-



- Queue contains element of same type arranged in sequential order.
- Operation take place in both end, insertion is done at rear end & the deletion is done at front.

a) Enqueue( ) :- Inserting an element at the end of the queue.

b) Dequeue( ) :- Removing & returning the 1st element of queue; if the queue is not empty.

c) Peek( ) :- It returns the element of the queue without removing it, if the queue is not empty.

d) size( ) :- Returning the no of element in queue.

e) isEmpty( ) :- Return 'True' if queue is empty, otherwise return 'False'.

f) isFull( ) :- Return 'True' if queue is full, otherwise return False.

→ Algorithm complexity :-

- The Perform of a program is the amount of computer memory & time needed to run a program.
- The Perform of a algorithm can be measured by time & space.

→ Time complexity of algorithm :-

- Time complexity algorithm is the no of dominating operation executed by the algorithm as the function of data size,
- Time complexity measures the amount of time done by algorithm during solving the problem in the way which is independent on the implementation & Particular i/p data.
- The lower time copiexity, the "faster" algorithm

Eg:

```
find (arr, len, key)
{   i=0
while (i < len)
{
if (array[i] == key)
   return i
}i++
return -1
}
```

→ Space complexity of algorithm :-

- $S(n)$ is the no of units of memory used by algorithm as a function of data size.

Eg:-
```
int mul, i
while i<=n
do
mul ← mul * array(i)
i ← i+1
end while
return mul
```

- Let $S(n)$ denotes algorithm space complexity in more system & integer occupies 4 bytes in memory.
- As a result, the no of allocated bytes would be a space complexity
- Line-1 denotes allocation of memory space of 2 integer, resulting in $S(n) = 4$ bytes multiplied by 2 equals 8 bytes.
- Line-2 represent loop.
- Line-3 & 4 assigns value to mul, i.
- The return statement will allocates one more memory space.
- As a result $s(n) = 4$ times $*2+4$ $= 12$ bytes.
- The algorithm to allocate n-cases of integer thens the final space complex will be high...

Eg: if $f(n) = n + 12 = O(n)$

→ Diff b/w Time & Space complexity

Time complexity	Space complexity
- Calculate time requirements.	- Estimate memory space requirement.
- Time is counted for all statements.	- Memory space is counted for all variable i/p & o/ps.

→ Asymptotic notation:-

- It is often used to design the size of i/p data affect in algorithm used in of computational resources.
- It is mathematical notation used to describe runing time of algorithm when i/p tends towards particular value or limited value.
- They are expressions used to represent computation of algorithm.

Eg: In bubble-sort, when i/p arrangments are sorted, then time taken by algorithm is linear.

↳ Types of Asymptotic notation :-

(i) Big - oh (O) :-

- If $f(n) = O.g(n)$, if there exists a positive integer 'n_0' & positive number 'c', such that:

$$f(n) \leq c.g(n)$$

for all: $n \geq n_0$

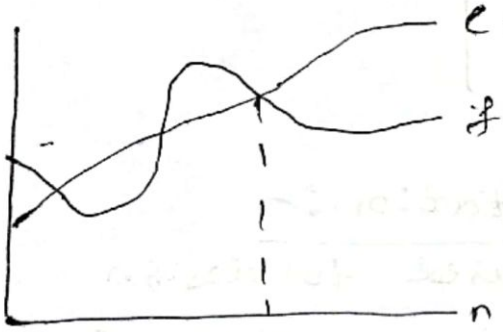
- Here, $g(n)$ is upper bound of function $f(n)$

Eg:- $16n^2 + 45n^2 + 12n$

$34n - 40$

50

n^2	$f(n) = O(n^3)$
n	$f(n) = O(n)$
1	$f(n) = O(1)$



$f(n) = O.g(n)$

Big-oh notation represents upper bound of the running time of an algorithm.
- Thus, it gives the worst-case complexity of an algorithm.

(ii) Big - omega (Ω) :-

- $f(n) = \Omega.g(n)$, if there exists a positive integer 'n_0' & positive number 'c', such that:

$$f(n) \geq c.g(n)$$

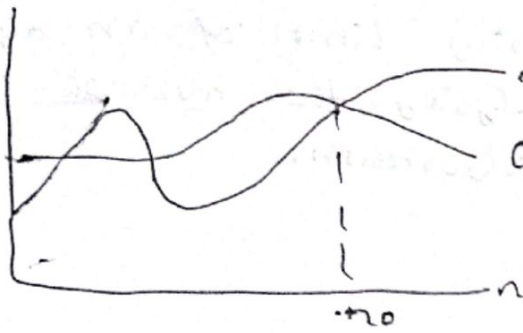
$\forall$ $n \geq n_0$

Here, $g(n)$ is lower bound of function $f(n)$

Eg:- $16n^2 + 8n^2 + 2$

$24n + 9$

n^3	$f(n) = \Omega(n^3)$
n	$f(n) = \Omega(n)$



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

It provides the best-case complexity of an algorithm.

(iii) theta (Θ) :-

- $f(n) = \Theta . g(n)$, if there exists a positive integer 'n_0' & 2 - positive constant 'c_1 & c_2 such that:

$$c_1 . g(n) \leq f(n) \leq c_2 . g(n)$$

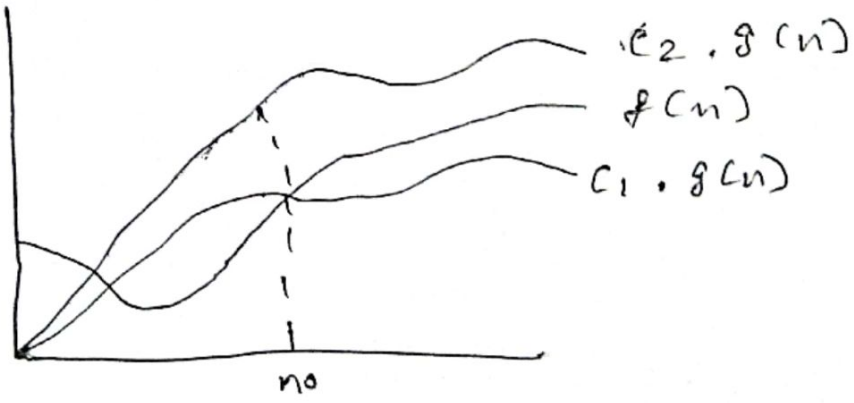
$\forall$ $n \geq n_0$

- The function $g(n)$ is both upper & a lower - bound for function $f(n)$ for all values of 'n',

$n \geq n_0$

Eg:-

$f(n)$	$g(n)$	
$6n^2 + 3n^2 - 9n$	n^2	$f(n) = \Theta(n^2)$
$7.2^n + 3n$	2^n	$f(n) = \Theta(2^n)$



$f(n) = \Theta . g(n)$

- Theta(Θ) notation encloses from above & below.

- Since it represents the upper & the lower bound of the running time of an algorith it is used for analysing the average-case complexity of an algorithm.

⟹ Time - Space Trade off :-

- It is a way of solving a problem or calculation in less time by using more storage space (memory) or by solving a problem in very little space by spending a long time.
- It is a case where an algorithm / program trades increased space usage with decreased time.
- Here, space refers to the data storage consumed in performing a given task.
- Time refers to the time consumed in performing a given task.

↳ Need of Time - Space Tradeoff :

- Less time by using more memory.
- By solving a problem in very little space by spending a long time.

Eg:- If data is stored uncompressed, it take- more space but less time.

- Storing only the source & rendering it as an image everytime the page is request would be trading time for space. More time used but less space.

- Best case :- The minimum possible value of f(n).
- Average case :- The expected value of f(n).
- Worst case :- The maximum value of f(n) for any key possible i/p.

Eg:- Time complexity of merge sort is O(n log n)
Quick sort has O(n log n) time complexity for best & average case.

↳ Types of Time-Space Tradeoff :-

i) compressed & uncompressed data :- A space-time tradeoff can be applied to the problem of data storage,

- If data is stored uncompressed, it takes more space but less time,
- If the data is stored compressed, it takes less space but more time to run the decompression algorithm.

ii) Re-rendering & Stored image :- Storing only the source & rendring it as an image everytime the page is requested would be trading time for space,

- storing the images would be trading space for time. More space used but less time.

iii) Smaller code (loop) & Larger code (without loop) :-

- Smaller code occupies less space in memory but it requires high computation time which is required for jumping back to the beginning of loop at the end of each iteration.
- Larger code can be traded for higher program speed. It occupies more space in memory but requires less computation time. (no jumping & loop)

iv) Look-up table & Recalculation :-

- In look-up table, an implementation can include the entire table which reduces computing time but increases the amount of memory needed.
- It can recalculate i.e, compute table entries as needed, increasing computing time but reducing memory requirements.

Eg:-

more time, less space

```
int a, b;
Printf("enter value");
Scanf("%d", &a);
Printf("enter value");
Scanf("%d", &b);
b = a + b;
Printf("output is %d",
        b);
```

less time, more space

```
int a, b, c;
Printf("enter value of a, b, c");
Scanf("%d %d %d", &a, &b, &c);
Printf("output is %d",
        c = a + b);
```

→ Preminiaries :- A algorithm is simply a set of instructions that step by step define how a work is to be executed upon to get the expected results.

- The appropriate algorithm that can be determin based on no of factors ;
  - How long the algorithm can takes to run.
  - What resources are required to execute the algorithm.
  - How much space (or) memory is required.
  - How exact is the solution provided by the algorithm.
  - Algorithm should be easy to understand code & debugging.

↳ Mathematical notations & functions :-

- Various mathematical functions which appear very often in the analysis of algorithm & in computer science in general, together with their notations are as follows ;

i) Floor & ceiling functions : Let x may be any real number !

$\lfloor x \rfloor$ - Floor of 'x' denotes the greatest integer that does not exceed 'x',

$\lceil x \rceil$ - ceil of 'x' denotes the least integer that is not less than 'x',

- If 'x' is a integer then $\lfloor x \rfloor = \lceil x \rceil$.
  otherwise $\lfloor x \rfloor + 1 = \lceil x \rceil$.

Eg :- Floor : $\lfloor 5.3 \rfloor = 5$, $\lfloor \sqrt{10} \rfloor = 3$, $\lfloor 2.9 \rfloor = 2$, $\lfloor -2.9 \rfloor = -3$

ceil : $\lceil 5.3 \rceil = 6$, $\lceil \sqrt{10} \rceil = 4$, $\lceil 3.1 \rceil = 4$, $\lceil 3.0 \rceil = 3.0$

ii) Remainder function; modular arithmetic :
Let 'K' be any integer & let M be a positive integer, then $K \pmod M$ will denote the integer remainder when 'K' is divided by M.

Eg :- $25 \pmod 7 = 4$,
$25 \pmod 5 = 0$,
$35 \pmod{11} = 2$.

iii) Integer & absolute values functions:
Let 'x' be any real no. The integer value of x, written INT(x), converts 'x' into an integer by deleting the factorial Part of no.

Eg: INT(3.14) = 3,
INT() = 2,
INT(-8.5) = -8.

- The absolute value of the real no 'x' written as ABS(x), is defined as the greatest of 'x' or '-x'.
- ABS(x) = x or -x for $x \neq 0$
- ABS(0) = 0 for $x = 0$

Eg: [-15] = 15,
ABS[-3.33] = 3.33.

iv) Summation symbol; sums: Consider a sequence $a_1, a_2, a_3 \ldots a_n$ then summation of this series can be denoted as:
$a_1 + a_2 + \ldots + a_n$ and $a_m + a_{m+1} + \ldots a_n$, respectively by: $\sum_{i=1}^{n} a_i = \frac{N*(N+1)}{2}$

v) Factorial function: The product of the positive integers from 1 to n. inclusively, is denoted by n! (or) $\underline{n}$
- $n! = 1*2*3 \ldots *(n-2)(n-1)n$

Eg: $4! = 1 \times 2 \times 3 \times 4 = 24$

vi) Permutations: Its of a set of 'n' elements is an arrangements of the elements in a given order.
- If there are 'n' elements then there are n! permutations.

Eg: Consider 3 alphabets a, b c with there, there can be $3! = 3 \times 2 \times 1$ permutation, they are: abc; acb, bac, bca, cab, cba

VII) Exponent & Logarithms:

- Exponent refers to no of times a number is multiplied by itself.

Eg: $a^m = a * a * \dots$ (m times)

$a^0 = 1$, $a^m = \frac{1}{a^m}$, $a^{m/n} = \sqrt[n]{a^m} = \left[\sqrt[n]{a}\right]^m$

- Logarithms refers to how many times a certain number called the base is multiplied by itself to reach another no.

Eg: Let 'b' is positive no, the logarithm of any +ve no 'x' to the base 'b' be written as $\log_b x$.

- $y = \log_b x$ is equivalent to $b^y = x$.
- $\log_2^8 = 3$, since $2^3 = 8$.

  $\log_{10} 100 = 2$, since $10^2 = 100$

↳ Algorithmic notations:- The format for the formal presentation of an algorithm consists of 2 parts:

- 1st part is a paragraph which tells the purpose of the algorithm.
- 2nd part of the algorithm consists of the lists of steps that is to be executed.

i) Identifying numbers: Each algorithm is assigned an identifying no.

ii) steps, control, exit: The steps of the algorithm are executed one after other, beginning with step-1.

- control may be transferred to step-n of the algorithm by the statement "Go to step-n".
- The algorithm is completed when statement Exit is encountered.

iii) Comments: Each step may contain a comment in brackets which indicates the main purpose of the step.

iv) Variable names: It will use capital letters, as MAX & DATA.

v) Assignment statement: It will use the dots-equal notation := that is used in Pascal.

vi) Input & output: Data may be i/p & assigned to variables by means of a read statement with following form: Read: variable names

- Data in variables may be output by means of a write or print statement with the following form;
  write: message.

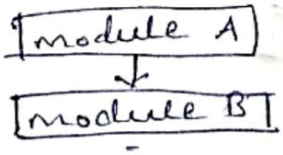
vii) Procedures: It will be used for an independent algorithm module which ~~Provide~~ solves a particular problem.

2) Control structures:-

- Control structures are just a way to specify flow of control in programs.
- Algorithms & their equivalent programs are more easily understood if they mainly use control structures.
- It basically analyses & chooses in which direction a program flows based on certain conditions.

(i) Sequential logic (sequential flow):- It follows a serial/sequential flow in which the flow depends on series of instructions given to the computer.

- modules are executed sequentially.



(ii) Selection logic (conditional flow):- It simply involves a no of conditions which decides one out of several written modules.

- It has 3 types:

(a) Single alternative: if (condition) then:
[module-A]
[End of if structure]

(b) Double alternative : if (condition), then:
[module A]
else
[module B]
[end of if structure]

(c) multiple alternatives:
if (condition A), then:
[module A]
Else if (condition B), then:
[module B]
=
Else if (condition N), then
[module N]
[End if structure]

(iii) Iteration logic (Repetitive flow) :- It employs a loop which involves a repeat statement followed by a module known as the body of a loop.

a) Repeat - for structure:

Eg: Repeat for i = A to N by I;
[module]
[End of loop]

A - Initial value, N is end value
I - Increment.

b) Repeat - while structure:

Eg: Repeat while condition;
[module]
[End of loop]

⟹ String processing :-

- A string is a sequence of characters.
- The class "String" includes methods for examining individual characters, comparing strings, searching characters (strings), extracting parts of strings & for converting an entire string uppercase & lowercase.
- String is always defined inside double quotes (" ")
- String processing tasks can be devided into detecting, locating, extracting or replacing patterns in strings.

↳ **Storing strings :-** Strings are stored in 3 types of structures;

i) **Fixed length structures (storage) :-** (Record oriented fixed length storage) In this each line of Print is viewed as record, where all have the same length i.e, each record accomodates same (fixed) no. of characters.

- char str[25] ⟶ 0 to 24.
- This means that 25 memory locations are allocated for string.

* **Advantage :-**

- Ease of accessing data from any given record,
- Updation is easy.

* **Disadvantages :-**

- memory is not utilised properly,
- Not extend the space,
- Time is wasted.

ii) **Variable length storage with fixed maximum**

- The storage of variable length strings in memory cells with fixed lengths can be don in 2 general ways:

a. one can use a marker, such as two dollar signs ($$), to signal the end of the string.

b. one can list the length of the string - as an additional item in the pointer array.

* Advantages :-

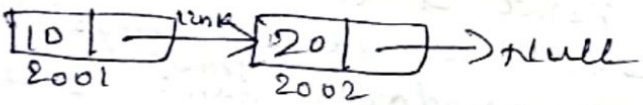
- Save space in memory,
- strings are relatively permanent.

* Disadvantage :-

- It is inefficient when the strings & their length are frequently being changed.

ii) Linked storage :-

- Linked list is a linear collection of data elements called nodes & the linear order is given by means of pointer.

Eg:- 

* Advantage :-

- storage representation is efficient (good).
- modification, insertion & deletion operations are easier.

* Disadvantages :-

- Extra memory is required for the link files.

↳ string as ADT :-

- ADT consists of a set of values, a defined set of properties of these values, and a set of operations for processing the values.
- The string ADT values are all sequences of characters upto a specified length.
- With the string ADT, you get to decide the implementation.

* Properties:-

- The component characters are form of ASCII character set.
- They are comparable in lexicographic order.
- They have a length, from '0' to specified length.

* Operations:-

- Input
- output.
- Initialization & Assignment.
- comparision greater, equal, less.
- Determination of length.
- Concatenation.
- Accessing component characters & substrings

* functions:-

- string newString(char *str); :- creates a new string.
- void destroyString(string s); :- releases (deletes) all resources associated with a string.
- int stringLength(string s); :- Get the length of a string.
- string stringConcat(string s1, string s2); :- Given two string s, concatenates them into a new string.
- int stringsEqual(string s1, string s2); :- checks whether two strings are equal or not.
- int stringFindChar(string s, char find); :- Given a string & a character to look for in the string, return either the index of the character, or -1 if the character could not be found in the string.

↳ String operations :-

i) **Substring :-** Accessing a substring from a given string requires 3 pieces of information
- Name of the string (or) string itself.
- Position of 1st character of substring.
- Length of the substring (or) position of the last character of the substring.

Syntax :- Substring (string, initial, length).

Eg :- substring ('The end', 4, 4) = '□End'.

ii) **Indexing :-** Also called pattern matching, refers to finding the position where a string pattern 'P' 1st appears in a given string text 'T'.

Syntax :- Index (text, pattern). "His father is the professor"

Eg :- Index (T, 'The') = 7

iii) **Concatenation :-** Let S_1 & S_2 be strings, that the concatenation of S_1 & S_2, which we denote by $S_1 // S_2$, is the string consisting of the characters of S_1 followed by the characters of S_2.

Eg :- S_1 = 'Chethan', S_2 = 'kumar'
$S_1 // S_2$ = 'Chethankumar'.

iv) **Length :-** The no of characters in a string for the length of a given string.

Eg :- S = 'computer'
length (S) = 8

v) **Uppercase :-** Replaces every lowercase string into uppercase from a given text.

Eg :- T : chethan ⟹ CHETHAN.

vi) **COPY :-** copies one string to another string.

Syntax :- source . copy (destination, sizeof source);

Eg :- str1. copy(str2, len);

vii) **compare :-** It compares the value of 2 string.

Syntax :- int K = str1 . compare(str2);

1> Word/Text Processing :-

- In earlier times, character data processed by the computer consisted mainly of data items, such as names & addresses.
- Today the computer also processes printed matter, such as letters, articles & reports.
- It is in this latter context that we use the term "word processing".
- Operations :-

i) **Replacement :-** This operation involves replacing one string in the text by another.

- Suppose in a given text 'T', the 1st occurance pattern 'P_1' should be replaced by a pattern 'P_2'. It is denoted by.

Syn :- REPLACE (text, pattern1, pattern2)

Eg :- REPLACE (T, P_1, P_2)

REPLACE ('RAMESH', 'M', 'J')

o/p :- RAJESH.

ii) **Insertion :-** involves inserting a string in the middle of the text.

syntax :- INSERT (text, position, string)

Eg :- INSERT (T, K, S)

INSERT ("Yogabatt", 5, 'raj')

o/p :- 'Yogarajbatt'.

iii) **Deletion :-** Deletion operation involves deleting a string from the text.

Syntax :- DELETE (text, Position, length)

Eg :- DELETE (T, K, L)

Eg :- DELETE ('Yogarajbatt', 1, 4)

o/p :- "rajbatt".

↳ Pattern Matching algorithms:-

- Pattern matching is the problem of deciding whether or not a given string Pattern 'P' appears in a string text 'T'.
- The length of 'P' does not exceed the length of 'T'.
- There are different algorithms, the main goal to design these type of algorithms to reduce the time complexity.
- The traditional approach may take lots of time to complete the Pattern searching task for a longer text.
- Here some of the different algorithms to get a better performance of Pattern matching:

1. Brute-force algorithm.
2. Boyer-moore algorithm.
3. Rabin-Karp algorithm.
4. KMP (Knuth-morris-Pratt) algorithm.
5. Naive Pattern searching algorithm.
6. String Pattern matching with finite automata.

① Brute - force algorithm :-

- Brute force string match (T[0 -- n-1], P[0 -- m-1])

```
{
for i ← 0 to n-m do
{
  j ← 0
  while j < m and P[j] = T[i+j] do
    j++
  if j = m then return i
}
return -1
}
```

- The outer loop is executed at most n - m + 1 times,
- the inner loop 'm' times, for each iteration of the outer loop,
- the running time of this algorithm is in O(nm).
- The simplest algorithm for string matching is a brute force algorithm, where we try to match the 1st character of the pattern with 1st character of the text, & if we succeed, try to match the 2nd character & so on.

Eg: T: W I S E M A N
P: E M A

n = 7
m = 3
n - m = 4

i = 0
j = 0

$P[j] = m[i+j]$ $m=3$,

$i=0$, $j=0$ ($j<m$)

$0<3$ & $P[0] = m[0+0] = E \neq W$ ✗ j+1

$j=1$, $1<3$ & $P[1] = m[0+1] = M \neq I$ ✗ j+1

$j=2$, $2<3$ & $P[2] = m[0+2] = A \neq S$ ✗ i+1

$i=1$, $j=0$

$0<3$ & $P[0] = m[1+0] = E \neq I$ ✗ j+1

$j=1$ → $1<3$ & $P[1] = m[1+1] = M \neq S$ ✗ j+1

$j=2$ → $2<3$ & $P[2] = m[1+2] = A \neq E$ ✗ i+1

$i=2$, $j=0$

$0<3$ & $P[0] = m[2+0] = E \neq S$ ✗ j+1

$j=1$, $1<3$ & $P[1] = m[2+1] = M \neq E$ ✗ j+1

$j=2$, $2<3$ & $P[2] = m[2+2] = A \neq M$ ✗ i+1

$i=3$, $j=0$

$0<3$ & $P[0] = m[3+0] = E = E$ ✓ j+

$j=1$, $1<3$ & $P[1] = m[3+1] = M = M$ ✓ j+

$j=2$, $2<m$ & $P[2] = m[3+2] = A = A$ ✓ i=3

Eg:- T: 0 1 2 3 4 5 6 7 8 9 10 11 12
O P P O R T U N I T I E S

P: 0 1 2 3
U N I T

$n=13$

$m=4$

$n-m=9$

$i=0$, $j=0$ ($j<m$)

$0<4$ & $P[0] = m[0+0] = U \neq O$ ✗ j+1

$j=1$, $1<4$ & $P[1] = m[0+1] = N \neq P$ ✗ j+1

$j=2$, $2<4$ & $P[2] = m[0+2] = I \neq P$ ✗ j+1

$j=3$, $3<4$ & $P[3] = m[0+3] = T \neq O$ ✗ i+1

$i=1$, $j=0$

$0<4$ & $P[0] = m[1+0] = U \neq P$ ✗ j+1

$j=1$, $1<4$ & $P[1] = m[1+1] = N \neq P$ ✗ j+1

$j=2$, $2<4$ & $P[2] = m[1+2] = I \neq O$ ✗ j+1

$j=3$, $3<4$ & $P[3] = m[1+3] = T \neq R$ ✗ i+1

$i=2$, $j=0$

$0<4$ & $P[0] = m[2+0] = U \neq P$ ✗ j+1

$j=1$, $1<4$ & $P[1] = m[2+1] = N \neq O$ ✗ j+1

$j=2$, $2<4$ & $P[2] = m[2+2] = I \neq R$ ✗ j+1

$j=3$, $3<4$ & $P[3] = m[2+3] = T = T$ ✓

i=3, j=0: 0<4 & P[0] = m[3+0] = U ≠ O ✗ j+1
j=1, 1<4 & P[1] = m[3+1] = N ≠ R ✗ j+1
j=2, 2<4 & P[2] = m[3+2] = I ≠ T ✗ j+1
j=3, 3<4 & P[3] = m[3+3] = T ≠ U ✗ i+1

i=4, j=0: 0<4 & P[0] = m[4+0] = U ≠ R ✗ j+1
j=1, 1<4 & P[1] = m[4+1] = N ≠ T ✗ j+1
j=2, 2<4 & P[2] = m[4+2] = I ≠ U ✗ j+1
j=3, 3<4 & P[3] = m[4+3] = T ≠ N ✗ i+1

i=5, j=0: 0<4 & P[0] = m[5+0] = U ≠ T ✗ j+1
j=1, 1<4 & P[1] = m[5+1] = N ≠ U ✗ j+1
j=2, 2<4 & P[2] = m[5+2] = I ≠ N ✗ j+1
j=3, 3<4 & P[3] = m[5+3] = T ≠ I ✗ i+1

i=6, j=0: 0<4 & P[0] = m[6+0] = U = U ✓ j+1
j=1, 1<4 & P[1] = m[6+1] = N = N ✓ j+1
j=2, 2<4 & P[2] = m[6+2] = I = I ✓ j+1
j=3, 3<4 & P[3] = m[6+3] = T = T ✓

i=6

(2) Boyer-moore algorithm:-

- It is a searching algorithm in which a string of length 'n' & a 'Pattern of length 'm' is searched.
- It prints all the occurences of the pattern in the text.
- In this algorithm, we start to check characters from the right & then move to the left.
- 2 strategies:
  - Bad match table
  - Good suffix table

- Boyer-moore algorithm runs in time:

$O(nm + S)$

- Algorithm:

```
Boyer moore match(T, P, Σ)
(L ← last occurence function(P, Σ))
i ← m - 1
j ← m - 1
repeat
    if T[i] = P[j]
    if j = 0
    return i {match at i}
    else
        i ← i - 1
        j ← j - 1
    else
        {character - jump}
        (l ← L[T[i]])
        i ← i + m - min(j, 1+l)
        j ← m - 1
    end if
until i > n - 1
return -1 {no match}
```

Eg: Value = length(P) - index(P) - 1

T : WISEMAN
P : EMA

char	E	M	A	*
badchar	2	1	3	3

value = length(P) - index(P) - 1
E = 3 - 0 - 1 = 2
M = 3 - 1 - 1 = 1

Eg: TEAM MATE (n=8)

0	1	2	3	4	5	6	7
T	E	A	M	M	A	T	E

Value = length(P) - index(P) - 1
T = 8 - 0 - 1 = 7
E = 8 - 1 - 1 = 6
A = 8 - 2 - 1 = 5
M = 8 - 3 - 1 = 4
M = 8 - 4 - 1 = 3
A = 8 - 5 - 1 = 2
T = 8 - 6 - 1 = 1
E = 8 - 7 - 1 = 0

Eg: SOCCER (n=6)

0	1	2	3	4	5
S	O	C	C	E	R

S = value = 6 - 0 - 1 = 5
O = 6 - 1 - 1 = 4
C = 6 - 2 - 1 = 3
C = 6 - 3 - 1 = 2
E = 6 - 4 - 1 = 1
R = 6 - 5 - 1 = 0 → come as value - 6 because n = 6

S	O	C	E	R	*
5	4	2	1	6	6

T	E	A	M	*
1				

Eg:- T: WELCOME to sahil world
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 n=19

P: sahil
0 1 2 3 4 m=5

- **Bad match table :-** rule

s	a	h	i	l	*
4	3	2	1	5	5

value = length(P) − index(P) − 1

s = 5 − 0 − 1 = 4
a = 5 − 1 − 1 = 3
h = 5 − 2 − 1 = 2
i = 5 − 3 − 1 = 1
l = 5 − 4 − 1 = 0 → come as value 6 because m = 5

0 1 2 3 4 5
→ WELCOME TO SAHIL WORLD
SAHIL

0 1 2 3 4
→ WELCOME TO SAHIL WORLD
4
SAHIL

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
→ WELCOME TO SAHIL WORLD
SAHIL

∴ 9th posith 'P' in 'T'.

- **Good suffix :-** rule

i = m − 1	j = m − 1	P[j] = T[i]
= 5 − 1	= 5 − 1	P[0] = T[4]
= 4	= 4	l ≠ 0

j=0, i=4	P[j] = T[i]	
	P[0] = T[4]	
	s ≠ 0	(i+1)
j=0, i=5	P[j] = T[i]	
	P[0] = T[5]	
	s ≠ m	(i+1)
j=0, i=6	P[j] = T[i]	
	P[0] = T[6]	
	s ≠ e	(i+1)
j=0, i=7	P[j] = T[i]	
	P[0] = T[7]	
	s ≠ t	(i+1)
j=0, i=8	P[j] = T[i]	
	P[0] = T[8]	
	s ≠ 0	(i+1)
j=0, i=9	P[j] = T[i]	
	P[0] = T[9]	
	s = s ✓	

j = 1, i = 10	P[j] = T[i]
	P[1] = T[10]
	a = a ✓
j = 2, i = 11	P[j] = T[i]
	P[2] = T[11]
	h = h ✓
j = 3, i = 12	P[j] = T[i]
	P[3] = T[12]
	i = i ✓
j = 4, i = 13	P[j] = T[i]
	P[4] = T[13]
	l = l ✓

∴ i = 9, the Pattern 'P' found in 9th position of text 'T'.

(3) Rabin - Karp algorithm :-

- It makes use of hash functions & the rolling hash technique.
- The purpose of hash is to take a large piece of data & be able to represented by a smaller form.
- The best & average case running - time of algorithm is $\boxed{O(m+n)}$.
  n - length of text,
  m - length of word.
- Algorithm :

```
Rabin Karp (T, P)
n = T.length
m = P.length
L^P = hash(P[--])
L^T = hash(T[--])
for s=0 to n-m
if (L^P == L^T)
if (P=[0 -- m-1])
T =([s+0 -- +s+m-1])
Print "pattern found at shift s"
if (s < n-m)
L^T = hash(T(s+1 -- s+m))
```

Eg:- T: c c a c c a a e d b a (1 2 3 4 5 6 7 8 9 10 11) n = 11
P: d b a (1 2 3) m = 3

T
a - 1
b - 2
c - 3
d - 4
e - 5
f - 6
g - 7
h - 8
i - 9
j - 10

P = d b a
= 4 + 2 + 1
= 7 (hash value) } → Spurious hit (when value match but not a pattern)

cca → 3 + 3 + 1 = 7 } → Spurious hit
cac → 3 + 1 + 3 = 7 } → Spurious hit
acc → 1 + 3 + 3 = 7 } → Spurious hit

- [Rolling hash function] → to overcome Spurious hit

dba = 4 + 2 + 1 = 7
cca = 3 + 3 + 1 = 7 ≠ pattern
cac = 3 + 1 + 3 = 7 now, 7 - 1 = 6 } 1st term.

(so, on)

dba $\Rightarrow P[1] \times 10^{m-1} + P[2] \times 10^{m-2} +$
421 $P[3] \times 10^{m-3}$

$= 4 \times 10^2 + 2 \times 10^1 + 1 \times 10^0$

$= 400 + 20 + 1$

$= 421$ (hash code / value)

cca $= 3 \times 10^2 + 3 \times 10^1 + 1 \times 10^0$
331

$= 300 + 30 + 1$

$= 331$ → now take rolling hashing!

cac $= 331 - 3 \times 10^2 \Rightarrow 31 \times 10$

$= 310 + 3$ ~~$\times 100$~~

$= 313$

acc $= 313 - 3 \times 10^2$

$= 313 - 300$

$= 13 \times 10$

$= 130 + 3$

$= 133$

cca $= 133 - 1 \times 10^2$

$= 133 - 100$

$= 33 \times 10$

$= 330 + 1$

$= 331$

caa $= 331 - 300$

$= 31 \times 10$

$= 310 + 1$

$= 311$

aae $= 311 - 300$

$= 11 \times 10$

$= 110 + 5$

$= 115$

aed $= 115 - 100$

$= 15 \times 10$

$= 150 + 4$

$= 154$

edb $= 154 - 100$

$= 54 \times 10$

$= 540 + 2$

$= 542$

dba $= 542 - 500$

$= 42 \times 10$

$= 420 + 1$

$= 421$

a – 1
b – 2
c – 3
d – 4
e – 5
f – 6
g – 7
h – 8
i – 9
j – 10
k – 11
l – 12
m – 13
n – 14
o – 15
p – 16
q – 17
r – 18
s – 19
t = 20

(00)

P: dba = 4 + 2 + 1 = 7 mod %/o 3

T: cca = 3 + 3 + 1 = 7 %. 3

= 1

(any Prime no)

P: dba = 421 mod 11 = 3

cca = 331 mod 11 = 1 (x)

cac = 313 mod 11 = 5

acc = 133 mod 11 = 1

cca = 331 mod 11 = 1

caa = 311 mod 11 = 3

aae = 115 mod 11 = 5

aed = 154 mod 11 = 0

edb = 542 mod 11 = 4

dba = 421 mod 11 = 3

Eg: T: c c a c c a a e d b a e g h j i d

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17

P: g h j i

1 2 3 4

n = 17

m = 4

(any Prime no)

ghji = 7 + 8 + 10 + 9 = 34 %. 11 = 1

ccac = 3 + 3 + 1 + 3 = 10 mod 11 = 10

cacc = 3 + 1 + 3 + 3 = 10 mod 11 = 10

acca = 1 + 3 + 3 + 1 = 8 mod 11 = 8

ccaa = 3 + 3 + 1 + 1 = 8 mod 11 = 8

caae = 3 + 1 + 1 + 5 = 10 mod 11 = 10

aaed = 1 + 1 + 5 + 4 = 11 mod 11 = 0

aedb = 1 + 5 + 4 + 2 = 12 mod 11 = 1

edba = 5 + 4 + 2 + 1 = 12 mod 11 = 1

dbae = 4 + 2 + 1 + 5 = 12 mod 11 = 1

baeg = 2 + 1 + 5 + 7 = 15 mod 11 = 4

aegh = 1 + 5 + 7 + 8 = 21 mod 11 = 10

eghj = 5 + 7 + 8 + 10 = 30 mod 11 = 8

ghji = 7 + 8 + 10 + 9 = 34 mod 11 = 1

hjid = 8 + 10 + 9 + 4 = 31 mod 11 = 9

---

(4) KMP (Knuth - Morris - Pratt) algorithm:-

- KMP algorithm is used to find a "Pattern" in a "text"
- This algorithm compares character by character from left to right. But whenever a mismatch occurs, it uses a prefix table to skip characters comparision while matching.
- Time complexity - $O(n+m)$

(i) Prefix function(n) : compute Prefix function(n)

1) $m \leftarrow$ length[P] // 'P' - pattern to be matched
2) $\Pi[1] \leftarrow 0$
3) $K \leftarrow 0$
4) for $q \leftarrow 2$ to m
5) do while $K > 0$ and $P[K+1] \neq P[q]$
6) do $K \leftarrow \Pi[K]$
7) If $P[K+1] = P[q]$
8) then $K \leftarrow K+1$
9) $\Pi[q] \leftarrow K$
10) return Π

- In the above pseudo code for calculating the prefix function, the for loop from step - (4) to (10) runs 'm' times.
- Hence the running time of computing Prefix function is $O(m)$.

Eg: P: a b a b a c a (1 2 3 4 5 6 7)

m = length[P] = 7

$\Pi[1] = 0$, $K = 0$

$q = 2$, $K = 0$ → $\Pi[2] = 0$

$q = 3$, $K = 0$ → $\Pi[3] = 1$

$q = 4$, $K = 1$ → $\Pi[4] = 2$

$q = 5$, $K = 2$ → $\Pi[5] = 3$

$q = 6$, $K = 3$ → $\Pi[6] = 0$

$q = 7$, $K = 1$ → $\Pi[7] = 1$

q	1	2	3	4	5	6	7
P	a	b	a	b	a	c	a
Π	0	0	1	2	3	0	1

(i) KMP-Matcher(T, P):

1) $n \leftarrow$ length[T]
2) $m \leftarrow$ length[P]
3) $\Pi \leftarrow$ compute-Prefix-function(P)
4) $q \leftarrow 0$ // no of characters matched
5) for $i \leftarrow 1$ to n // scan 'S' from left to right
6) do while $q > 0$ and $P[P+1] \neq T[i]$
7) do $q \leftarrow \Pi[q]$ // next character does not match
8) If $P[q+1] = T[i]$
9) then $q \leftarrow q+1$ // next character matches
10) If $q = m$ // is all of P matched?
11) then print "pattern occurs with shift" $i - m$
12) $q \leftarrow \Pi[q]$

- The for loop beginning in step-(5) runs 'n' times, Thus running time of the matching function is $O(n)$.

Eg: T: a b a b c a b c a b a b a b d
P: a b a b d

- (Π:-table)

	1	2	3	4	5
	a	b	a	b	d
	0	0	1	2	0

prefix	sufix
a	d
ab	b d
aba	a b d
a b a b	b a b d
a b a b d	a b a b d

-

1	2	3	4
a	b	a	b
0	0	1	2

-

1	2	3	4	5	6	7	8
a	b	c	d	a	b	c	y
0	0	0	0	1	2	3	0

-

1	2	3	4	5	6	7	8	9	10
a	b	c	d	a	b	c	a	b	f
0	0	0	0	1	2	3	1	2	0

	1	2	3	4	5	6	7	8	9	10	11
	a	b	c	d	e	a	b	f	a	b	c
	0	0	0	0	0	1	2	0→1	2	3	

after "0" starts with "1".

	1	2	3	4	5	6	7	8	9	10	11
	a	a	a	b	c	a	d	a	a	b	e	a
	0	1	1	0	0→1	0	→1	1	2	0→1	

	1	2	3	4	5	6	7	8	9
	a	a	a	a	b	a	a	c	d
	0	1	1	1	0	1	1	0	0

Tracing :-

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
T :	a	b	a	b	c	a	b	c	a	b	a	b	a	b	d
P :	a	b	a	b	d										

	1	2	3	4	5
	a	b	a	b	d
	0	0	1	2	0

i = a
j = 0

j = 0+1 = 1

0

j

1's character & j+1
(0 & 1) → 1 (a)

i = b	i = a	i = b	i = c
j = 1 ✓	j = 2	j = 3	j = 4 (d)

i = 0
j = 0

i = b
j = 1

T: b a c b a b a b a b a c a c a

P: a b a b a c a

	1	2	3	4	5	6	7
	a	b	a	b	a	c	a
	0	0	1	2	3	0→1	

	1	2	3	4	5	6
	a	a	a	a	b	c
	0	1	2	3	0	0

T: b a d b a b a b a d a a b
P: a b a b a d a
0 1 2 3 4 5 6

n = 15
m = 7

	1	2	3	4	5	6	7
P	a	b	a	b	a	d	a
π	0	0	1	2	3	0→1	

LPS - longest prefix suffix

⇒ (ab)
a ≠ b

(aba)
a, ab, ~~aba~~ (prefix)
a, ba, ~~aba~~ (suffix)
match length: (1)

(abab)
a, ab, aba (P)
b, ab, bab (S)
match (2) [LPS]

⇒ (ababa)
a, ab, aba, abab (P)
a, ba, aba, baba (S)
match length (3) [LPS]

(ababad)
a, ab, aba, abab, ababa
d, ad, bad, abad, babad
(NO - match = 0)

⇒ (ababada)
a, ab, aba, abab, ababa, ababad (P)
a, da, ada, bada, abada, babada (S)
match length = 1 [LPS]

T: b a n a n a n a n o
0 1 2 3 4 5 6 7 8 9
P: n a n o
0 1 2 3

n = 10
m = 4

i = 0
j = n

	1	2	3	4
P	n	a	n	o
π	0	0	1	0

T: a a a b a b a a b a a b a b a a b
P: a a b a b

	1	2	3	4
a	a	b	a	b
0	1	0	1	2

(aba)
a, ab (P)
a, ba
(1) → LPS

(abab)
a, ab, aba (P)
b, ab, bab (S)
2

* (bab)
b, ba (P)
b, ab (S)

	0	1	2	3	4	5	6	7	8	9	10	
• T:	a	b	a	b	a	b	a	c	a	b	a	n=11

P: a b a b a c a m=7

	1	2	3	4	5	6	7
P	a	b	a	b	a	c	a
Π	0	0	1	2	3	0 → 1	

0 → starts from 0 (a)

---

⑤ **Naive algorithm:-** It is the simplest method among other pattern searching algorithms.

- It finds one or all exact occurences of a pattern in a text.
-

- **Algorithm:-**

Naive-String-matcher (T, P)

1) n ← length[T]
2) m ← length[P]
3) for s ← 0 to n-m
4) do if P[1--m] = T[s+1--s+m]
5) then Print "pattern occurs with shift".

- This 'for' loop from (3) to (5) executes for n-m+1 times & in iteration we are doing 'm' comparisions.
- Total complexity is $\boxed{O(n-m+1)}$

Eg:- T:

0	1	2	3	4	5	6	7	8	
a	b	c	a	b	a	a	b	c	n=9

P:

0	1	2	3	
a	b	a	a	m=4

s=0, n-m = 9-4 = 5

$s=0$, $j=0$

~~$P[s+j] = T[j]$~~

$P[j] = T[s+j]$

$P[0] = T[0+0] = a = a$ ✓ $j+1$

$j=1$, $P[1] = T[0+1] = b = b$ ✓ $j+1$

$j=2$, $P[2] = T[0+2] = a \neq c$ ✗ $s+1$

$s=1$, $j=0$ $P[0] = T[1+0] = a \neq b$ ✗ $s+1$

$s=2$, $j=0$ $P[0] = T[2+0] = a \neq c$ ✗ $s+1$

$s=3$, $j=0$ $P[0] = T[3+0] = a = a$ ✓ $j+1$

$j=1$, $P[1] = T[3+1] = b = b$ ✓ $j+1$

$j=2$, $P[2] = T[3+2] = a = a$ ✓ $j+1$

$j=3$, $P[3] = T[3+3] = a = a$ ✓ $s=3$

∴ Pattern occurs with shift $s=3$

↳ String Pattern matching with finite automata :-

Finite automata $[T, \delta, m]$

1) $n \leftarrow$ length $[T]$
2) $q \leftarrow 0$
3) for $i \leftarrow 1$ to n
4) do $q \leftarrow \delta(q, T[i])$
5) if $q = m$
6) then Print "Pattern occurs with shift" : $-m$

- If the finite automation is available, the algorithm needs only $O(n+m)$ time.

eg :- T : a b a b a b a c a b a (0 1 2 3 4 5 6 7 8 9 10) $n = 11$

P : a b a b a c a (0 1 2 3 4 5 6) $m = 7$

length of $P + 1 = 7 + 1 = 8$

δ	a	b	c
0	(1)	0	0
1	1	(2)	0
2	(3)	0	0
3	1	(4)	0
4	(5)	0	0
5	1	4	(6)
6	(7)	0	0
7	1	2	0

a : a - aa → a = a → 1

ab : b - abb → ab ≠ bb → 0
c - abc → ab = bc → 0

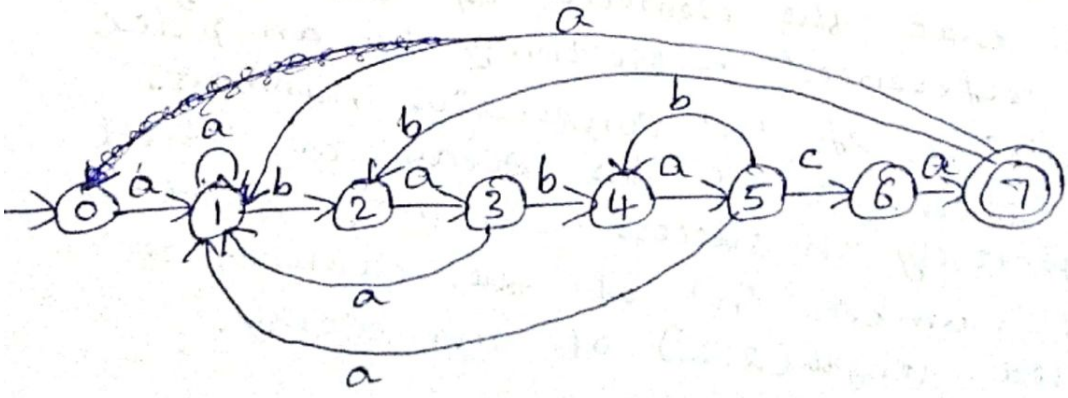
aba : a - abaa → aba = baa → 1
c - abac → aba ≠ bac → 0

ababa : b - ababb → abab ≠ babb → 0
c - ababc → abab ≠ babc → 0

abababa : a - abababa → ababa = babaa → 1
b - abababab → ababab = bababab → 4

ababaca : b - abababacb → ababac ≠ babacb → 0
c - abababacc → ababac ≠ babacc → 0

a - abababacaa → ababaca = babacaa → 1
b - abababacab → ababaca = babacab → 2
c - abababacac → ababaca ≠ babacac → 0



states : 0 1 2 3 4 5 6 7
a b a b a c a
↓ ↓ ↓ ↓ ↓ ↓ ↓
1 2 3 4 5 6 7

Unit - 2
ARRAYS

- Array is a homogeneous datatype, which has group of elements with same name & same datatype.

Types:
i) one - dimensional array.
ii) Two - dimensional array.
iii) multi - dimensional array.

i) one - dimensional array:- It has only one sub-script (size).

Syntax:- datatype arrayname[size];
Eg:- int a[10];

ii) Two - dimensional array:- It has two subscript rows & column size.

Syntax:- datatype arrayname[row size][column size];
Eg:- int a[2][3];

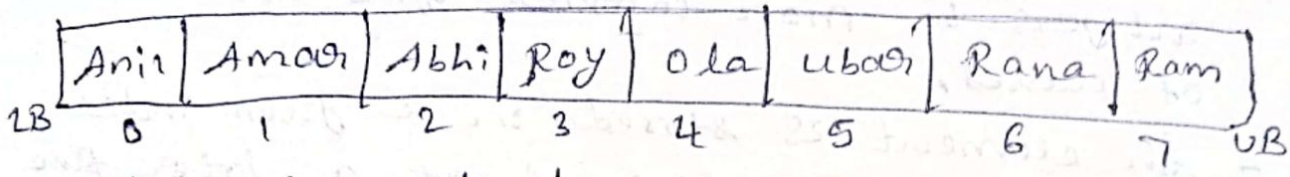
iii) Multi - dimensional array:- It has more than 2 - subscripts.

Syntax:- datatype arrayname[s1] --[sn];
Eg:- int a[2][3][4];

⟹ Linear arrays:-

- A linear array is a list of finite no of 'n' homogeneous data elements such that the elements of the array are referenced respectively by an index consisting of 'n' consecutive numbers.
- The elements of the array are stored respectively in successive memory location.
- The number 'n' of the elements is called length(size) of an array.

Eg:- Consider an array of employee names with index set from 0 to 7, which contains 8 elements:



- Now, if want to retrieve the name - 'Roy' from the given array, we can do so by calling "employee[3], It gives the element stored in that location, i.e, 'Roy'.
- We can calculate the length (or) a total no of elements of a linear array (LA) by:

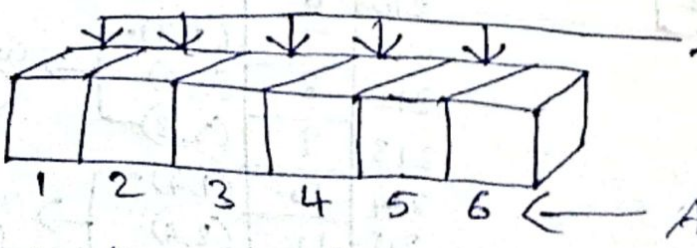
$$\begin{aligned} \text{Length}(LA) &= UB - LB + 1 \\ &= 7 - 0 + 1 \\ &= 8 \end{aligned}$$

* Operations:-

- Traversing:- processing each element of array list.
- Inserting:- Adding new elements in the array list.
- Deleting:- Removing an element from the array list.
- Sorting:- Arranging the elements of the list in some sorting order.
- Merging:- combining the elements of two array lists in a single array list.

↳ Array as ADT:-

- Array is an abstract datatype that holds a collection of elements accessible by an index.



(one-dimensional array with 6 elements)

- The elements stored in an array can be anything from primitives types such as integers to more complex types like instances of classes.
- An element is stored in a given index & they can be retrieved at a later time by specifying the same index.
- The array (ADT) have one property, they store & retrieve elements using an index.

(2) Arrays have vastly different functionality across various implementations:

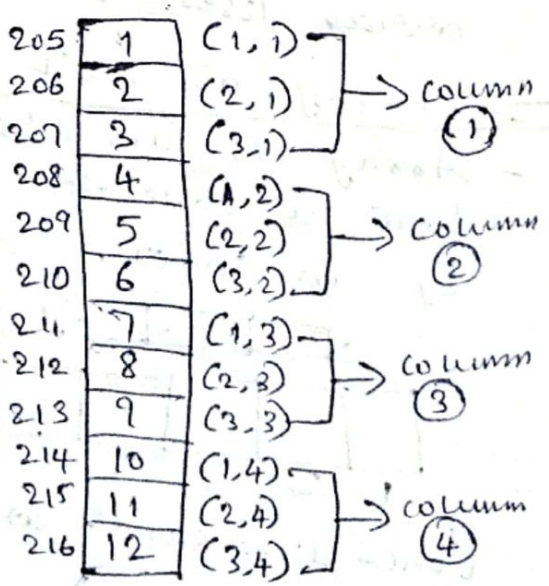
Function name	Provided functionality
set (i, V)	sets the value of index i to V
get (i)	Returns the value of index 'i'.

Representation of linear arrays in memory :-

- The process to determine the address in a memory.

i) Column major order :- The elements of the array are stored column by column ie, column elements, then next column element & so on.

	1	2	3	4
1	1	4	7	10
2	2	5	8	11
3	3	6	9	12



- Base address Base(A) = 205. word length w = 1. Find address of element 9 in array A.
- m = 3, n = 4
- $Loc(A[J,K]) = Base(A) + w[m(K-1) + (J-1)]$

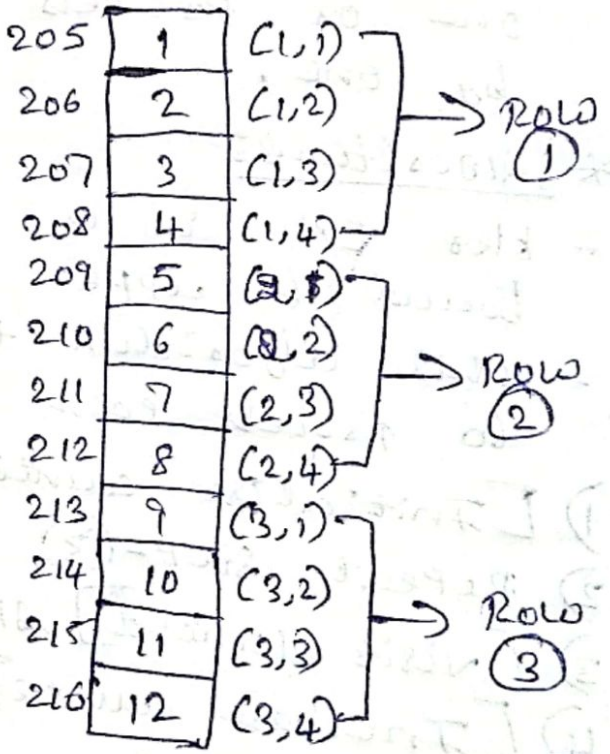
$Loc(A[3,3]) = 205 + 1[3(3-1) + (3-1)]$

$= 205 + (6+2)$

$= 213$

ii) Row major order :- The elements of the array are stored row by row.
i.e, 1st row elements, then next row elements & so on.

	1	2	3	4
1	1	4	7	10
2	2	5	8	11
3	3	6	9	12



- $Loc(A[J,K]) = Base(A) + w[N(J-1) + (K-1)]$

$Loc(A[3,3]) = 205 + 1[4(3-1) + (3-1)]$

$= 205 + 1[4(2) + (2)]$

$= 205 + 1[8+2]$

$= 205 + 10$

$= 215$ (11) data

* Advantages :-

- Used to represent stacks & queues in memory,
- Used for matrix manipulation.
- Easy to access elements in memory location.

* Disadvantages :-

- Time consuming.
- Memory space is waste.

⟹ Traversing linear array :-

- Print all the array elements one by one or process the each element one by one.

* Algorithms :-

- Her LA is a linear array with lower bound & upper bound - UB.
- This algorithm traverse LA applying as to process each element of LA.

1) [Initialize counter] set K := LB
2) Repeat step-(3) & (4) while K ≤ UB
3) [Visit element] APPLY PROCESS to LA[K].
4) [Increase counter] set K := K+1.
   [End of step-2 loop]
5) Exit.

(Alternative method)

- (Traversing a linear array) This algorithm traverses a linear array (LA) of LB & UB.

1) Repeat for K = LB to UB.
   APPLY PROCESS to LA[K].
   [End of loop]
2) Exit.

- The complexity of traversal in Linear array algorithm :
- Step-1 is executed once, so it contributes 1 to complexity function $f(n)$.
- Step-2 is a loop control step that executes step-3 & step-4 N times (once for each element of array data having 'n' elements.

So, $f(n) = 2 * n + 1$

(or)

$f(n) = 2n + 1$

$O(n)$

Algorithm for inserting linear array :-

- [Inserting in to a linear array]
  INSERT(LA, N, K, ITEM).
- Here, LA is an linear array with N-elements & 'K' is +ve integer such that $K \leq N$.
- This algorithm inserts an element ITEM at K^{th} position in LA.

1) [Initialize counter] set J = N
2) Repeat steps (3) & (4) while J ≥ K.
3) [move J^{th} element downward] set LA[J+1] = LA[J]
4) [Decrease counter] set J = J - 1
   (end of step-2 loop)
5) [Insert element] set LA[K] = ITEM
6) [Reset N] set N = N + 1
7) Exit.

O(n)

Eg:-

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

LA[N] = LA[7]

N = 7, K = 5, ITEM = 5

(counter)

J = 7, 7 ≥ 5 ✓

LA[8] = LA[7]

LA[8] = 8

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

J = 6, 6 ≥ 5 ✓

LA[7] = LA[6]

LA[7] = 7

J = 5, 5 ≥ 5 ✓

LA[6] = LA[5]

LA[6] = 6

J = 4, 4 ≥ 5 ✗

LA[5] = 5

N = N + 1

N = 7 + 1

N = 8

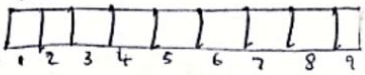
Eg:-

1	2	3	4	5	6	7	8	9
A	B	F	G	Y	I	J	M	P

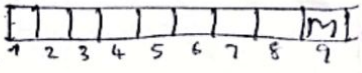
N = 9, K = 3, ITEM = C

J = N

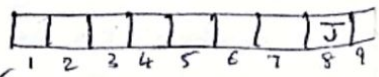
- J = 9, 9 ≥ 3 ✓
  LA[10] = LA[9]
  LA[10] = P



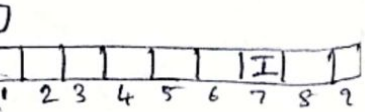
- J = 8, 8 ≥ 3 ✓
  LA[9] = LA[8]
  LA[9] = M



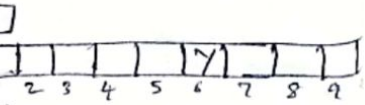
- J = 7, 7 ≥ 3 ✓
  LA[8] = LA[7]
  LA[8] = J



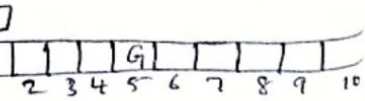
- J = 6, 6 ≥ 3 ✓
  LA[7] = LA[6]
  LA[7] = I



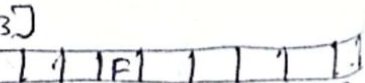
- J = 5, 5 ≥ 3 ✓
  LA[6] = LA[5]
  LA[6] = Y



- J = 4, 4 ≥ 3 ✓
  LA[5] = LA[4]
  LA[5] = G



- J = 3, 3 ≥ 3 ✓
  LA[4] = LA[3]
  LA[4] = F



- J = 2, 2 ≥ 3 ⊗
  ~~LA[3] = LA[2]~~
  LA[3] = C

N = 9 + 1 = 10

1	2	3	4	5	6	7	8	9	10
A	B	C	F	G	Y	I	J	M	P

Algorithm for deleting linear array :-

- (Deleting from an linear array) DELETE (A, N, K, ITEM)
- Here Linear array with 'N' elements & 'K' is a +ve integer such that $K \leq N$.
- This algorithm deletes K^{th} element from LA.

1) Set ITEM = LA(K)
2) Repeat for J = K to N-1
   [move J+1st element upward] set:
   LA[J] = LA[J+1]
   [End of loop]
3) Reset the no 'N' of element in LA:
   set N = N-1.
4) Exit.

O(n)

Eg:

1	2	3	4	5	6	7
1	2	3	4	5	6	7

8 = LA[3]

K=3, N = 7-1 ⇒ 6

J = 3 to 6 ✓
LA[3] = LA[3+1]
LA[3] = 4

J = 4 to 6 ✓
LA[4] = LA[4+1]
LA[4] = 5

J = 5 to 6 ✓
LA[5] = LA[5+1]
LA[5] = 6

J = 6 to 6 ✓
LA[6] = LA[6+1]
LA[6] = 7

J = 7 to 6 ✗

⇒ N = 7-1 = 6

1	2	3	4	5	6
1	2	4	5	6	7

Eg:

1	2	3	4	5	6	7	8	9
A	B	F	G	Y	I	J	M	P

K = 5, item = Y

9 = LA[5], N = 9-1 = 8

J = 5 to 8 ✓
LA[5] = LA[5+1]
LA[5] = I

J = 6 to 8 ✓
LA[6] = LA[6+1]
LA[6] = J

J = 7 to 8 ✓
LA[7] = LA[7+1]
LA[7] = M

J = 8 to 8 ✓
LA[8] = LA[8+1]
LA[8] = P

J = 9 to 8 ✗

⇒ N = 9-1 = 8

1	2	3	4	5	6	7	8
A	B	F	G	I	J	M	P

(N-dimensional)

Multi-dimensional array :-

$A(1 -- m_1)(1 -- m_2) -- (1 -- m_n)$

subscript like,

$A(K_1, K_2, K_3, K_4 -- K_n)$

DL for string array is :

1) column-major representation.
2) Row-major representation.

→ An array with each dimension has UB & L the length of each subscript is calculated as

$L_i = UB - LB + 1$

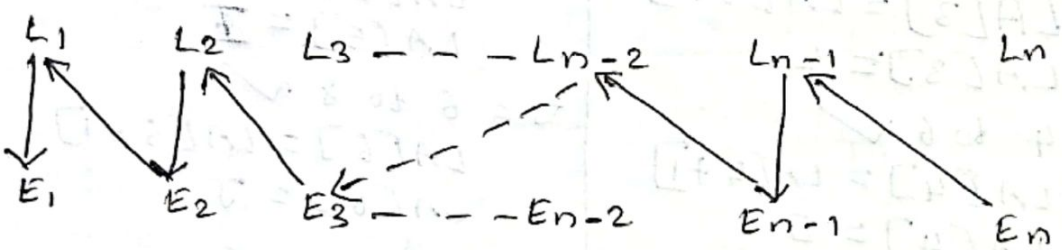
→ For a given index K_i, the effective inde E_i is given by :

$E_i = K_i - LB$

→ Address of $A(K_1, K_2 -- K_n)$ calculated as

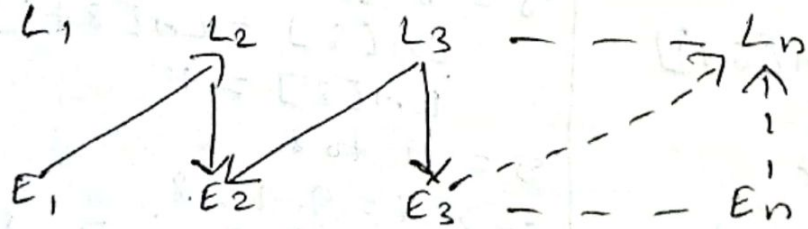
1) column-major representation :

$$Loc(A(K_1, K_2 -- K_n)) = B + w((((Enter-1)L_{n-2}) + + E_3)L_2 + E_2)L_1 + E_1$$



2) Row-major representation :

$$Loc(A(K_1 K_2 -- K_n) = B + w(((E_1 L_2 + E_2)L_3 + E_3)L_4 + -- + E_{n-1})L_n + E_n)$$



W - Size of element
B - Base of address
L - Length of element
E - Effective index address

Eg problem: 3D array, MAZE is declared using MAZE[2, 8, -4, 1, 6, 10]
find no of elements
$\Rightarrow L_i = (UB - LB + 1)$

MAZE is memory 2M representation.

Length find:
(-4, 2, 7, 9, -3, 5)
7 × 3 × 9 = 189
L_1 L_2 L_3

$L_1 = 7$
$L_2 = 6$
$L_3 = 5$
$7 \times 6 \times 5 = 210$

Base(MAZE) = 200 w = 4
add of (MAZE) is, MAZE[5, -1, 8] is
$E_i = K_i - LB$
$E_1 = 5 - 2 = 3$
$E_2 = -1 + 4 = 3$
$E_3 = 8 - 6 = 2$.

- Formula for Row-major representation:

$E_1 L_2 = 3 \times 6 = 18$
$E_1 L_2 + E_2 = 18 + 3 = 21$
$(E_1 L_2 + E_2) L_3 = 21 * 5 = 105$
$(E_1 L_2 + E_2) L_3 + E_3 = 105 + 2 = 107$

- Loc of MAZE[5, -1, 8)
$B + w(E_1 L_2 + E_2) L_3 + E_3$
$200 + 4(107)$
$200 + 428 = 628$

→ **Matrix:-** used to represent numbers in rows & columns & used in computer graphic to display 2-dimension.

$C_{ij} = C_{ij} + A[i][k] * B[k][j]$

- Basic algorithm:

```
for (int i=0, i<n, i++)
{
 for (int j=0, j<=n, j++)
{
 C[i][j]=0;
  for (int k=0; k<n, k++)
{
 (c[i][j] = c[i][j] + A[i][k] * B[k][j]);
}}}
```

matrix is a 2-dimension array, an M×N matrix is an array of M-N nos arranged in 'M' rows & 'N' columns

- Time complexity: $\sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{k=1}^{n} C \quad C \cdot n^3$

↳ Sparse matrix :- It has more zero elements & less non-zero elements.

- It is used to traversing & 3D representa

Eg :-

	C_0	C_1	C_2	C_3
R_0	6	1	0	0
R_1	5	0	4	0
R_2	0	2	0	3
R_3	0	3	0	0

4 × 4

	Rows	columns	Value
	4 (row)	4 (column)	6 (no. of non-zero)
R_0	0	1 (C. of non-zero)	1 (which zero)
R_1	1	2	9 → 5+
R_2	2	2	5 → 2+
R_3	3	1	3
C_0	0	1	5
C_1	1	3	6 → 1+2+
C_2	2	1	4
C_3	3	1	3

Eg :-

	C_0	C_1	C_2	C_3	C_4
R_0	0	0	4	0	5
R_1	0	0	0	3	6
R_2	0	0	0	2	0
R_3	0	2	3	0	0

Rows	0	1	1	2	3	3
columns	2	3	4	3	1	2
Value	4	3	6	2	2	3

↳ Triangular matrix :-

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

↳ Trigonal matrix :- (Tridiagonal)

1	-2	0	0	0
1	-1	3	0	0
0	5	0	-1	0
0	0	-2	1	4
0	0	0	1	3

↳ Bubble-sort :- It is also known as interchange sort, sinking sort or shifting sort which sorts by comparing each adjascent pair of items in a list & swaping the items if necessary & repeating the pass through list until no swaps are done.

* Algorithm :-

(Bubble sort) BUBBLE (DATA, N)
Here, DATA is an array with 'N' elements, this algorithm sorts elements & data.

1) Repeat step (2) & (3) for K=1 to N-1.
2) set PTR = 1 [Initialises pass pointer PTR]
3) Repeat while PTR ≤ N-K [Execute pass]
   a) If DATA[PTR] > DATA[PTR+1], then interchange DATA[PTR] & DATA[PTR+1]
      [End of if structure]
   b) set PTR = PTR+1
   [End of inner loop]
   [End of step-1 outer loop]
4) Exit.

* Time complexity :-

Worst case : O(n*n)
Average case : O(n*n)
Best case : O(n)

* Space complexity :- O(1).

$= (N-1) + (n-2) + (N-3) - - - + 2 + 1$
$= (N) * (N-1)/2$
$= O(N^2)$

Eg: 32, 51, 27, 85, 66, 23, 13, 57

(1 2 3 4 5 6 7 8)

N = 8, K = 1 to N-1

⇒ 1 to 7

Ptr = 1, 1 ≤ N-K ⇒ 1 ≤ 7 ✓

DATA[PTR] > DATA[Ptr + 1]

P[1] > D[1+1]

32 > D[2]

32 > 51 ✗

⇒ Ptr = 1+1 ⇒ 2

2 ≤ 7 ✓

D[2] > D[2]

51 > 27

32	27	51	85	66	23	13	57
1	2	3	4	5	6	7	8

⇒ Ptr = 2+1 ⇒ 3

= 3 ≤ 7 ✓

D[3] > D[4]

51 > 85 ✗

⇒ Ptr = 3+1 = 4

= 4 ≤ 7 ✓

D[4] > D[5]

85 > 66 ✓

32	27	51	66	85	23	13	57
1	2	3	4	5	6	7	8

⇒ Ptr = 4+1 = 5

5 ≤ 7 ✓

D[5] > D[6]

85 > 23 ✓

32	27	51	66	23	85	13	57
1	2	3	4	5	6	7	8

⇒ Ptr = 5+1 = 6

6 ≤ 7 ✓

D[6] > D[7]

85 > 13 ✓

32 27 51 66 23 13 85

$\Rightarrow$ ptr = 7 , $7 \leq 7$

D[7] > D[8]

85 > 57 ✓

32 27 51 66 23 13 57 85

$\Rightarrow$ ptr = 8, $8 \leq 7$ ✗ so increment Pass (2)

Pass-2 K = 2 to 7
N = 8

ptr = 1,

ptr $\leq$ N - K

$1 \leq 6$ ✓

D[1] > D[2] = 32 > 27 ✓

27 32 51 66 23 13 57 85

ptr = 2, $2 \leq 6$ ✓

D[2] > D[3] = 32 > 51 ✗

ptr = 3, $3 \leq 6$ ✓

D[3] > D[4] = 51 > 66 ✗

ptr = 4, $4 \leq 6$ ✓

D[4] > D[5] = 66 > 23 ✓

27 32 51 23 66 13 57 85

Ptr = 5, $5 \leq 6$ ✓

D[5] > D[6] = 66 > 13 ✓

27 32 51 23 13 66 57 85

Ptr = 6 , $6 \leq 6$ ✓

D[6] > D[7] = 66 > 57 ✓

27 32 51 23 13 57 66 85

ptr = 7, $7 \leq 6$ ✗

Pass-3 $K = 3$ to 7 , $N = 8$

$ptr = 1$, $ptr \leq N - K \Rightarrow 1 \leq 5$ ✓

$D[1] > D[2] = 27 > 32$ ✗

$ptr = 2$, $2 \leq 5$ ✓

$D[2] > D[3] = 32 > 51$ ✗

$ptr = 3$, $3 \leq 5$

$D[3] > D[4] = 51 > 23$ ✓

27 32 23 51 13 57 66 85

$ptr = 4$ $4 \leq 5$

$D[4] > D[5] = 51 > 13$ ✓

27 32 23 13 51 57 66 85

$ptr = 5$ $5 < 5$

$D[5] > D[6] = 51 > 57$ ✗

$ptr = 6$ $6 < 5$ ✗

Pass-4 $K = 4$ to 7, $N = 8$

$ptr = 1$, $ptr \leq N - K \Rightarrow 1 \leq 4$ ✓ ($8 - 4$)

$D[1] > D[2] = 27 > 32$ ✗

$ptr = 2$, $2 \leq 4$

$D[2] > D[3] = 32 > 23$ ✓

27 23 32 13 51 57 66 85

$ptr = 3$, $3 \leq 4$

$D[3] > D[4] = 32 > 13$ ✓

27 23 13 32 51 57 66 85

$ptr = 4$, $4 \leq 4$

$D[4] > D[5] = 32 > 51$ ✗

$ptr = 5$, $5 \leq 4$ ✗

Pass-5 $K = 5$ to 7, $N = 8$

$ptr = 1$, $ptr \le N - K \Rightarrow 1 \le 3$

$D[1] > D[2] = 27 > 23$ ✓

23 27 13 32 51 57 66 85

$ptr = 2$, $2 \le 3$

$D[2] > D[3] = 27 > 13$ ✓

23 13 27 32 51 57 66 85

$ptr = 3$, $3 \le 3$

$D[3] > D[4] = 27 > 32$ ✗

$ptr = 4$, $4 \le 3$ ✗

Pass-6 $K = 6$ to 7, $N = 8$

$ptr = 1$, $ptr \le N - K \Rightarrow 1 \le 2$

$D[1] > D[2] = 23 > 13$ ✓

13 23 27 32 51 57 66 85

$ptr = 2$, $2 \le 2$

$D[2] > D[3] = 23 > 27$ ✗

$ptr = 3$, $3 \le 2$ ✗

∴ 13 23 27 32 51 57 66 85

Eg:

1	2	3	4	5
47	50	-2	85	-17

$N = 5$, $K = 1$ to $N - 1 \Rightarrow 1$ to 4

$ptr = 1$, $ptr \le N - K \Rightarrow 1 \le 4$

$D[1] > D[2] = 47 > 50$ ✗

$ptr = 2$, $2 \le 4$

$D[2] > D[3] = 50 > -2$ ✓

47 -2 50 85 -17

$ptr = 3$, $3 \le 4$

$D[3] > D[4] = 50 > 85$ ✗

Ptr = 4, 4 ≤ 4 ✓

D[4] > D[5] = 85 > -17 ✓

47 -2 50 -17 85

Ptr = 5, 5 ≤ 4 ✗

Pass-2 K = 2 to 4, N = 5

Ptr = 1, Ptr ≤ N - K ⇒ 1 ≤ 3 ✓

D[1] > D[2] ⇒ 47 > -2 ✓

-2 47 50 -17 85

Ptr = 2, 2 ≤ 3 ✓

D[2] > D[3] = 47 > 50 ✗

Ptr = 3, 3 ≤ 3 ✓

D[3] > D[4] = 50 > -17 ✓

-2 47 -17 50 85

Ptr = 4, 4 ≤ 3 ✗

Pass-3 K = 3 to 4, N = 5

Ptr = 1, 1 ≤ 2 ✓

D[1] > D[2] = -2 > 47 ✗

Ptr = 2, 2 ≤ 2

D[2] > D[3] = 47 > -17 ✓

-2 -17 47 50 85

Ptr = 3, 3 ≤ 2 ✗

∴ -2 -17 47 50 85

Pass-4 K = 4 to 4, N = 5

Ptr = 1, Ptr ≤ N - K ⇒ 1 ≤ 1 ✓

D[1] > D[2] = -2 > -17 ✓

-17	-2	47	50	85

Linear Search :-

Search in sequential manner over all items one by one. Every item is checked & if a match is found, then that particular item is returned otherwise search continues until end of data collection.

- A linear array DATA with 'N' elements of a specific ITEM of info is given.
- Algorithm finds the location LOC of ITEM in the array DATA / sets LOC = 0

1) [Initialize] set K = 1 and LOC = 0
2) Repeat step (3) & (4) while LOC = 0 & K ≤ N
3) If ITEM = DATA[K], then set LOC = K
4) set K = K + 1 [Increment counter]
   [end of step-2 loop]
5) [Successful !]
   If LOC = 0, then
   write ITEM is with in array DATA
   else
   write LOC is location of ITEM.
   [end of if structure]
6) Exit.

* Time complexity :-

Best case : O(1)
Worst case : O(n)
Average case : O(n)

* Space complexity :-

O(n)

$$f(n) = 1 \cdot 1/n + 2 \cdot 1/n + \dots + n \cdot 1/n + (n-1) = 0$$
$$= (1 + 2 + \dots + n) \cdot 1/n$$
$$= n(n+1)/2 \cdot 1/n$$
$$= (n+1)/2$$

1 2 3 4 5 6 7

Eg: 10, 20, 30, 40, 50, 60, 70

$\Rightarrow K = 1,\ 1 \leq 7$ ✓

ITEM = 50

50 = D[1]

50 ≠ 10 ✗

$\Rightarrow K = 2,\ 2 \leq 7$ ✓

50 = D[2]

50 ≠ 20 ✗

$\Rightarrow K = 3,\ 3 \leq 7$ ✓

50 = D[3]

50 ≠ 30 ✗

$\Rightarrow K = 4,\ 4 \leq 7$ ✓

50 = D[4]

50 ≠ 40 ✗

$\Rightarrow K = 5,\ 5 \leq 7$ ✓

50 = D[5]

50 = 50 ✓ $\Rightarrow$ LOC = 5

(other method): (Linear search) Linear (DATA, N, ITEM, LOC)

- Here, data is in linear array with 'n' elements, & a item is a given ITEM of info, This algorithm finds the location of item in data:

1) [Insert ITEM at the end of DATA] set data[n+1] = item + 1
2) [initialize counter] set LOC = 1
3) [search for ITEM]
   Repeat while DATA[LOC] ≠ ITEM
   set LOC = LOC + 1
   [End of LOOP]
4) [Successfull!] →
   if LOC = N + 1, then
   set LOC = 0
5) exit.

Worst case complexity of linear search:

$O(n)$

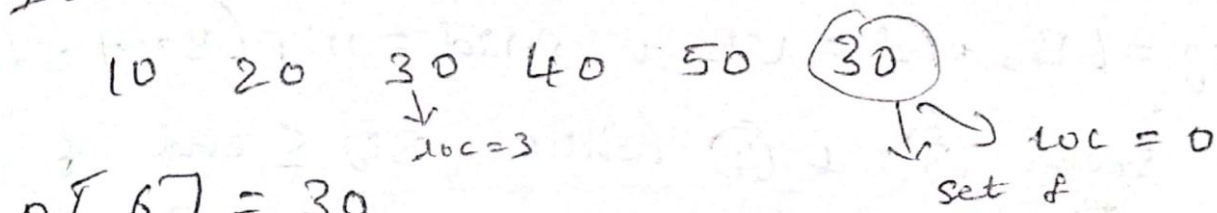
$C(n) = n$

Avg case

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

Eg :- (10, 20, 30, 40, 50)

Item = 30



D[6] = 30

$\Rightarrow$ loc = 1

D[1] $\neq$ item $\Rightarrow$ 10 $\neq$ 30

$\Rightarrow$ loc = 2

D[2] $\neq$ item $\Rightarrow$ 20 $\neq$ 30

$\Rightarrow$ loc = 3

D[3] = item $\Rightarrow$ 30 $\neq$ 30

Binary Search :-

1) [Initialise segment variable]
   Set Beg = LB, end = UB & Mid = INT (Beg + end / 2)
2) Repeat steps (3) & (4) while Beg ≤ end & Data[MID] ≠ ITEM.
3) if ITEM < DATA[MID] then
   set END = MID − 1
   else
   set BEG = MID + 1
   [End of if structure]
4) set MID = INT [BEG + END / 2]
   [End of step-2 loop]
5) if data[MID] = item, then
   set LOC = MID
   else
   set LOC = null.
   [End of if structure]
6) Exit.

- Here, Data is sorted array with LB & UB & item is a given ITEM of information, the variable Beg, End, mid denote respectively. The BIG, MID, END, loc of segments of element of data. then algorithm found loc of item in DATA are set loc = null.
- Binary search is a fast search algorithm with runtime complexity of $O(\log n)$ $f(n) = \lceil \log 2n \rceil + 1$
- It works on the principle of divide & conquer, for this algorithm, data collection should be in sorted form.
- It looks for a particular item by comparing the middle most item, if match occurs, then the index of item is returned.

Eg:- 11 (1), 22 (2), 30 (3), 33 (4), 40 (5), 44 (6), 55 (7), 60 (8), 66 (9), 77 (10), 80 (11), 88 (12), 99 (13)

Item = 40

Beg = 1, end = 13, mid = int $\frac{(1+13)}{2} = \frac{14}{2} = 7$

$\Rightarrow$ beg $1 \leq$ end 13, D[7] $\neq$ item.

$55 \neq 40$ → -1

$\Rightarrow 40 < 55$ ✓ End = mid − 1

End = 6, Beg = 1, mid = $\frac{1+6}{2} = \frac{7}{2} = 3.5 = 3$

beg end

$1 \leq 6$, D[3] $\neq$ item

$30 \neq 44$

$\Rightarrow 40 < 30$ ✗ beg = mid + 1 (3 + 1)

Beg = 4, end = 6, mid = $\frac{4+6}{2} = \frac{10}{2} = 5$

beg end

$4 \leq 6$, D[5] $\neq$ item

$40 = 40$

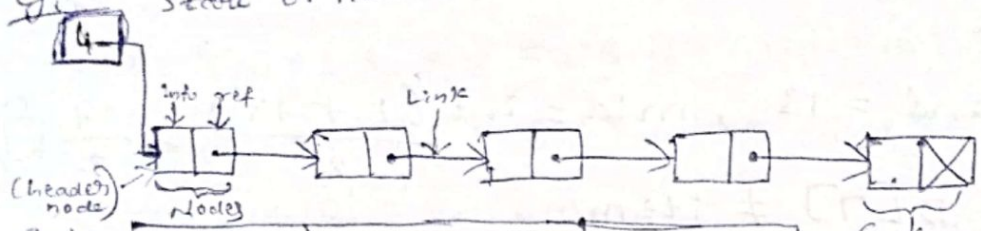
D[5] = item

40 = 40

Loc = mid = 5

→ Singly Linked List :-

Eg:- Start or name



Eg:-

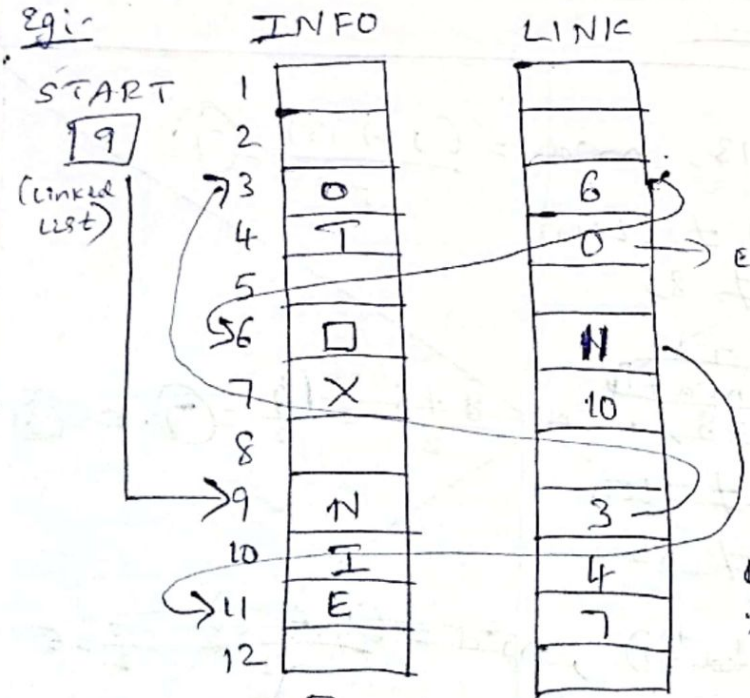
Bed no	Patient	Next
1	Kark	7
2		
3	Dean	11
4	maxwell	12
5	Adams	3
6		
7	Lane	4
8	Green	1
9	Samuels	0
10		
11	Fields	8
12	Nelson	9

START [5]

Link

- Reallocation of Linked-List in memory

Eg:-



LINK [10] = 4

LINK [11] = 7

INFO [7] = 10 → To add 'X' add

START [9] = N, INFO [9] = N is 1st character

LINK [9] = 3, INFO [3] = O is 2nd character

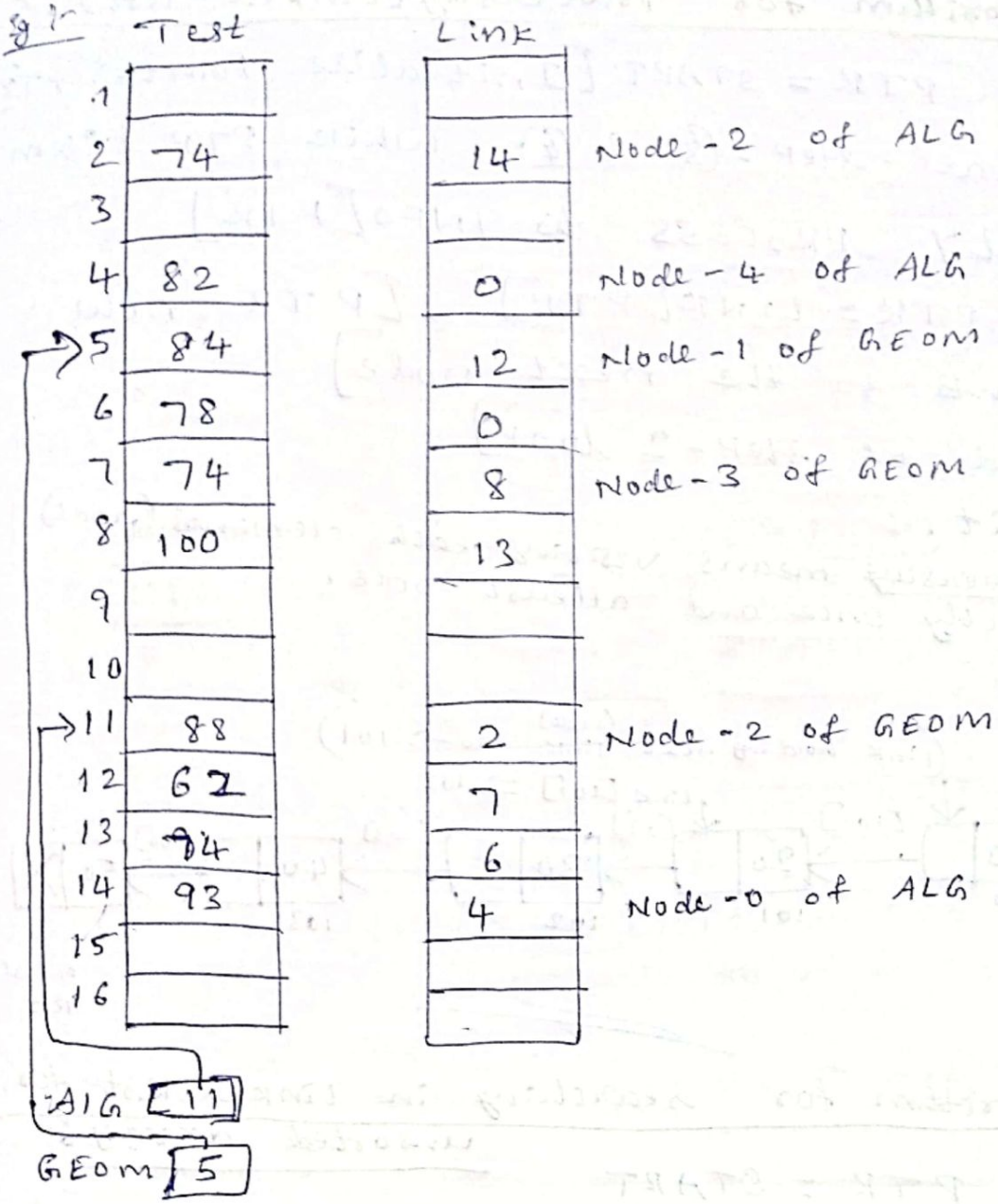
LINK [3] = 6, INFO [6] = □ is 3rd character

LINK [6] = 11, INFO [11] = E is 4th character

LINK [11] = 7, INFO [7] = X is 5th character

LINK [7] = 10, INFO [10] = I is 6th character.

Sir

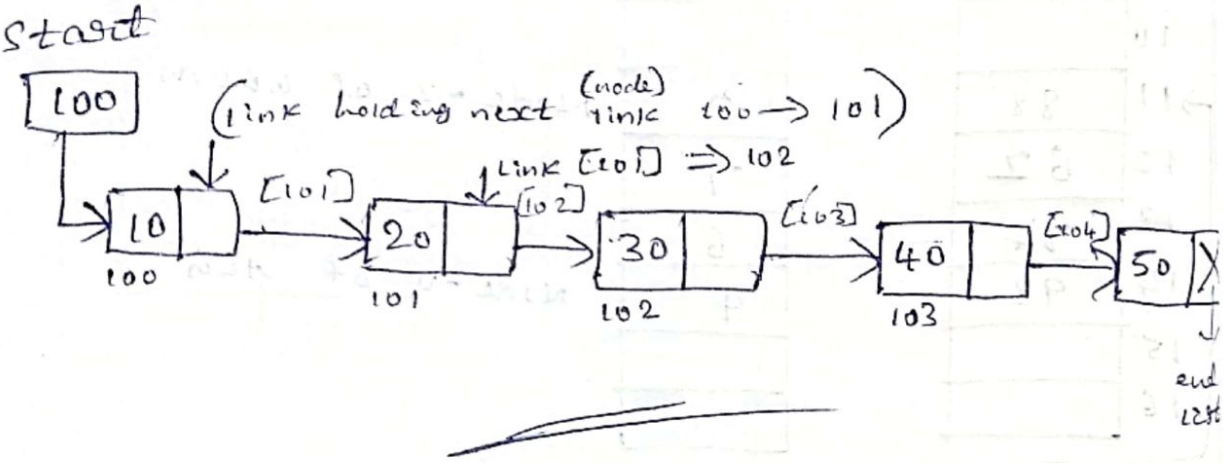


- A linked list is a linear data structure in which the elements are linked using pointers.
- Linked list consists of nodes where each node contains a data field & a link (reference) to the next node in the list.
- Types:
  - Simple linked list : forward only
  - Doubly linked list : forward & backward
  - Circular linked list : Last item contains link of the 1st element as next, the 1st element has a link to the last element as previous.

↳ Algorithm for Traversing (Linked - list):

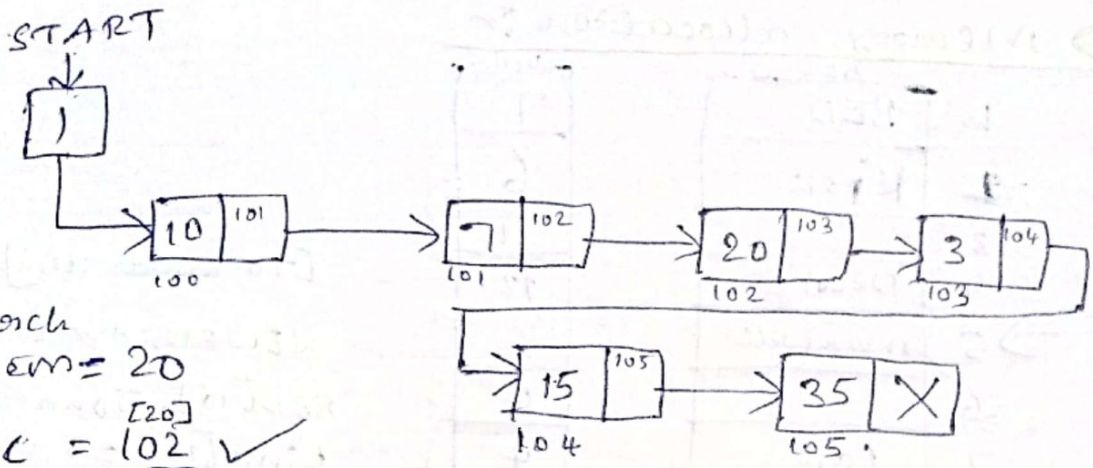
1) Set PTR = START [Initialise Pointer p
2) Repeat step - (3) & (4) while PTR ≠ ite
3) APPLY PROCESS to INFO[PTR]
4) Set PTR = LINK[PTR] . [PTR now points to the next node]
   [End of step-2 loop]
5) Exit.

- Traversing means visiting each elements (node) exactly once and atleast once.



↳ Algorithm for searching in linked list fo unsorted array or

1) Set PTR = START
2) Repeat step-3 while PTR ≠ NULL
3) if ITEM = INFO[PTR], then:
   Set LOC = PTR and Exit
   Else
   Set PTR = LINK[PTR] . [PTR now points to next node]
   [End of if structure]
4) [Search is unsuccessful!] set LOC = NUL
5) Exit

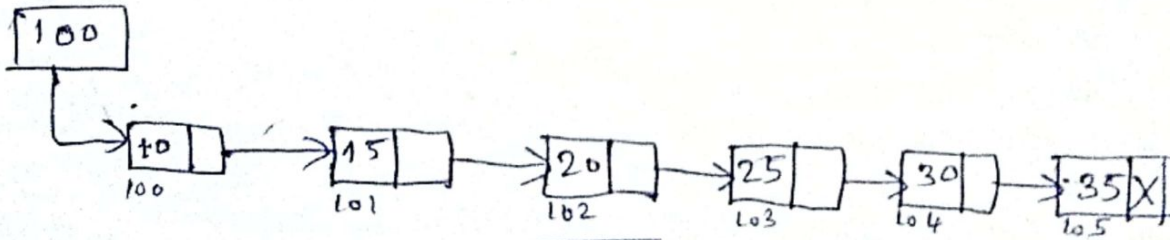


Search
Item = 20
[20]
LOC = 102 ✓
found

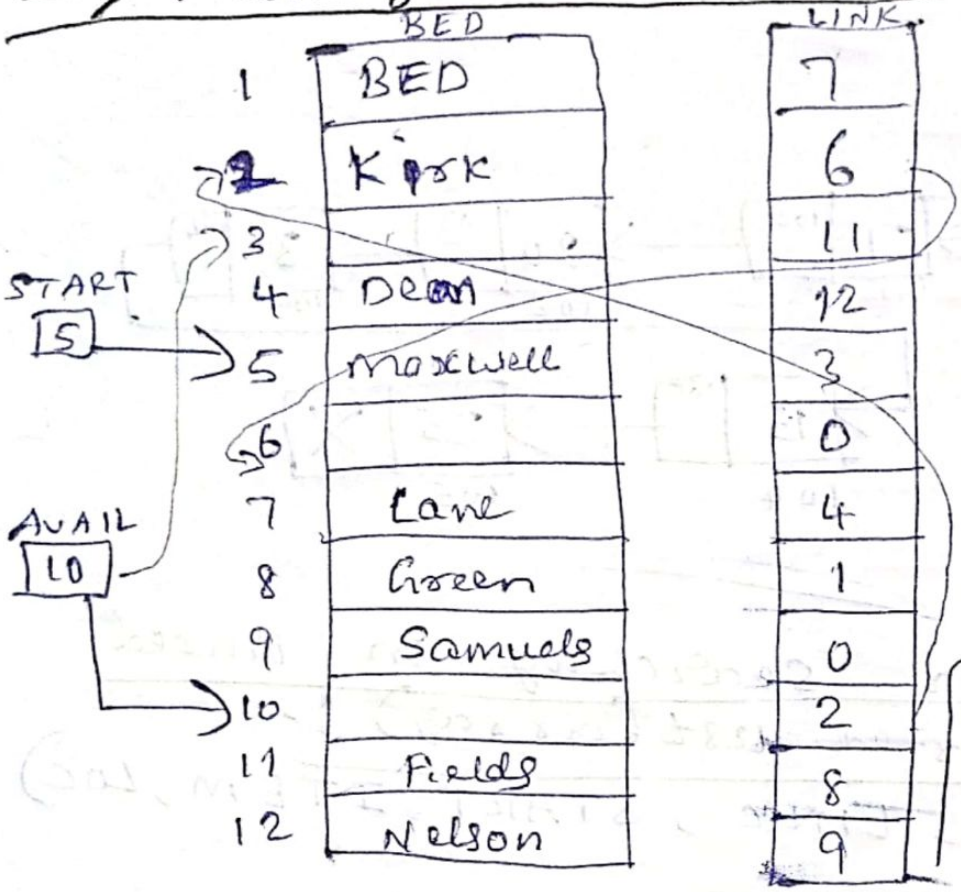
↳ Algorithm for searching in linked list for sorted list (array) :-

0) Search [INFO, LINK, START, ITEM, LOC)
1) set PTR = START
2) Repeat step-3 while PTR ≠ NULL
3) if ITEM > INFO[PTR], then
set PTR = LINK[PTR], [PTR now points to next node]
else (if fail the statement)
if ITEM = INFO[PTR], then
set LOC = PTR and exit
[search is successful]
else
set LOC = NULL, and Exit [ITEM now exceeds INFO[PTR]
[End of if structure]
[End of step-2 loop]
4) set LOC = NULL
5) Exit

start
Item = 35



Memory allocation :-



[To insert item]

NEW = 10, AVAIL = 3

BED[10], John

LINK[10] = 5

START = 10

[FINDING LOC]

ITEM = BACON

ITEM < INFO[STAR

BACOM < ADAM

Sno = 5

BED[10], BED[3], BED[6]

- Linked list is considered a data structure for which size is not fixed & memory is allocated from the heap section as & when needed.
- The elements can be accessed through index mechanism.
- whenever a new element needs to be added, a separate memory is allocated to store both key & the pointer to the next element.

↳ Garbage collector :-

- Garbage collection is a dynamic approach to automatic memory management & heap allocation that identifies dead memory blocks & reallocates storage for reuse.
- The primary purpose of garbage collection is to reduce memory leaks.
- The free storages use by other programs.
- The garbage collection management is done by operating system.
- It is invisible to the programer.

↳ Overflow & Underflow :-

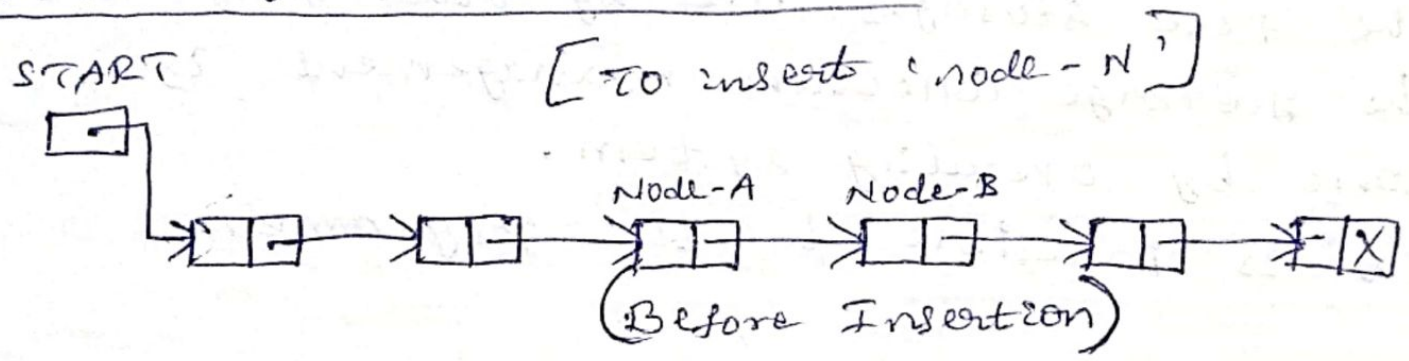
- Both are errors resulting from a shortage of space.
- Overflow is a condition that arises when we want to insert a new data into the list, but there is no available space in the linked list.
  - It means that, there are no any empty list to store a new data.
- Underflow is a condition that arises when we want to delete some data from an array (or) list, but there are no any data available in the linked list.
  - It means that given data is empty, so if empty then there is no any element is available to delete.

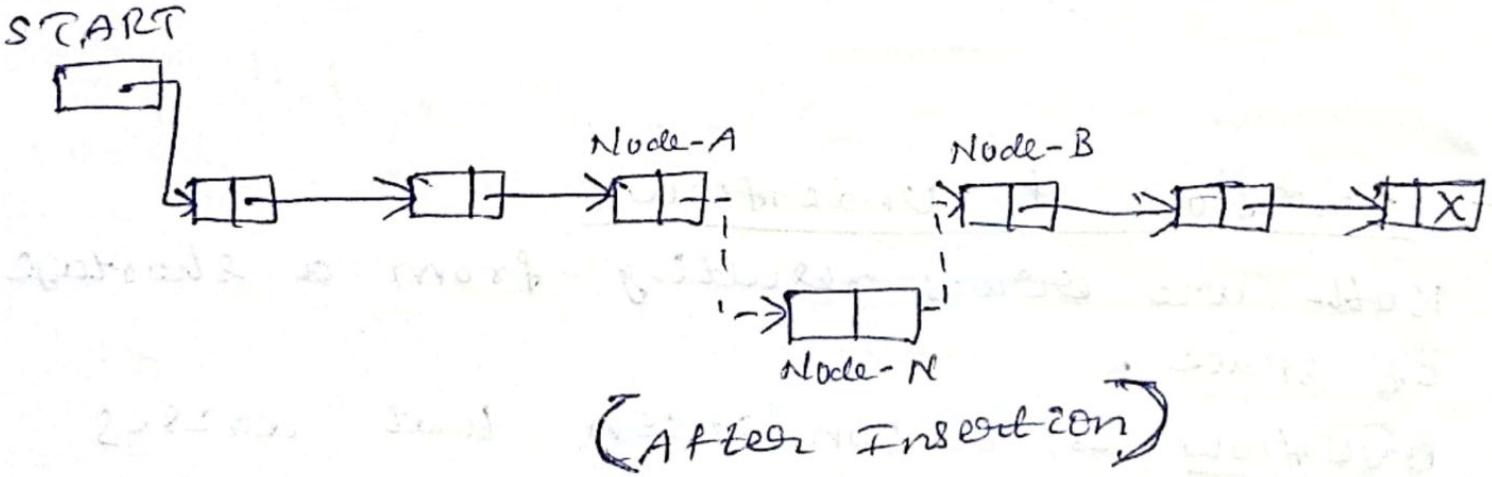
↳ Insertion into a Singly Linked list :-

Algorithm :-

1) Insert node at beginning of list.
2) Insert node at end of list.
3) Inser node at given position in list.

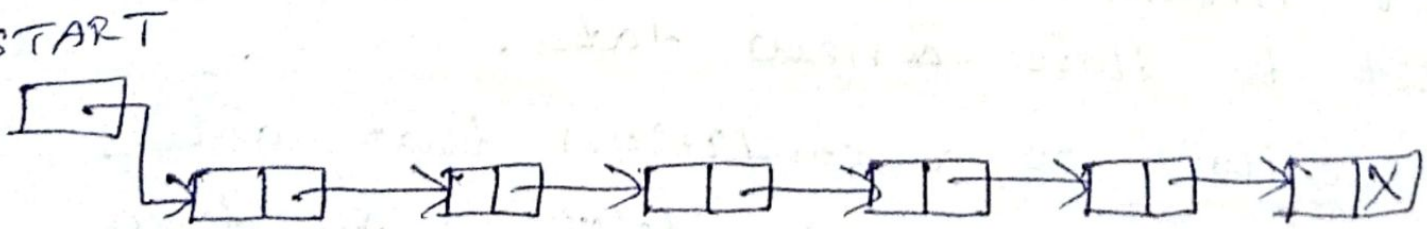
Inserting in Linked List :





Suppose, storing on linked list in memory.

LIST (INFO, LINK, START, NULL)



1) Algorithm :- [starting of list]:

- INSFIRST(INFO, LINK, START, AVAIL, ITEM)
- This algorithm inserts ITEM as the first node in the list.

1) [OVERFLOW?]
If AVAIL = NULL, then:
write: OVERFLOW, and Exit:

2) [Remove first node from AVAIL list]
Set NEW := AVAIL and AVAIL := LINK[AVAIL]

3) set INFO[NEW] := ITEM [copies new data into new node]

4) set ~~NEW[LINK]~~ LINK[NEW] := START [New node new Points to Original first node]

5) set START := NEW [changes START so it Points to a new node]

6) EXIT

gic

START = GEOM

ALG 11

GEOM 5

AVAIT 9

	(INFO) TEST	LINK
1		16
2	74	14
3		1
4	82	0
5	84	12
6	78	0
7	74	8
8	100	13
9	75	10
10	100	3
11	88	2
12	62	7
13	74	6
14	73	4
15		0
16		15

ITEM = 75
NEW = 9
LINK = [9] = 10
TEST [9] = 75
~~NEW~~ LINK [9] = 5
GEOM = 9

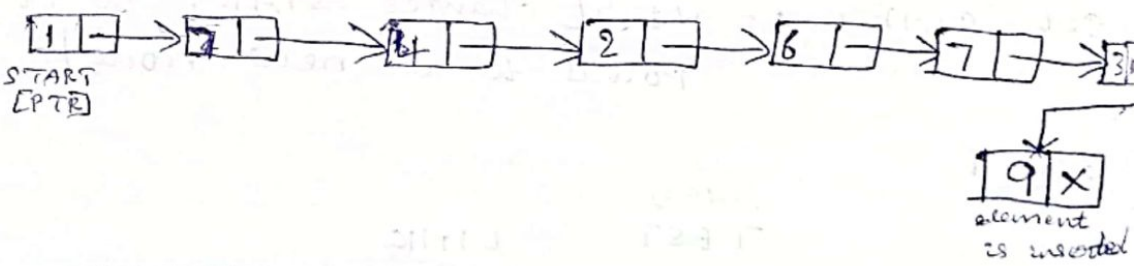
AVAIL [10]

Item = 100
NEW = 10
LINK = [10] = 3
TEST [10] = 100
LINK [10] = 9
GEOM = 10

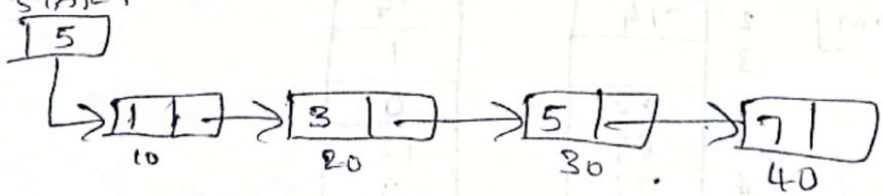
(2) Insert in
Algorithm [End of list] :-

1) If AVAIL = NULL write OVERFLOW
   Go to step - 10 [End of if]
2) set NEW - NODE = AVAIL
3) set AVAIL = AVAIL → NEXT
4) set NEW - NODE → DATA = VAL
5) set NEW - NODE → NEXT = NULL
6) set PTR = START
7) Repeat step-8 while PTR → NEXT != NUL
8) set PTR = PTR → NEXT
   [End of loop]
9) set PTR → NEXT = NEW - NODE
10) EXIT

Eg:-



Eg:- START



- ITEM < INFO [START]
  4 < 5 ✓
- SAVE = START
  = 5
- PTR = LINK [START]
  = 30
- PTR ≠ NULL → Fails
- ITEM < INFO [PTR]
  4 < 5 ✓
  set LOC = 5
- SAVE = 30, PTR = LINK [PTR]
  = 40
- LOC = 30

4 item

3) Inserting after a given node (location):

- INSLOC (INFO, LINK, START, AVAIL, LOC, ITEM)
- This algorithm inserts an item, so that item follows node with location LOC or inserts an item as a 1st node when LOC = NULL

1) [OVERFLOW?] If AVAIL = NULL item, write OVERFLOW and EXIT.
2) [Remove first node from AVAIL list]
   set NEW = AVAIL & AVAIL = LINK[AVAIL]
3) set INFO[NEW] = ITEM
   [Copies new data into a new node]
4) If LOC = NULL, then [Insert in FIRST NODE]
   set LINK[NEW] = START and START = NEW
   else [Insert after node with location
5) set LINK[NEW] = LINK[LOC] and LINK[LOC] = NEW
   [End of if structure]
6) EXIT

Alg - 5.5

(ii) Inserting into a sorted linked list: (Finding a location)

- FINDA [INFO, LINK, START, ITEM, LOC]
- This Procedure finds the location LOC of the last node in a sorted list such that:

1) [list empty!] If START = NULL, then set LOC = NULL and Return
2) [Special Case?]
   If ITEM < INFO[START], then set LOC = NULL and return.
3) set SAVE = START and PTR := LINK[START]
   [Initializes pointers],

4) Repeat Step-5 & 6 while PTR ≠ NULL

5) If ITEM < INFO[PTR], then
set LOC := SAVE and return
[End of if structure]

6) set SAVE := PTR and PTR := LINK[PTR]
[Updates pointers]
[End of step-4 loop]

7) set LOC = SAVE

8) EXIT.

(9) Inserting item into a linked list :-

- INSERT (INFO, LINK, START, AVAIL, ITEM)
- This algorithm inserts ITEMS into a sorted linked list.

1) [Use Procedure 5, 6 to find the location of node preceding ITEM]
call FINDA (INFO, LINK, START, ITEM, LOC)

2) [Use algorithm 5.5 to insert ITEM after the node with location LOC]
call INSLOC (INFO, LINK, START, AVAIL, LOC, ITEM)

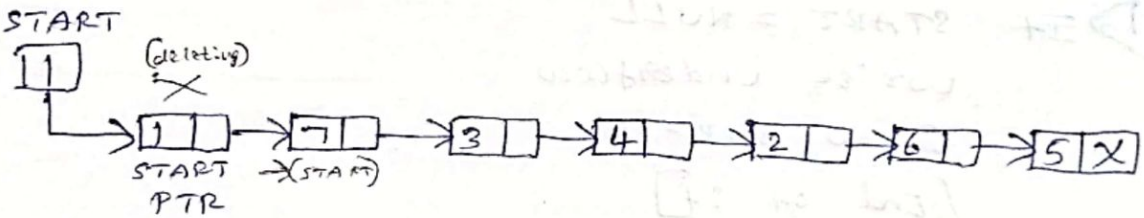
3) EXIT

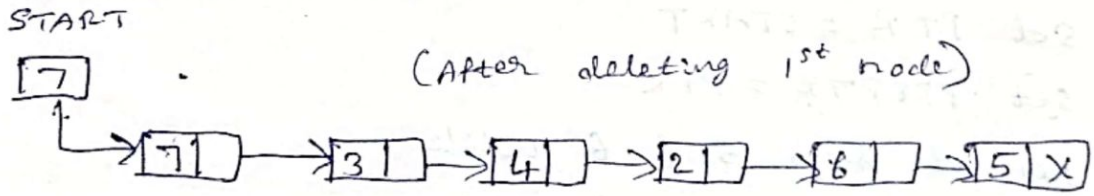
Deletion from a Singly linked list :-

case-1 :- Deleting the 1st node from linked list

Algorithm :

1) If start = NULL
   write underflow
   Go to step-5
   [End of if]
2) Set PTR = START.
3) set START = START → NEXT
4) FREE PTR (delete)
5) Exit.

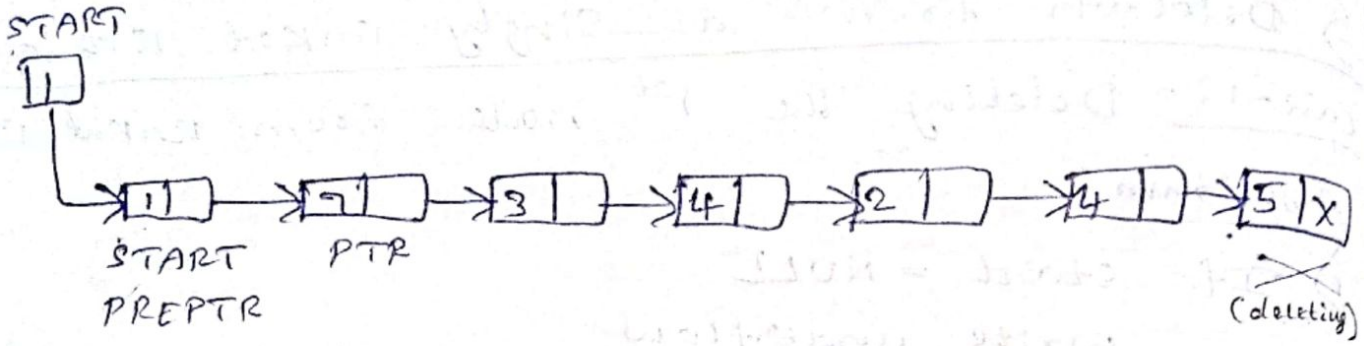


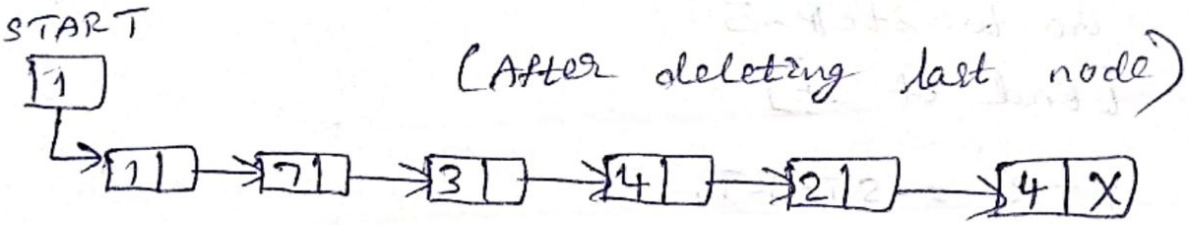


case-2 :- Deleting the last node from Linked list

Algorithm :

1) If START = NULL
   write underflow
   Go to step - 8
   [End of if]
2) set PTR = START
3) Repeat step 4 & 5 while
   PTR → NEXT != NULL
4) set PREPTR = PTR
5) set PTR = PTR → NEXT
   [End of loop]
6) set PREPTR → NEXT = NULL
7) FREE PTR
8) EXit





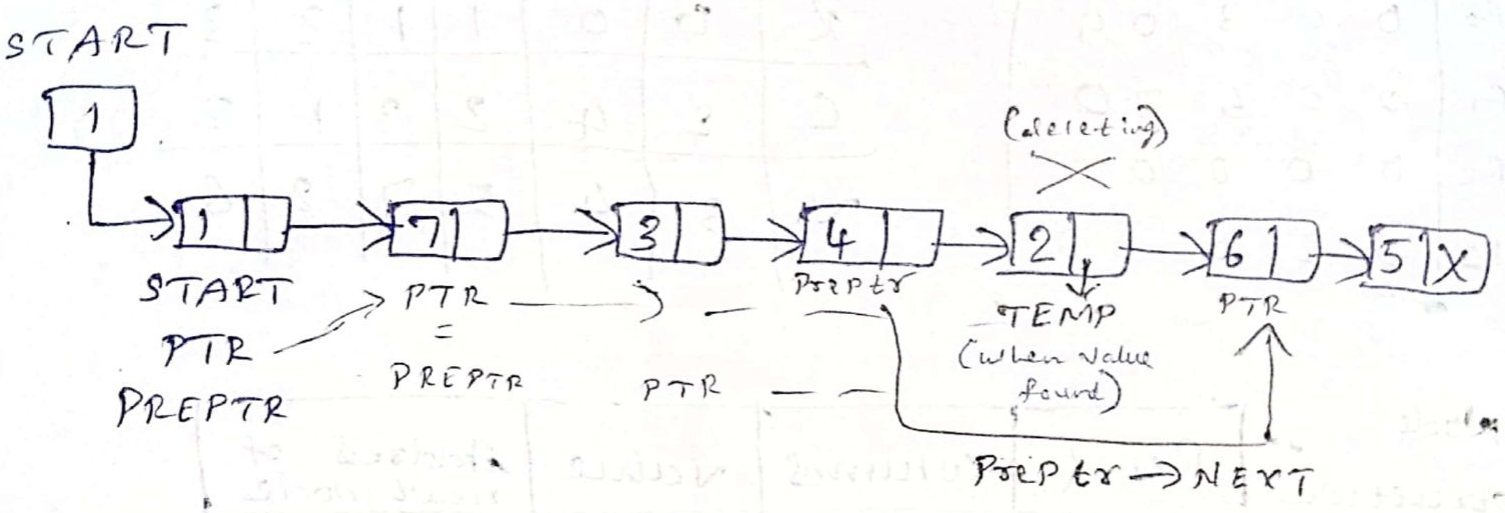
(After deleting last node)

case-3 :- Deleting after a given node in Linked list

Algorithm :

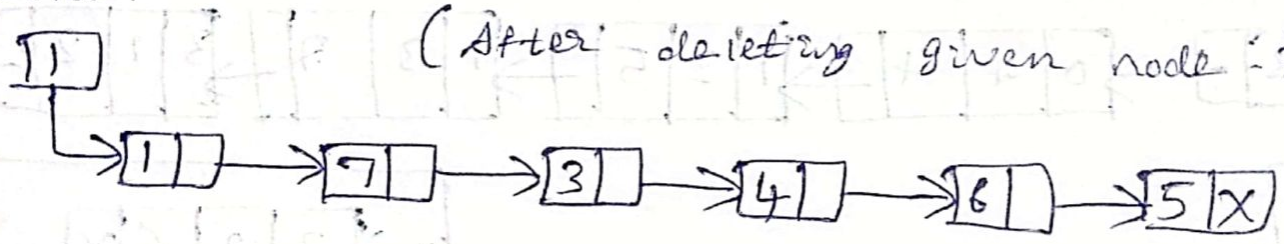
1) If START = NULL
   write underflow
   Go to step - 10
   [End of if]
2) set PTR = START
3) set PREPTR = PTR
4) Repeat steps 5 & 6 while
   PREPTR → DATA != NUM
5) set PREPTR = PTR
6) set PTR = PTR → NEW
   [End of loop]
7) set TEMP = PTR
8) set PREPTR → NEXT = PTR → NEW
9) FREE TEMP
10) Exit.

(Deleting a node that succeeds the element value - 4) = 2



START

(After deleting given node - 2)



↳ **Applications:-** [Singly linked list]

- Implementation of stacks & queues.
- Implementation of graphs & hash tables.
- Dynamic memory allocation. (management)
- Performing arithmetic operations.
- manipulation of polynomials.
- Representing sparse matrices.

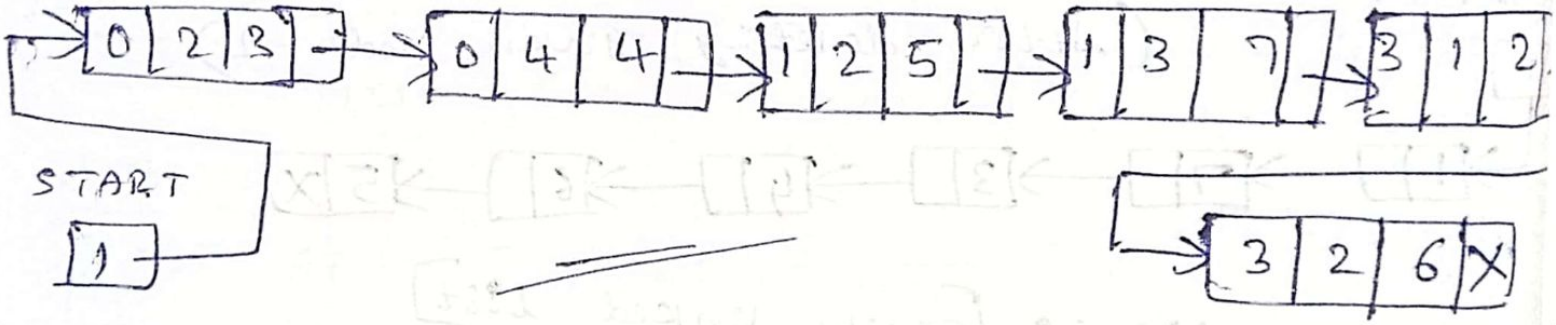
↳ Sparse matrix in linked list :-

	C_0	C_1	C_2	C_3	C_4
R_0	0	0	3	0	4
R_1	0	0	5	7	0
R_2	0	0	0	0	0
R_3	0	2	6	0	0

R	0	0	1	1	3	3
C	2	4	2	3	1	2
V	3	4	5	7	2	6

node structure

Rows	columns	value	Address of next node

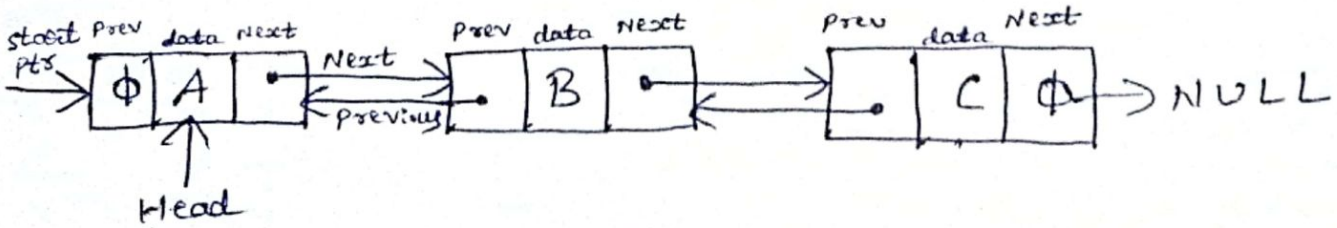


→ Doubly linked list :- It is a variation linked list in which navigation is possible in both ways, either forward (successor) & backward (predecessor) easily as compared to single linked list. (Bidirectional)

- It is a linked data structure that consists of a set of sequentially linked records called nodes.
- Each node contains 3 fields :
  - Two link (Pointer) fields references to the previous & to the next node in the sequence of nodes.
  - One data field.

↳ Inserting at beginning of nodes in doubly linked list :

1) Begin.
2) if fr = NULL then
   Print overflow and exit
3) Input New item.
4) Set New ← fr. fr ← next [fr]
5) data [NEW] ← item,
6) Next [NEW] ← first.
7) Previous [New] ← last.
8) first ← New
9) Exit



Eg:-

	Previous	Data	Next
first → 1	Null	A	2
2	1	B	3
3	2	C	4
last → 4	3	D	Null
last, fr → 5	4	X	6
new → 6	5	Y	7
fr → 7	6		8
8	7		Null

FT - Forward Tracing

BT - Backward Tracing

FT - ABCD

BT - DCBA

FT :

new = 5, fr = 6, Item = X

Data[New] = Data[5] ← X (inserting X)

Next[New] = Next[6] ← 1 A B C D

(fr)

BT : X D C B A

- new = 6, fr = 7, Item = Y

Data[New] = Data[6] ← Y

Next[New] = Next[7] ← 1

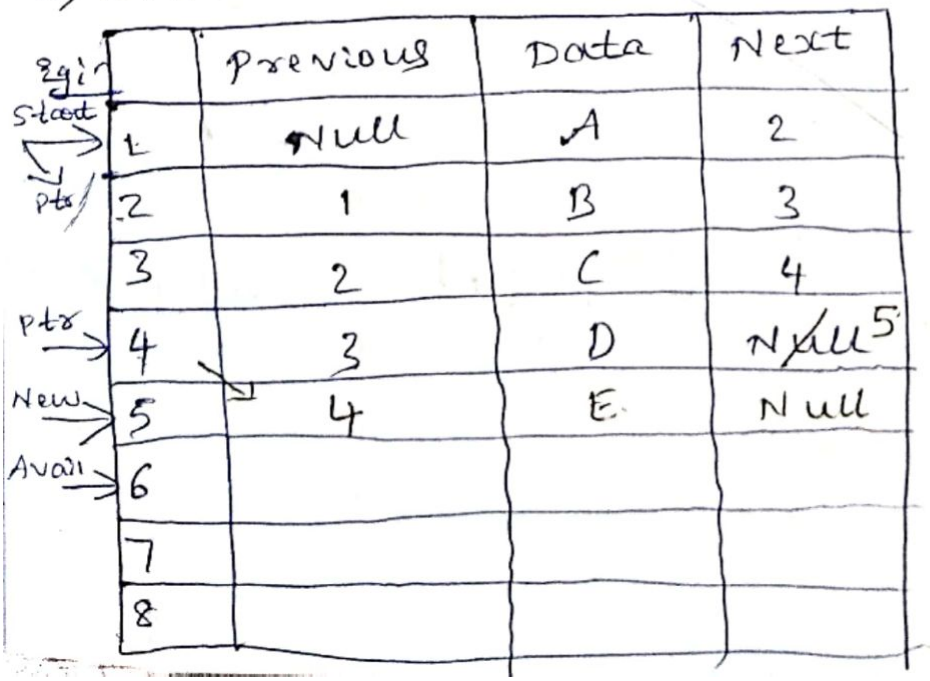
Previous[New] = Prev[6] ← last(6)

first ← new = first ← 6

Inserting a node at the end of doubly linked list :-

Algorithm :-

1) If AVAIL = NULL
   Write underflow
   Go to step - 11
   [End of if]
2) Set NEW - Node = AVAIL
3) set AVAIL = AVAIL → NEXT
4) set NEW - NODE → DATA = VAL[item]
5) set NEW - NODE → NEXT - NULL
6) set PTR = START
7) Repeat step - 8 while
   PTR → NEXT != NULL
8) set PTR = PTR → NEXT
   [End of LOOP]
9) set PTR → NEXT = New - NODE
10) set NEW - NODE → PREV = PTR
11) EXIT



	Previous	Data	Next
1	Null	A	2
2	1	B	3
3	2	C	4
4	3	D	Null 5
5	4	E	Null
6			
7			
8			

FT :
A B C D E

BT :
E D C B A

Eg:-

	Previous	Data	Next
start → 1	Null	H	3
2	1	E	6
3	3	L	7
4	6	L	9
ptr → 5	7	O	Null 6
→ New 6		X	

- Avail = Null
- New - node = Avail → 6
- Avail = Avail → Next →
- New - node → data → X
- New - node → Next - Null →

↳ Inserting a node at the given position in DLL :-

Algorithm :-

1) If AVAIL = NULL
   write overflow
   Go to step - 12
   [End of if]
2) set NEW - NODE = AVAIL
3) set AVAIL = AVAIL → NEXT
4) set NEW - NODE → DATA = VAL
5) set PTR = START
6) Repeat step - 7 while (B = 2)
   PTR → DATA != NUM.
7) set PTR = PTR → NEXT
   [End of loop]
8) set NEW - NODE → NEXT = PTR → NEXT
9) set NEW - NODE → PREV = PTR
10) set PTR → NEXT = NEW - NODE
11) Set PTR → NEXT → PREV = NEW - NODE
12) EXIT

		Previous	Data	Next
Start →	1	Null	A	2
ptr →	2	1	B	5
	3	5	C	4
	4	3	D	Null
New →	5	2	X	3
Avail →	6			7
	7			8
	8			Null

FT : AB X CD

BT : DC X BA

↳ Deleting a node at the beginning in DLL:-

Algorithm:-

1) If START = NULL
   write underflow
   Go to step - 4
   [End of if]
2) set PTR = START
3) set START = START → NEXT
4) set START → PREV = NULL
5) FREE PTR
   (delete)
6) EXIT

Eg:

	Previous	Data	Next
start PTR → 1	NULL	A	2
2	1	B	3
3	2	C	4
4	3	D	Null
5			6
6			7
7			8
8			NULL

[After deleting node at beginning]

	Previous	Data	Next
2	NULL	B	3
3	2	C	4
4	3	D	Null

Deleting a last node in DLL :-

Algorithm :-

1) If START = NULL
   write underflow
   Go to step-7
   [End of if]

2) Set PTR = START

3) Repeat step-4 while

4) PTR → NEXT != NULL
   set PTR = PTR → NEXT
   [End of LOOP]

5) set PTR → PREV → NEXT = NULL

6) FREE PTR

7) EXIT

Eg:- start PTR

	Previous	Data	Next
1	Null	A	2
2	1	B	3
3	2	C	~~4~~ Null
4	3	D	Null
5			~~6~~
6			7
7			8
8	—		Null

[After deleting last node]

	Previous	Data	Next
1	Null	A	2
2	1	B	3
3	2	C	Null

Deleting the node after given node in DLL :-

Algorithm :-

1) If start = Null
   write underflow
   Go to step-9
   [End of If]
2) set PTR = start.
3) Repeat step-4 while PTR → DATA 1 = NUM
4) Set PTR = PTR → NEXT 2
   [End of loop]
5) set TEMP 3 = PTR → NEXT.
6) set PTR → NEXT = TEMP → NEXT
7) set TEMP → NEXT → PREV = PTR
8) FREE TEMP
9) EXIT

Eg:

	Previous	Data	Next
start PTR → 1	Null	A	~~2~~ 3
2	1	B	3
3	~~2~~ 1	C	4
4	3	D	Null
5			6
6			7
7			8
8			Null

FT : A C D

BT : D C A

Eg:-

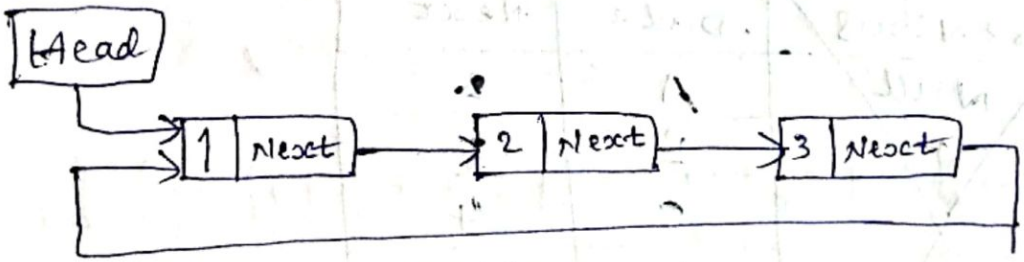
	Previous	Data	Next
1	Null	3	100
2	200	2	500/150
3	100	7	150
4	500/200	9	Null

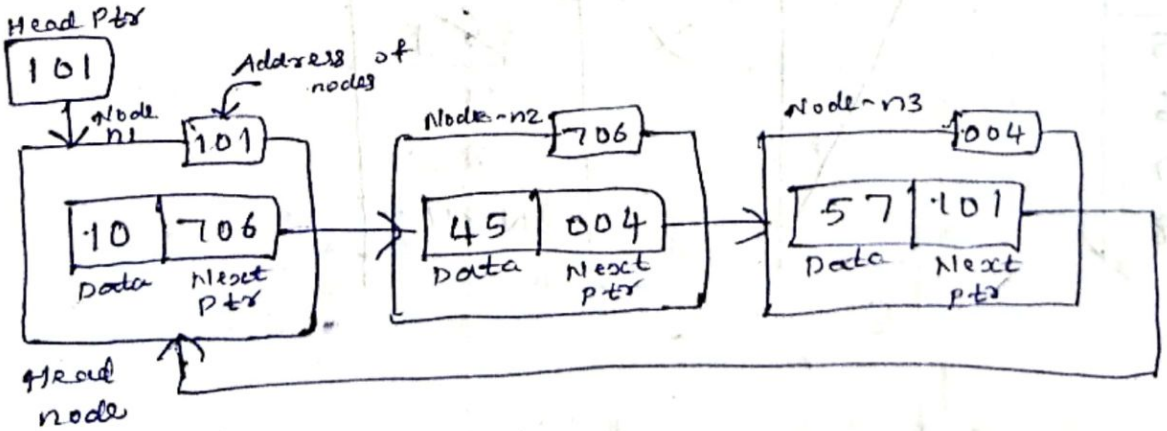
(start PTR → 1, TEMP → 3)

200
100
500
150

→ Circular Linked list :-

- Circular linked list is a type of linked list which last node points to the first node, completing a full circle of nodes.
- It doesn't have a null element at the end.
- It consists 2 fields :
  - Data field
  - Pointer field (Next)

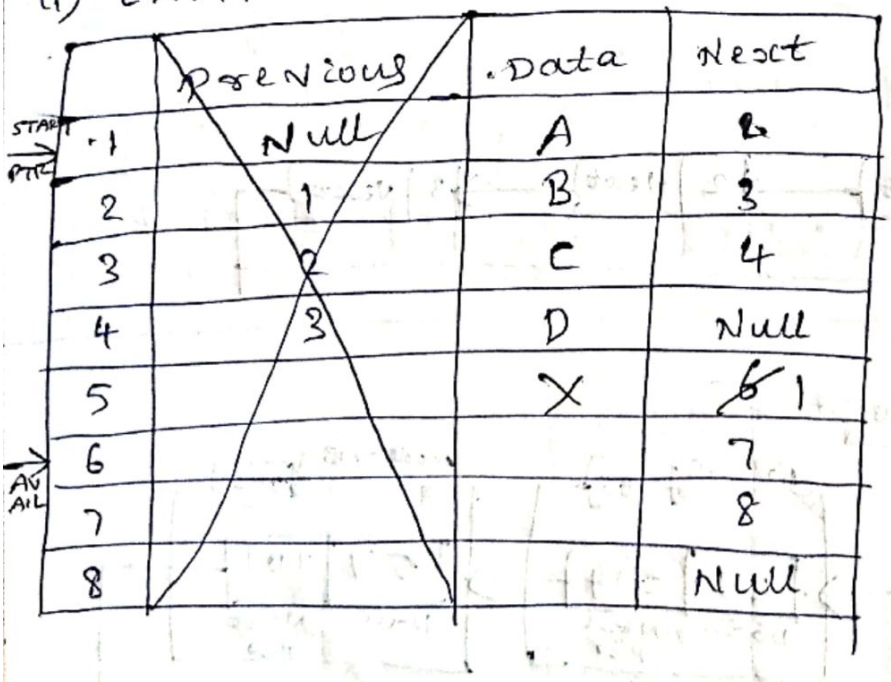




Inserting new node at beginning of Circular linked list:-

Algorithm:-

1) If AVAIL = NULL
   write OVERFLOW
   Go to step-11
   [End of If]
2) set NEW-NODE = AVAIL
3) set AVAIL = AVAIL → NEXT
4) set NEW-NODE → DATA = VAL
5) set PTR = START
6) Repeat step-7 while PTR → NEXT != START
7) PTR = PTR → NEXT
   [End of loop]
8) set NEW-NODE → NEXT = START
9) set PTR → NEXT = NEW-NODE
10) set START = NEW-NODE
11) EXIT.



	Previous	Data	Next
1	Null	A	2
2	1	B	3
3	2	C	4
4	3	D	Null
5		X	1
6			7
7			8
8			Null

	Data	Next
START PTR → 200	3	100
PTR → 100	2	500
500	7	150
150	9	600
Avail →	11	200

↳ **Inserting new node at end of circular linked list :-**

1) If AVAIL = NULL
   write OVERFLOW
   Go to step-11
   [End of If]
2) set NEW-NODE = AVAIL
3) set AVAIL = AVAIL → NEXT
4) set NEW-NODE → DATA = VAL
5) set NEW-NODE → NEXT = START
6) set PTR = START
7) Repeat step-8 while PTR → NEXT != START
8) set PTR = PTR → NEXT
   [End of loop]
9) set PTR → NEXT = NEW_NODE
10) EXIT

	Data	Next
start PTR → 200	3	100
100	2	500
500	7	150
Avail → 150 PTR	11	200
200		

FT :
3, 2, 7, 11

BT :
11, 7, 2, 3

Deleting a node for CLL :-

Case-1 :- The 1st node is deleted.

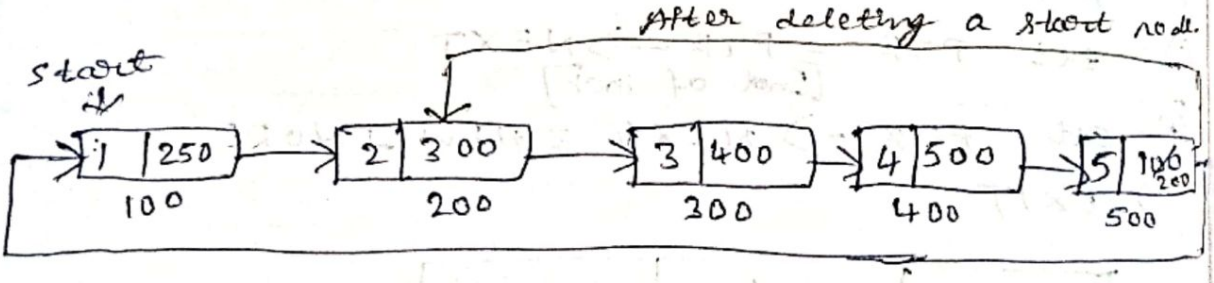
case-2 :- The last node is deleted.

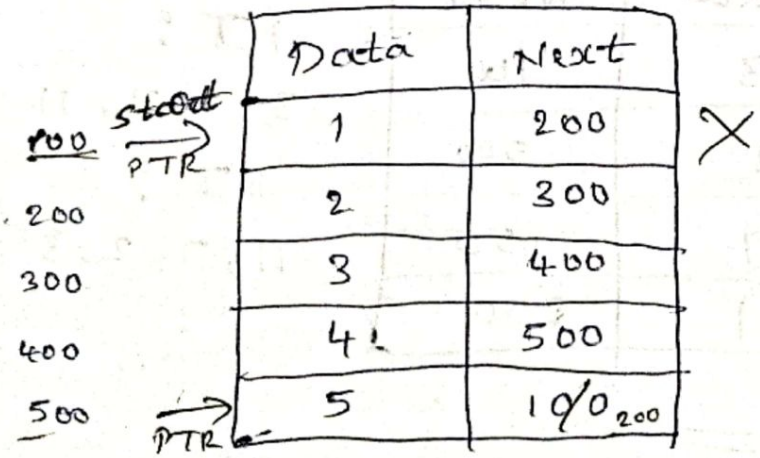
case-1 :-

Deleting the 1st node from a CLL.

Algorithm :-

1) If START = NULL
   write underflow
   Go to step-8
   [End of If]
2) set PTR := START
3) Repeat the step-4 while : PTR → NEXT != START
4) set PTR = PTR → NEXT
   [End of loop]
5) set PTR → NEXT = START → NEXT
6) FREE START.
7) set START = PTR → NEXT
8) EXIT



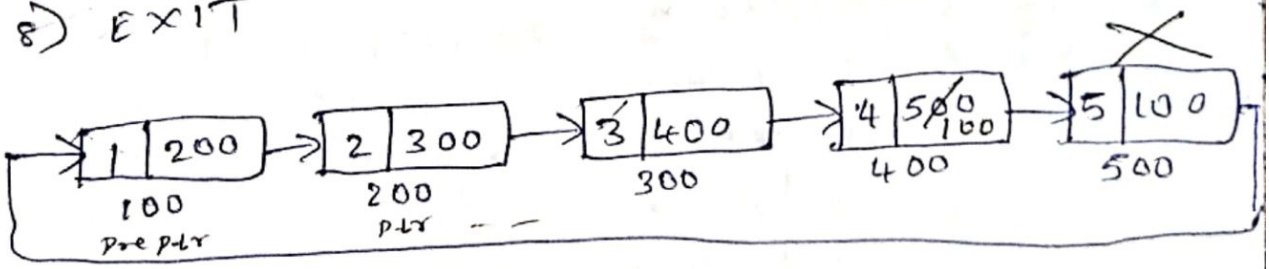


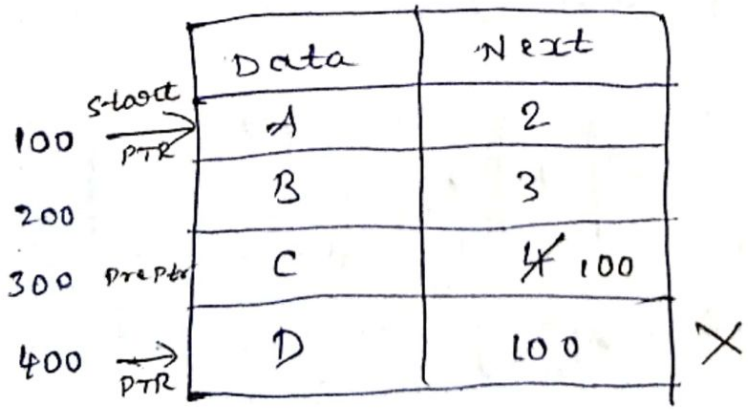
case-2:-

Deleting the last node from a CLL.

Algorithm:-

1) If START = NULL
   write underflow
   Go to step-5
   [End of If]
2) set PTR = START
3) Repeat step 4 & 5 while PTR → NEXT != START
4) set PREPTR = PTR
5) set PTR = PTR → NEXT
   [End of loop]
6) set PREPTR → NEXT = START
7) FREE PTR
8) EXIT





→ STACKS :-

- Stack is a non-primitive linear data structure.
- Stack is a order collection of items where the inserting of new data item & removing of existing data item at the same end is called top of stack.
- It is also known as LIFO (Last in first out)

Eg:- Stack of books, coins.
- No of plates in cafeteria.
- Shunting of trains in railway system.

* Applications :-

- To solve tower of hanoi.
- To implement recursion.
- Parenthesis matching.
- Used to convert infix expressions to postfix expression.
- Run-time memory management.
- Conversion of decimal no into binary.

* IsEmpty() :-

```
IsEmpty()
{
if(TOP==-1)
return true;
else
return false;
}
```

* IsFull() :-

```
IsEmpty()
{
if(MAXSTK-1)
return true;
else
return false;
}
```

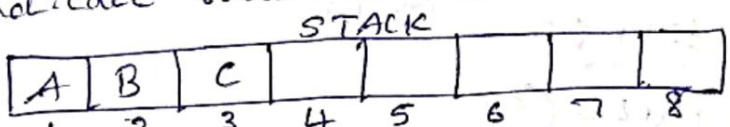
* TOP :-

```
TOP()
{
A[TOP];
}
```

↳ Array representation of stacks :-

- Stack may be represented in various way, usually by means of one-way in a linear array.
- In this, each of our stacks will be maintained by a array stack; a pointer variable TOP, which contains the location of the top element of the stack & a variable MAXSTK which gives the maximum. no of elements that can be added to the stack.
- The condition TOP = 0 (or) TOP = NULL will indicate that the stack is empty.

Eg:-



↳ Operations in Stacks :-

(1) **Stack :-** Creat a new stack that is empty

(2) **Push( ) :-** Insert a new data item to the top of the stack.

(3) **Pop( ) :-** Remove the existing data item from the top of the stack.

(4) **Peek( ) :-** It can be used to access only top of the stack.

(5) **isEmpty( ) :-** (TOP = -1) check whether the stack is empty or not (1 for true, 0 for false) (stack underflow)

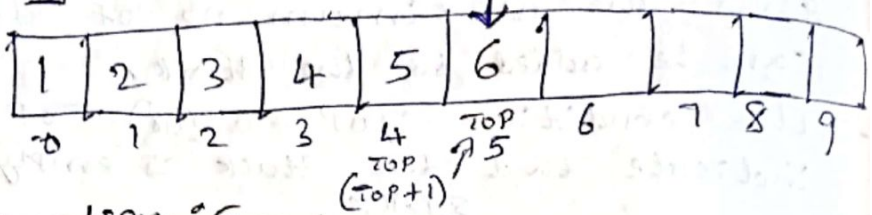
(6) **isFull( ) :-** (MAXSTK - 1) check whether the stack is full or not. It returns true if stack is full, otherwise false. (stack overflow)

(7) **Size( ) :-** The no of size returns the top of the stack.

(8) **Display :-** Display the contents of elements of the stack.

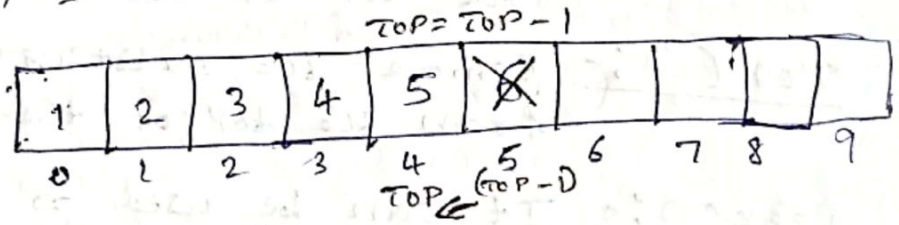
↳ Push operation :-

1) If TOP = MAX - 1 (4)
   Print Overflow
   Go to step - 4
   [End of If]
2) Set TOP = TOP + 1 = 5
3) STACK [TOP] = value (6)
4) END



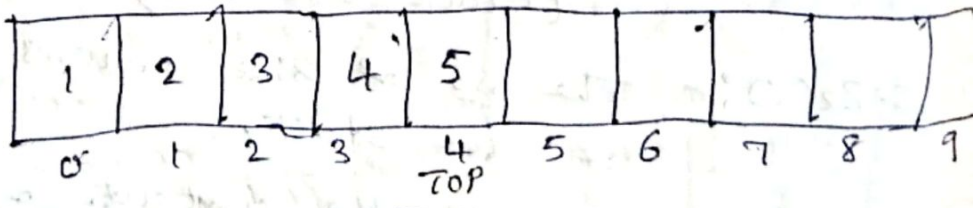
↳ POP operation :-

1) If TOP = NULL (5)
   Print underflow
   Go to step - 4
   [End of If]
2) set VAL = STACK [TOP]
3) set TOP = TOP - 1 = 4
4) END



↳ Peak operation :-

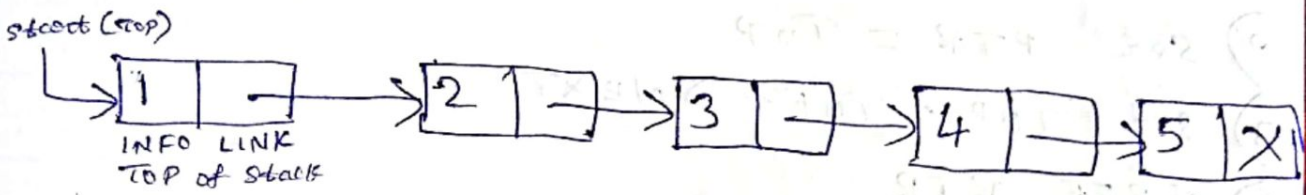
1) If TOP = NULL
   Print "STACK IS EMPTY"
   Go to step - 3
2) RETURN TOP [STACK]
3) END



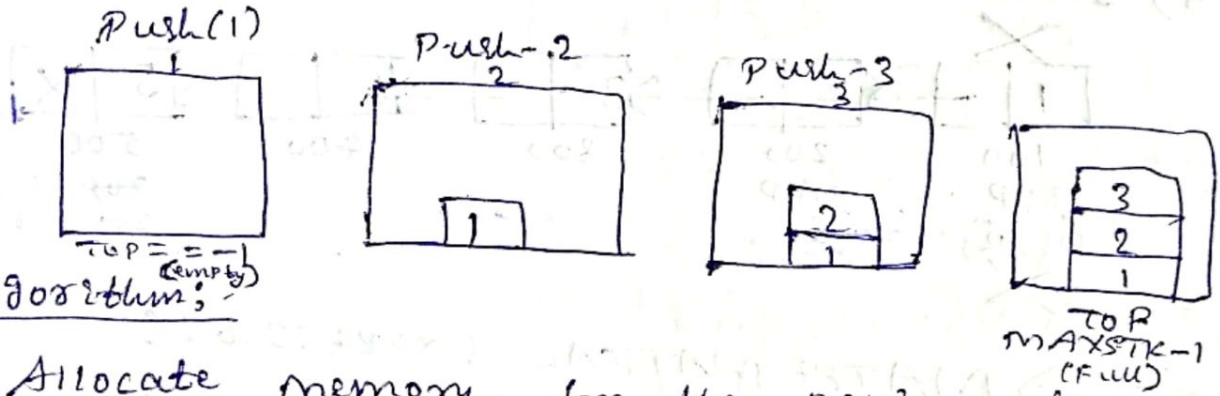
return TOP [4]
return 5

↳ Linked representation of stack :-

- The linked representation of stack commonly termed as a linked stack which is implemented using a singly linked list.
- The (INFO) fields of the nodes hold the elements of the stack.
- The (LINK) fields hold the pointers to the neighboring elements in the stack.



* Push operation :-



Algorithm :-

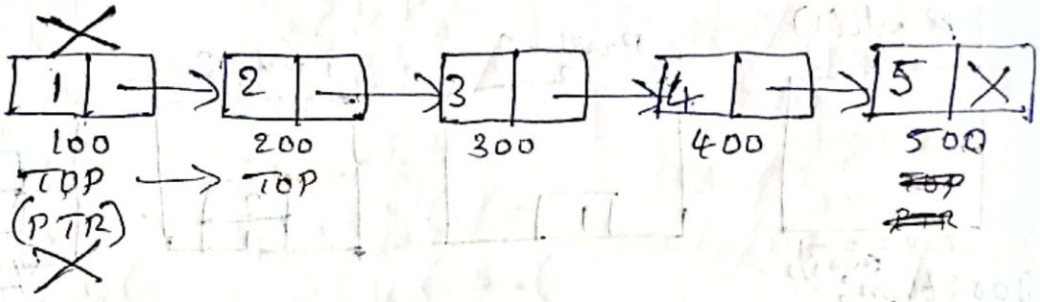
1) Allocate memory for the new node & Name it as NEW-NODE
2) Set NEW-NODE → DATA = VAL
3) If TOP = NULL
   set NEW-node → NEXT = NULL
   set TOP = NEW-NODE
   Else
   set NEW-NODE → NEXT = TOP
   set POP = NEW-NODE
   [End of If]
4) END

- The (START) pointer of the linked list behaves as the TOP pointer variable of the stack & the (null) pointer of the last node in the list signals the bottom of stack.

* POP operations:-

Algorithm:-

1) If TOP = NULL
   Print "Underflow"
   Go to step - 4
   [End of if]
2) Set PTR = TOP
3) Set TOP = TOP → NEXT
4) FREE PTR
5) END



⟹ MATHEMATICAL EXPRESSION :-

↳ **Polish notation :-** Polish notation is a notation form for expressing arithmetic, logic & algebraic equations.

- Polish notation is also known as Prefix notation, prefix Polish notation, normal Polish notation, Warsaw notation & the Lukasiewicz notation.
- The idea is simply to have a parethesis-free notation that makes each equation shorter & easier to parse in terms of defining the evaluation priority of the operators.

Operator	precedence	Value
Exponentiation ($, ↑, ^)	Highest	3
*, /	Next highest	2
+, −	Lowest	1

($, ↑, ^ *, / +, −

① $(A+B)/(C+D)-(D*E)$ — AB+CD+/DE*-

Infix expression	stack	Postfix expression
(	(	
A	(	A
+	(+	A
B	(+	AB
)	(+)	AB
/	()/	AB+
(	()/(	AB+
C	()/(	AB+C
+	()/(+	AB+C
D	()/(+	AB+CD
)	()/(+)	AB+CD
-	()/()-	AB+CD+
(	()/()-(	AB+CD+/
D	()()-(	AB+CD+/D
*	()()-(*	AB+CD+/D
E	()()-(*	AB+CD+/DE
)	()()-(	AB+CD+/DE*
	()()(	AB+CD+/DE*-

[Conversion from infix expression to postfix expression]

Infix notation :- A+B

Postfix notation :- (Reverse Polish notation) (Sufix Postfix) (AB+)

Prefix notation :- (Polish notation) (+AB)

(2) A + (B / C - (D * E ^ F) * G) * H

Infix Expression	Stack	postfix expression
A		A
+	+	A
(	+(	A
B	+(	AB
/	+(/	AB
C	+(/	ABC
-	+(-	ABC/
(	+(-(	ABC/
D	+(-(	ABC/D
*	+(-(*	ABC/D
E	+(-(*	ABC/DE
^	+(-(*^	ABC/DE
F	+(-(*^	ABC/DEF
)	+(-	ABC/DEF^*
*	+(-*	ABC/DEF^*
G	+(-*	ABC/DEF^*G
)	+	ABC/DEF^*G*-
*	+*	ABC/DEF^*G*-
H	+*	ABC/DEF^*G*-H
	+*	ABC/DEF^*G*-H*+

③ K + L − M * N + (O ^ P) * W / U / V * T + Q

Infix Expression	Stack	Postfix Expression
K		K
+	+	K
L	+	KL
− (Left to right)	−	KL+
M	−	KL+M
*	−*	KL+M
N	−*	KL+MN
+	+	KL+MN*−
(	+(	KL+MN*−
O	+(	KL+MN*−O
^	+(^	KL+MN*−O
P	+(^	KL+MN*−OP
)	+	KL+MN*−OP^
*	+*	KL+MN*−OP^
W	+*	KL+MN*−OP^W
/	+/	KL+MN*−OP^W*
U	+/	KL+MN*−OP^W*U
/	+/	KL+MN*−OP^W*U/
V	+/	KL+MN*−OP^W*U/V
*	+*	KL+MN*−OP^W*U/V/
T	+*	KL+MN*−OP^W*U/V/T
+	+	KL+MN*−OP^W*U/V/T*+
Q	+	KL+MN*−OP^W*U/V/T*+Q
		KL+MN*−OP^W*U/V/T*+Q+
^ (optional)	+^	KL+MN*−OP^W*U/V/T*+Q+
J	+^	KL+MN*−OP^W*U/V/T*+Q+J
^	+^^	KL+MN*−OP^W*U/V/T*+Q+J
A	+^^	KL+MN*−OP^W*U/V/T*+Q+JA
		KL+MN*−OP^W*U/V/T*+Q+JA^^+

(4) (A-B)*(C+D)

Infix	stack	postfix
(	(	
A	(	A
-	(-	A
B	(-	AB
)		AB-
*	*	AB-
(	*(	AB-
C	*(	AB-C
+	*(+	AB-C
D	*(+	AB-CD
)	*	AB-CD+
		AB-CD+*

(5) 9-((3*4)+8)/4

Infix	stack	postfix
9		9
-	-	9
(	-(	9-
(	-(	9-
3	-(	9-3
*	-(*	9-3
4	-(*	9-34
)	-	9-34*
+	+-	9-34*
8	+-	9-34*8
)	-	9-34*8+
/	-/	9-34*8+
4	-/	9-34*8+4
	-	9-34*8+4/

* Infix to Prefix :-

(1) A + (B * C) → Infix

A + (*BC)
x y
+xy
+A*BC → Prefix

A + (BC*)
x y
xy+
ABC*+ → Postfix

(2) (A + B) / (C - D) → Infix

(+AB) / (-CD)
x y
/xy
/+AB-CD → Prefix

(AB+) / (CD-)
x y
xy/
AB+CD-/ → Postfix

↳ Transforming Infix expression into Postfix expression :-

POLISH (Q, P)

1) Push '(' on to STACK and add ')' to the end of Q.
2) Scan Q from left to right & repeat step 3 to 6 for each element of Q until the STACK is empty.
3) If the operand is encountered, add it to 'P'
4) If a left parenthesis is encountered, Push it on to STACK.
5) If an operands (*) is encountered, then:
   (a) Repeatedly POP from stack & add to 'P' each operator (on the top of stack) which has the same procedure as or higher precedence than *.
   (b) Add (*) to STACK.

6) If a right parenthesis is encountered then:

(a) Repeatedly pop from stack and add to P each operator (on the top of stack) until a left parenthesis is encountered.

(b) Remove the left parenthesis (Do not add the left parenthesis to P) [End of step-2 loop]

7) Exit

Eg: (6) A * (B + D) / E - F * (G + H / K)

Infix	Stack	postfix
(	(	
A	(	A
*	(*	A
(	(*(	A
B	(*(	AB
+	(*(+	AB
D	(*(+	ABD
)	(*	ABD+
/	(/	ABD+*
E	(/	ABD+*E
-	(-	ABD+*E/
F	(-	ABD+*E/F
*	(-*	ABD+*E/F
(	(-*(	ABD+*E/F
G	(-*(	ABD+*E/FG
+	(-*(+	ABD+*E/FG
H	(-*(+	ABD+*E/FGH
/	(-*(+/	ABD+*E/FGH
K	(-*(+/	ABD+*E/FGHK
)		ABD+*E/FGHK/+*-

(7) (A + B / C + (D + E) - F)

Infix	Stack	Postfix
(	(	
A	(	A
+	(+	A
B	(+	AB
/	(+/	AB
C	(+/	ABC
+	(+	ABC/+
(	(+	ABC/+
D	(+	ABC/+D
+	(++	ABC/+D
E	(++	ABC/+DE
)	(+	ABC/+DE+
-	(-	ABC/+DE+
F	(-	ABC/+DE+F
)		ABC/+DE+F-

(8) (A + B / C * (D + E) - F)

Infix	Stack	Postfix
(	(	
A	(	A
+	(+	A
B	(+	AB
/	(+/	AB
C	(+/	ABC
*	(+*	ABC/
(	(+*(	ABC/
D	(+*(	ABC/D
+	(+*(+	ABC/D
E	(+*(+	ABC/DE
)	(+*	ABC/DE+
-	(-	ABC/DE+*
F	(-	ABC/DE+*F
		ABC/DE+*F-

(9) a/b+c/d → ab/cd/+

Infix	stack	postfix
(	(	
a	(	a
/	(/	a
b	(/	ab
+	(+	ab/
c	(+	ab/c
/	(+/	ab/c
d	(+/	ab/cd
)	)	ab/cd/+

(10) A-(B/C*(D/E*F)(G)*H

Infix	Stack	postfix
(	(	
A		A
-	-	A
(	(-	
B	(-	AB
/	(-/	AB
C	(-/	ABC
+	(-+	ABC/
(	(-+(	ABC/
D	(-+(	ABC/D
/	(-+(/	ABC/D
E	(-+(/	ABC/DE
*	(-+(*	ABC/DE/
F	(-+(*	ABC/DE/F
)	(-+	ABC/DE/F*
(	(-+(	ABC/DE/F*
G	(-+(	ABC/DE/F*G
)	(-+	ABC/DE/F*G
*	(*	ABC/DE/F*G+-
H	(*	ABC/DE/F*G+-H
)	(	ABC/DE/F*G+-H*

↳ Evaluation of Postfix expression :-

Algorithm :-

This algorithm finds the value of a arithmetic expression P written in postfix notation.

1) Add a right parenthesis ')' at the end of 'P' [This acts as a sentinal]
2) Scan P from left to right and repeat step 3 & 4 for each element of P until the sentinal ')' is encountered.
3) If an operand is encountered, put it on STACK.
4) If the operand (X) is encountered, then
   (a) Remove the top two elements of STACK where A is the top element & B is the next-to-top element.
   (b) Evaluate B (X) A
   (c) Place the result of (b) back on STACK [End of if structure]
   [End of step-2 loop]
5) Set VALUE equal to the top element on stack.
6) Exit-

Eg: ① 5, 6, 2, +, *, 12, 4, /, −,)

Symbol	Stack
5	5
6	6
2	2
+	5, 8
*	40
12	40, 12
4	40, 12, 4
/	40, 3
−	37

② 12 / (7 − 3) + 2 * (1 + 5)

Symbol	STACK
12	12
/	12
(	12
7	12, 4
−	12, 4
3	12, 4
)	3
+	3
2	3, 0
*	3, 2
(	3, 2, 6
1	3, 12
+	15
5	
)	

15

12 / 4 + 2 *

6

12 / 4 + 2 * 6

3 + 12 = 15

3 + 2 * (6)

5 * 6 = 30

(3) 6 2 3 4 8 + / + 2 3 ^ +

Symbol	STACK
6	6
2	6, 2
3	6, 2, 3
+	6, 5
8	6, 5, 8
4	6, 5, 8, 4
/	6, 5, 2
+	6, 7
2	6, 7, 2
3	6, 7, 2, 3
^	6, 7, 8
+	6, 15

6, 15

(4) A B + C D / A D - E A ^ + *

2 7 + 9 3 / 2 3 - 5 2 ^ + *

Symbol	STACK
2	2
7	2, 7
+	9
9	9, 9
3	9, 9, 3
/	9, 3
2	9, 3, 2
3	9, 3, 2, 3
-	9, 3, 1
5	9, 3, 1, 5
2	9, 3, 1, 5, 2
^	9, 3, 1, 25
+	9, 3, 26
*	9, 78

③ 12, 7, 3, −, 1, 2, 1, 5, +, *, +

Symbol	Stack
12	12
7	12, 7
3	12, 7, 3
−	12, 4
1	−, 3
2	*, 3, 2
1	*, 3, 2, 1
5	*, 3, 2, 1, 5
+	*, 3, 2, 6
*	*, 3, 12
+	15

[APPLICATION OF STACK]

→ Quick sort (Divide & conquer):

```
Partition (l, h)  → lower bound = l
                    higher bound = h
{
  pivot = A[l];
    i = l; j = h;
  while (i < j)
  {
    do
    {
      i++;
    }
    while (A[i] ≤ pivot);
    do
    {
      j--;
    }
    while (A[j] > pivot)
    if (i < j)
    swap (A[i], A[j]);
  }
  swap(A[l], A[j]);
  } return j;
```

```
Quick (l, h)
{
  if (l < h)
  {
  j = partition (l, h);
  Quicksort (l, j)
  Quicksort (j+1, h);
  }
}
```

Eg:-

44	33	11	55	77	90	40	60	99	22	88	66
0	1	2	3	4	5	6	7	8	9	10	11
l											h

Pivot = A[l] = A[0] = 44

- i = l = 0 , j = h = 11

(i<j) = (0<11) ✓ , i++

i = 1, A[i] ≤ pivot

A[i] ≤ pivot = (33 ≤ 44) ✓ ; i++

i = 2, A[2] ≤ pivot = (11 ≤ 44) ✓ , i++

i = 3, A[3] ≤ pivot = (55 ≤ 44) ✗ , j--

j = 10, A[j] > pivot

A[10] > 44 = (88 > 44) ✓ , j--

j = 9, A[9] > 44 = (22 > 44) ✗

(i<j) = (3<9) ✓

swap(A[i], A[j]) = (A[3], A[9])

= 55 , 22

= 22 , 55

44	33	11	22	77	90	40	60	99	55	88	66
0	1	2	3	4	5	6	7	8	9	10	11

- i = 3, j = 9 (i<j) = (3<9) ✓ , i++

i = 4, A[i] ≤ pivot

A[4] ≤ pivot = (77 ≤ 44) ✗ , j--

j = 8, A[j] > pivot

A[8] > 44 = (99 > 44) ✓ , j--

j = 7, A[7] > 44 = (60 > 44) ✓ , j--

j = 6, A[6] > 44 = (40 > 44) ✗ , j--

~~j = 5, A[5] > 44 = (90 > 44) ✓ , j--~~

~~j = 4, A[4] > 44 = (77 > 44) ✓ , j--~~

~~j = 3, A[3] > 44 = (22 > 44) ✗~~

(i<j) = (4 < 6) ✓

swap(A[i], A[j]) = (A[4], A[6])

= ~~77, 44~~ 40, 77

= ~~22, 44~~ 40, 77

44	33	11	22	40	90	77	60	99	55	88	66
0	1	2	3	4	5	6	7	8	9	10	11
l				i		j					h

Pivot = A[l] = A[0] = 44

i=4, j=6 (i<j) = (4<6) ✓, i++

i=5, A[i] ≤ Pivot

A[5] ≤ Pivot = (90 ≤ 44) ×, j--

j=5, A[j] > Pivot

A[5] > Pivot = (90 > 44) ✓, j--

j=4, A[4] > Pivot = (40 > 44) ×

(i<j) = (5<4) ×

swap(A[l], A[j]) = A[0], A[4]

= 44, 40

= 40, 44

40	33	11	22	44	90	77	60	99	55	88	66
0	1	2	3	4	5	6	7	8	9	10	11
l				j	i						h

lesser than Pivot (LHS) | Pivot | greater than Pivot (RHS)

40	33	11	22
i 0	1	2	3 j
l			h

Pivot = A[l] = A[0] = 40

i=l=0, j=h=3

(i<j) = (0<3) ✓, i++

i=1, A[i] ≤ Pivot

A[1] ≤ 40 = (33 ≤ 40) ✓, i++

i=2, A[2] ≤ 40 = (11 ≤ 40) ✓, i++

i=3, A[3] ≤ 40 = (22 ≤ 40) ✓,

j=3, A[j] > Pivot

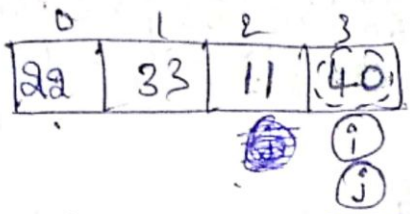
A[3] > Pivot = (22 > 40) ×

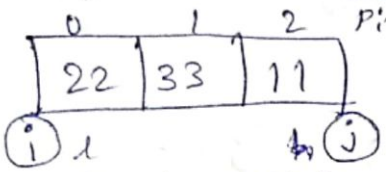
(i<j) = (3<3) ×

swap(A[l], A[j]) = A[0], A[3]

= 40, 22

= 22, 40





Pivot = A[l] = A[0] = 22

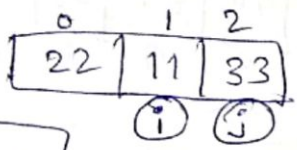
i = l = 0
j = h = 2 (i < j) = (0 < 2) ✓, i++

i = 1, A[1] ≤ Pivot = (33 ≤ 22) ✗

j = 2, A[2] > Pivot = (11 > 22) ✗

(i < j) = (1 < 2) ✓

swap(A[i], A[j]) = A[1], A[2]
= 33, 11
= 11, 33



i = 1
j = 2 (i < j) = (1 < 2) ✓, i++

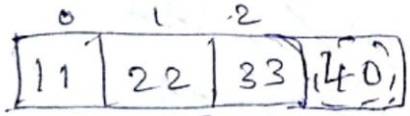
i = 2, A[2] ≤ Pivot = (33 ≤ 22) ✗

j = 2, A[2] > Pivot = (33 > 22) ✓, j--

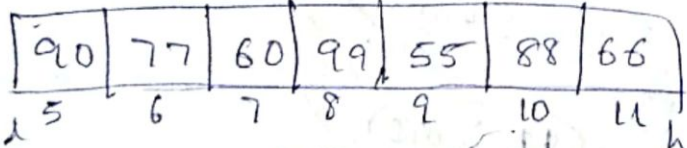
j = 1, A[1] > Pivot = (11 > 22) ✗

(i < j) = (2 < 1) ✗

swap(A[l], A[j]) = A[0], A[1]
= 22, 11
= 11, 22



LHS



Pivot = A[l] = A[5] = 90

i = l = 5
j = h = 11 (i < j) = (5 < 11) ✓, i++

i = 6, A[6] ≤ 90 = (77 ≤ 90), i++

i = 7, A[7] ≤ 90 = (60 ≤ 90), i++

i=8, A[8] ≤ 90 = (99 ≤ 90) ✗,
j=11, A[11] > Pivot = (66 > 90) ✗
(i<j) = (8<11) ✓
Swap(A[i], A[j]) = A[8], A[11]
= 99, 66
= 66, 99

5	6	7	8	9	10	11
90	77	60	66	55	88	99
			i			j

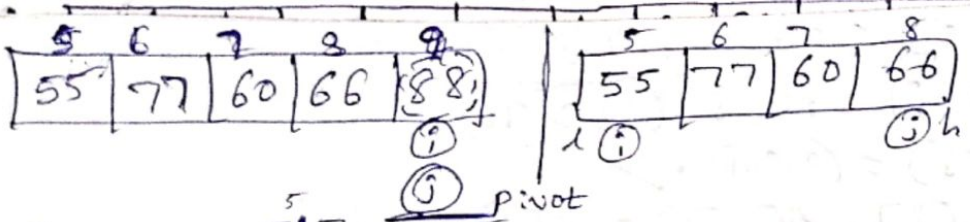
i=8, j=11 (i<j) = (8<11) ✓, i++
i=9, A[9] ≤ Pivot = (55 ≤ 90) ✓, i++
i=10, A[10] ≤ Pivot = (88 ≤ 90) ✓, i++
i=11, A[11] ≤ Pivot = (99 ≤ 90) ✗
j=11, A[11] > Pivot = (99 > 90) ✓, j--
j=10, A[10] > Pivot = (88 > 90) ✗
(i<j) = (11<10) ✗
Swap(A[l], A[j]) = A[5], A[10]
= 90, 88
= 88, 90

5	6	7	8	9	10	11
88	77	60	66	55	90	99
					j	i

Pivot

5	6	7	8	9
88	77	60	66	55
l, i				j, h

Pivot = A[l] = A[5] = 88
i=l=5, j=h=9 (i<j) = (5<9) ✓, i++
i=6, A[6] ≤ Pivot = (77 ≤ 88) ✓, i++
i=7, A[7] ≤ Pivot = (60 ≤ 88) ✓, i++
i=8, A[8] ≤ Pivot = (66 ≤ 88) ✓, i++
i=9, A[9] ≤ Pivot = (55 ≤ 88) ✓,
j=9, A[9] > Pivot = (55 > 88) ✗
(i<j) = (9<9) ✗
Swap(A[l], A[j]) = A[5], A[9]
= 88, 55
= 55, 88



Pivot = A[l] = 55

i=5, j=8 (i<j) = (5<8) ✓, i++

i=6, A[6] ≤ Pivot = (77 ≤ 55) ✗

j=8, A[8] > Pivot = (66 > 55) ✓, j--

j=7, A[7] > Pivot = (60 > 55) ✓, j--

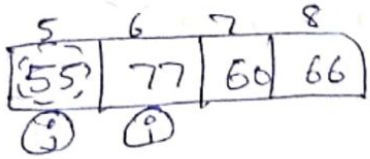
j=6, A[6] > Pivot = (77 > 55) ✓, j--

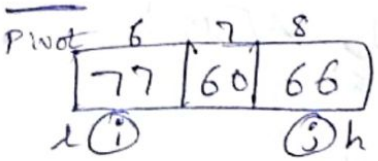
j=5, A[5] > Pivot = (55 > 55) ✗

(i<j) = (6<5) ✗

swap(A[l], A[j]) = (A[5], A[5])

= 55, 55





Pivot = A[l] = A[6] = 77

i=6, j=8 (i<j) = (6<8) ✓, i++

i=7, A[7] ≤ Pivot = (60 ≤ 77) ✓, i++

i=8, A[8] ≤ Pivot = (66 ≤ 77) ✓, i

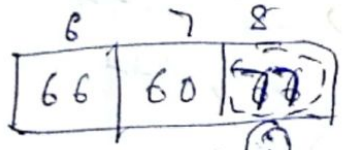
j=8, A[8] > Pivot = (66 > 77) ✗

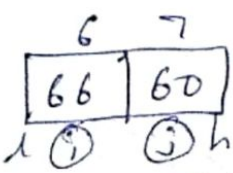
(i<j) = (8<8) ✗

swap(A[l], A[j]) = A[6], A[8]

= 77, 66

= 66, 77





Pivot = A[l] = A[6] = 66

i=6, j=7 (i<j) = (6, 7) ✓, i++

i=7, A[7] ≤ Pivot = (60 ≤ 66) ✓

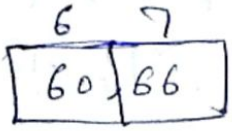
j=7, A[7] > Pivot = (60 > 66) ✗

(i<j) = (7<7) ✗

swap(A[l], A[j]) = A[6], A[7]

= 66, 60

= 60, 66



→ Application of stack :-

* Recursion :-

Algorithm :-

Factorial :

1) FACTORIAL (FACT, N)

This Procedure calculate N! & returns the value in the variable FACT

1. If N = 0, then set FACT = 1 and return.
2. Set FACT = 1 [Initialises FACT for loop]
3. Repeat for K = 1 to N

   set FACT = K * FACT ;

   [End of loop]
4. Return.

2) FACTORIAL (FACT, N)

This Procedure calculates N! & returns the value in the variable FACT,

1. If N = 0, then : set FACT := 1 & return
2. call FACTORIAL (FACT, N-1)
3. set FACT := N * FACT
4. Return.

Eg:- N = 7, K = 1, FACT = 1

Fact = 1*1 = 1

K = 2, FACT = 1 — Fact = 2 X 1 = 2

K = 3, Fact = 2 — Fact = 3 X 2 = 6

K = 4, Fact = 6 — Fact = 4 X 6 = 24

K = 5, Fact = 24 — Fact = 5 X 24 = 120

K = 6, Fact = 120 — Fact = 6 X 120 = 720

K = 7, Fact = 720 — Fact = 7 X 720 = 5040

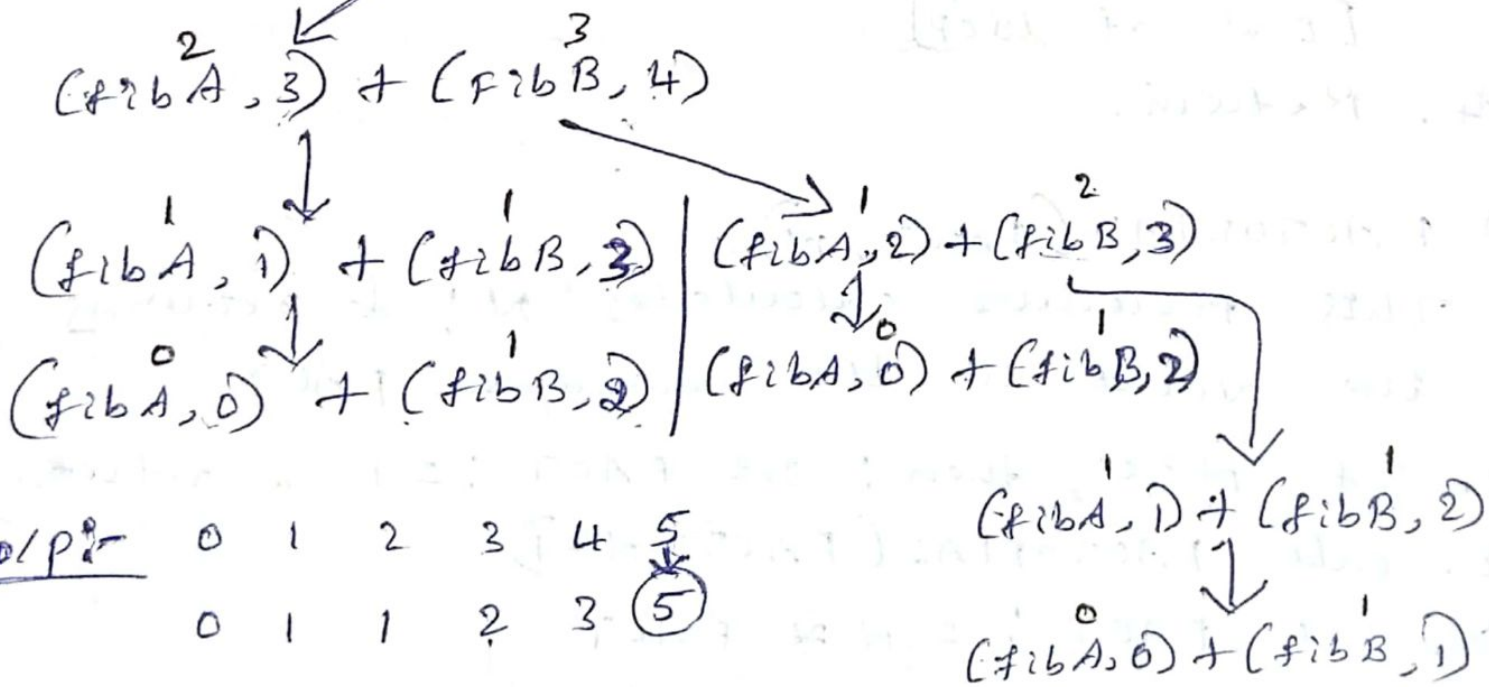
Fibonacci

FIBONACCI(FIB, N)

This Procedure calculates F_N & returns the value in the 1st parameter FIB.

1) If N = 0 (or) N = 1, then set FIB = N & return
2) call FIBONACCI(FIBA, N-2)
3) call FIBONACCI(FIBB, N-1)
4) :Set FIB := FIBA + FIBB
5) return.

Eg:- fibonacci(5)



o/p:- 0 1 2 3 4 5

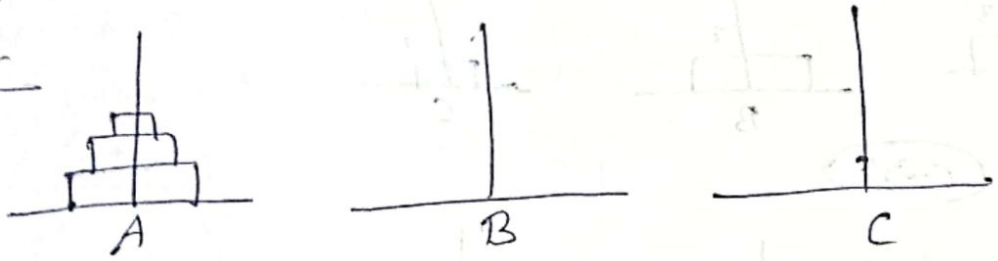
0 1 1 2 3 (5)

Tower of hanoi :-

This Procedure gives a recursive solution to the tower of hanoi problem of N disks:

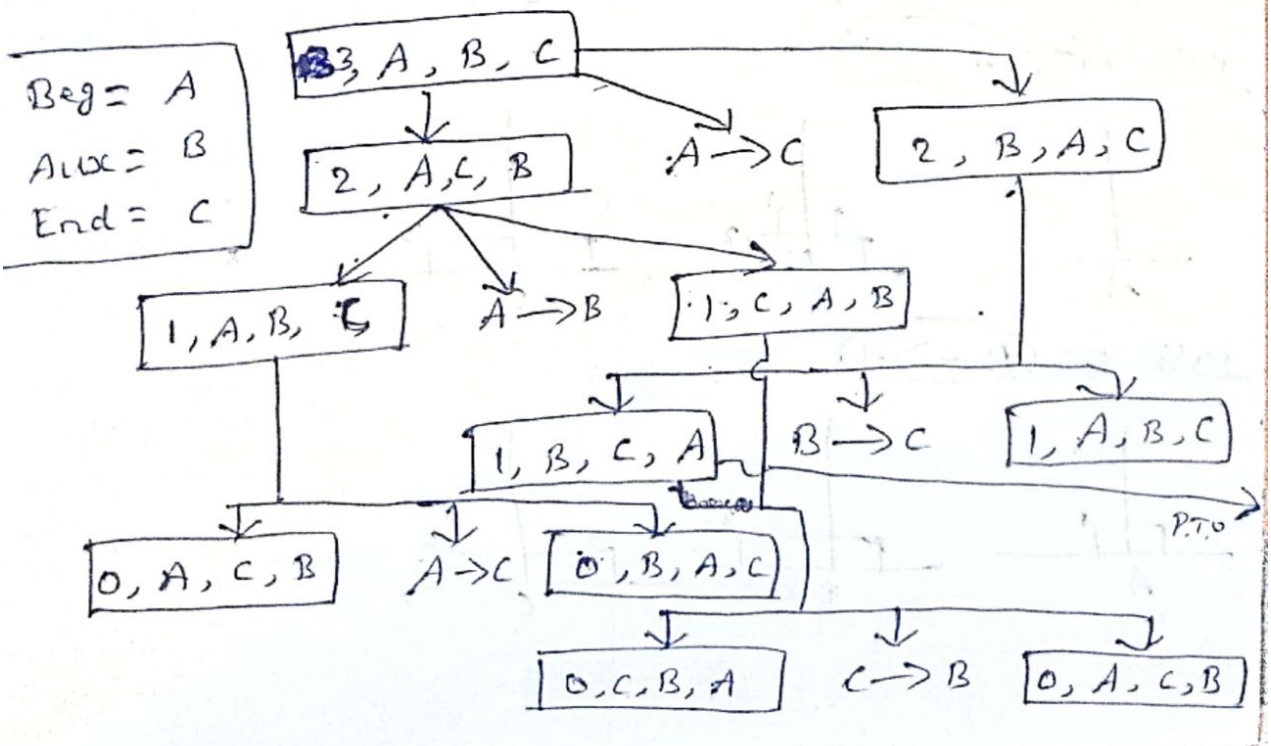
1) If N = 1, then
   (a) write : BEG → END
   (b) Return
   (End of : if structure)
2) [move N - 1 disk from peg BEG to peg AUX]
   call TOWER(N - 1, BEG, END, AUX)
3) write BEG → END
4) [move the disks from peg AUX to peg END]
   call tower(N - 1, AUX, BEG, END)
5) Return

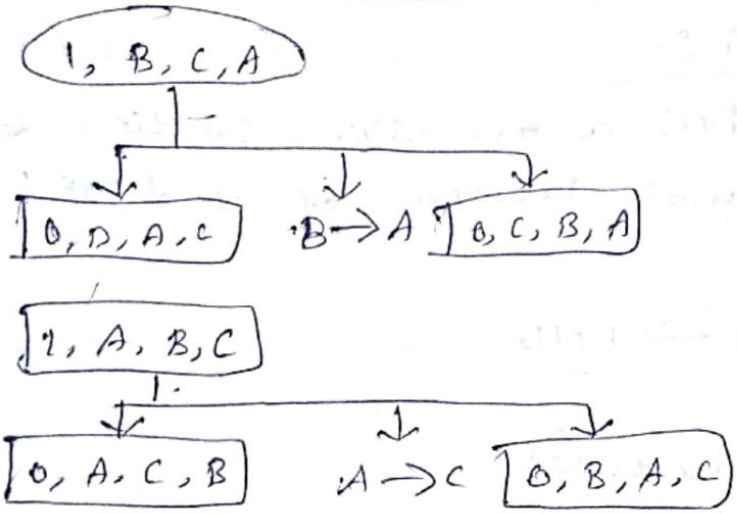
Eg:-



Probability of moves :- $2^n - 1$

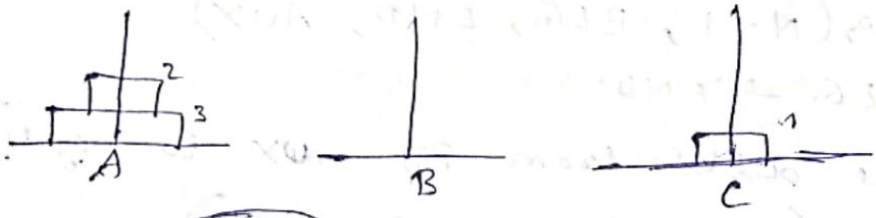
$2^3 - 1 = 8 - 1 = 7$



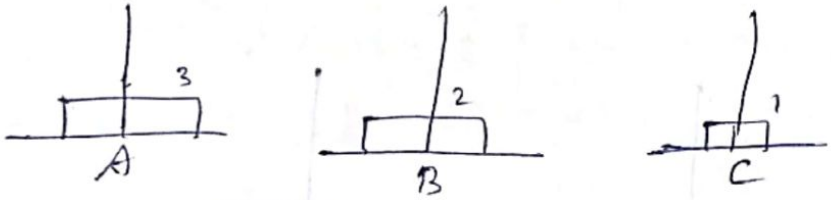


A → C
A → B
C → B
A → C
B → A
B → C
A → C

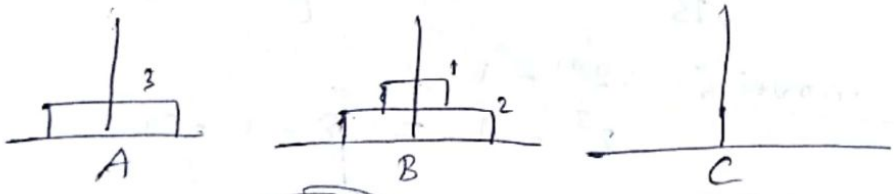
Pass-1:- (A → C)



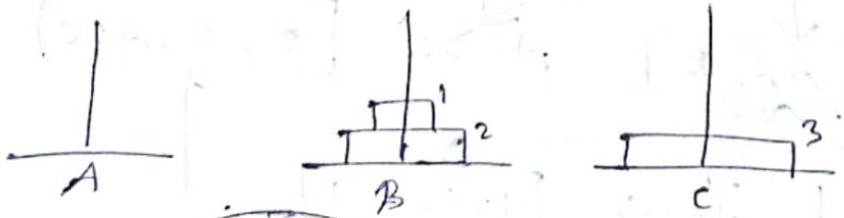
Pass-2:- (A → B)



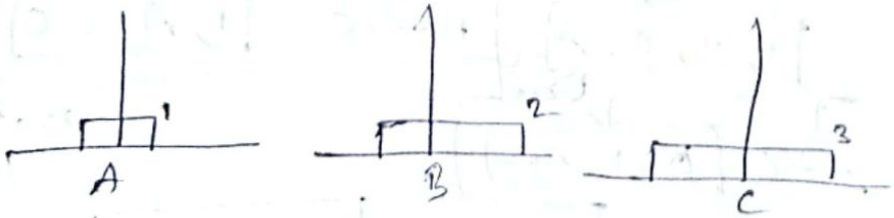
Pass-3:- (C → B)



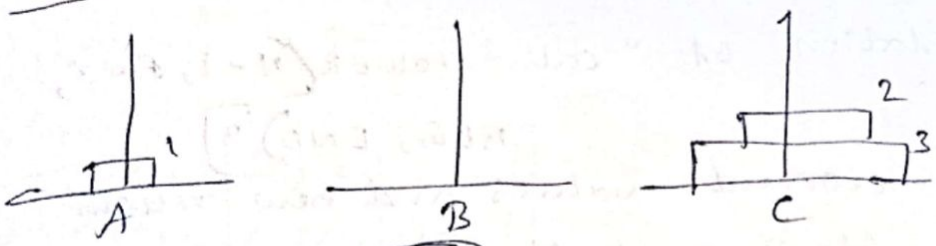
Pass-4:- (A → C)



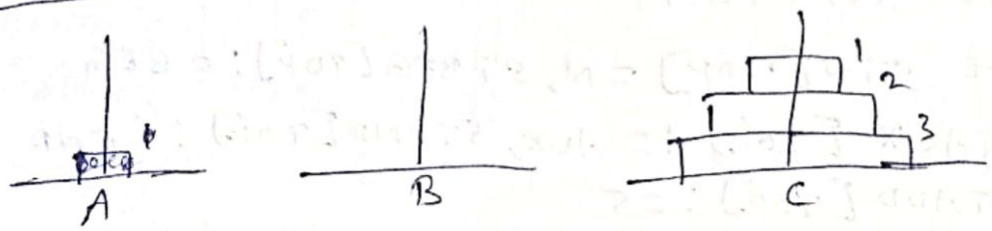
pass-5:- (B → A)



pass - 6 :- (B → C)



pass - 7 :- (A → C)



↳ Implementation of recursive Procedures by stacks :-

TOWER (N, BEG, AUX, END)

1) Set TOP = NULL
2) If N = 1, then
   (a) write BEG → END
   (b) Go to step-5
   [End of if structure]
3) [Translation of "call TOWER (N-1, BEG, END, AUX)"]
   (a) [Push current values & new values address on to stack]
   (1) Set TOP = TOP + 1
   (2) Set STN[TOP] = N, STBEG[TOP] = BEG
   STAUX[TOP] = AUX, STEND[TOP] = END
   STADD[TOP] = 3
   (b) (Reset parameters)
   Set N = N - 1 BEG = BEG, AUX = END,
   END = AUX
   (c) Go to step-1.

3) Write : BEG → END

4) [Translation of "call TOWER(N-1, AUX, BEG, END)"]

(a) [Push current values and new return address on to stacks]

(i) Set TOP = TOP+1

(ii) Set STN[TOP] = N, STBEG[TOP] := BEG
STAUX[TOP] := AUX, STEND[TOP] := END
STADD[TOP] := 5

(b) [Reset Parameters]
Set N = N-1, BEG = AUX, AUX = BEG,
END = END

(c) Go to step-1.

5) [Translation of "Return"]

(a) If TOP = NULL, then Return

(b) [Restore top values on stacks]

(i) Set N = STN[TOP], BEG = STBEG[TOP]
AUX = STAUX[TOP], STEND[TOP]
ADD = STADD[TOP]

(ii) Set TOP = TOP-1

(c) Go to step ADD.

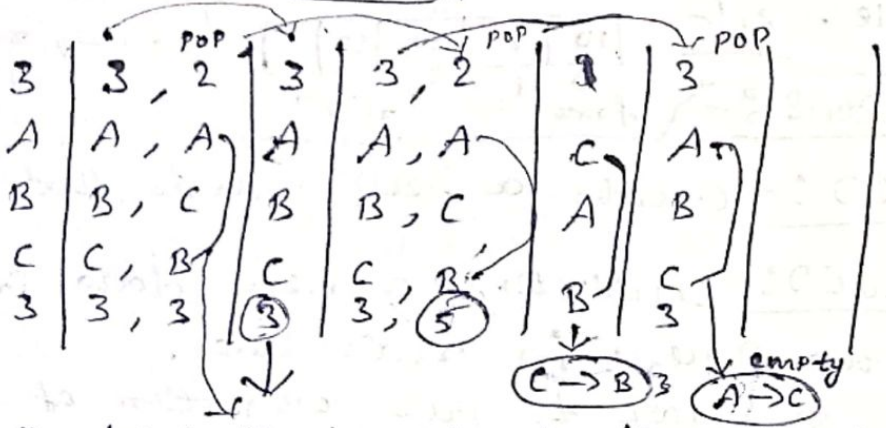
Eg:- STN, STBEG, STAUX, STEND, STADD.
3 A B C 3

N = 3 | 3, A, B, C |

A → C
A → B
C → B
A → C
B → A
B → C
A → C

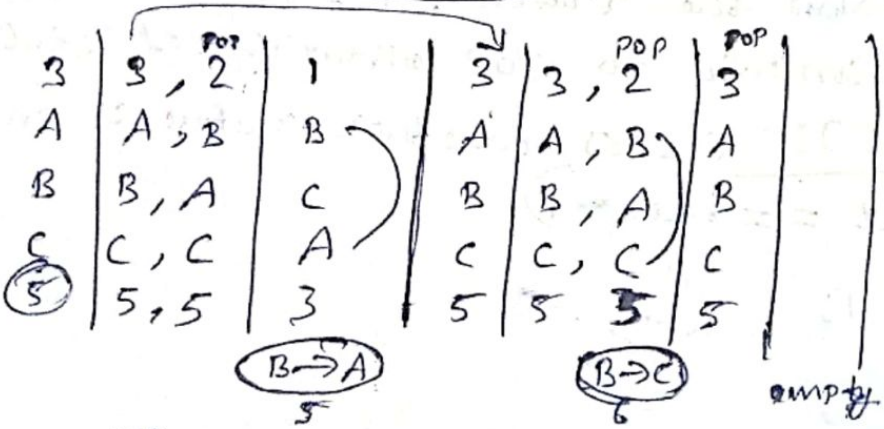
STN - 3	3	2	1
STBEG - A	A	A	A
STAUX - B	B	C	B
STEND - C	C	B	C
STADD - 3	3	3	3

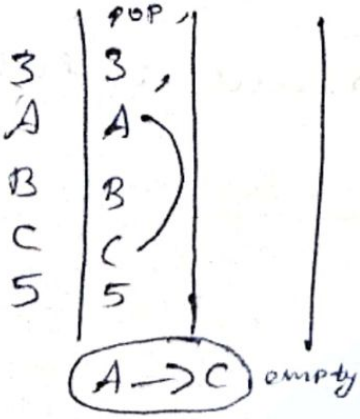
A → C



Control is transfered to 4 & write

ADD = 3 A → B 2

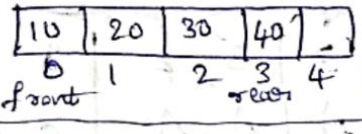




(9) (Simple queue)

→ QUEUES :-

- A queue is defined as a non-primitive linear data structure which is an ordered collection of items from which the elements may deleted at one end called "front" & elements can be inserted from other end called "rear" end.
- It follows FIFO (First in first out) Principle. Eg:-



array representation of queue

↳ operations :-

(i) Queue() :- create a new queue that is empty.

(ii) Enqueue() :- Inserting a new data item into the queue in rear end.
- It is similar to Push operation of stack.

(iii) Dequeue() :- Removing the existing data item from the queue in front end.
- It is similar to POP operation of stack.

(iv) isempty() :- checks whether queue is empty.

```
if (front == rear - 1)
{
 return 1;
else
 return 0;
}
```

(v) isfull() :- checks whether queue is full.

```
if(rear = max-1)
{
 return 1;
else
 return 0;
}
```

(vi) Size() :- no of elements (size) in the queue.

(vii) display :- Display contents of elements of the queue.

↳ Algorithm to insert an element in queue :-

1) If REAR = MAX - 1
   write Overflow
   Go to step - 4
   [End of if]
2) If FRONT = -1 & REAR = -1
   Set FRONT = REAR = 0
   else
   set REAR = REAR + 1
   [End of if]
3) set QUEUE[REAR] = NUM
4) Exit

↳ Algorithm to delete an element from queue :-

1) If FRONT = -1 (or) FRONT > REAR
   write Underflow.
   else
   SRt VAL = QUEUE[FRONT]
   set FRONT = FRONT + 1
   [End of if]
2) Exit.

↳ Applications :-

- Simulation.
- multi programing platform system.
- Different type of scheduling algorithm.
- Round robin technique.
- printer server routines.
- Networking, routers & switches.
- sharing resources b/w CPU & disk scheduling.

Linked representation & implementation of queues :-

- A linked queue is a queue implemented as a linked list with 2 pointer variables FRONT & REAR pointing to the node which is in the FRONT & REAR of the queue.
- The INFO fields of the list hold the element of the queue & the LINK fields hold pointers to the neighboring elements in the queue.

Eg :-

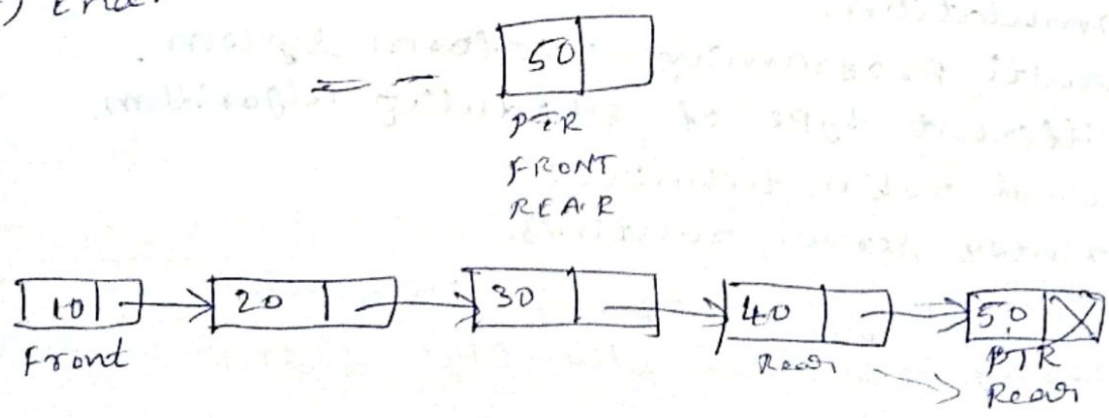
(1) FRONT = 0, REAR = MAX - 1

(2) REAR != MAX - 1

(3) FRONT != 0 & REAR = MAX - 1

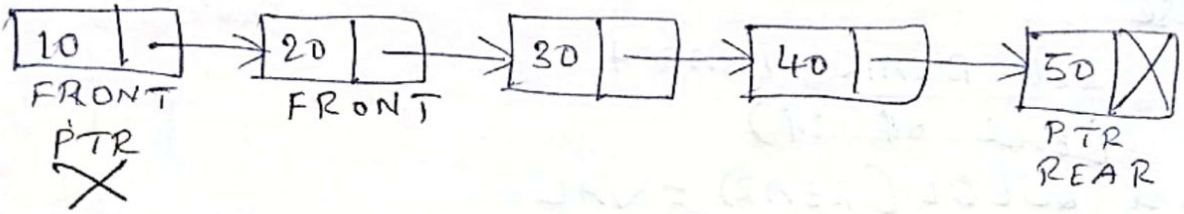
Algorithm for Insertion :

1) Allocate memory for the new node & new it as PTR.
2) Set PTR → DATA = VAL
3) If FRONT = NULL
   set FRONT = REAR = PTR
   Set FRONT → NEXT = REAR → NEXT = NULL
   else
   set REAR → NEXT = PTR
   set REAR = PTR
   Set REAR → NEXT = NULL
   (End of if)
4) End.



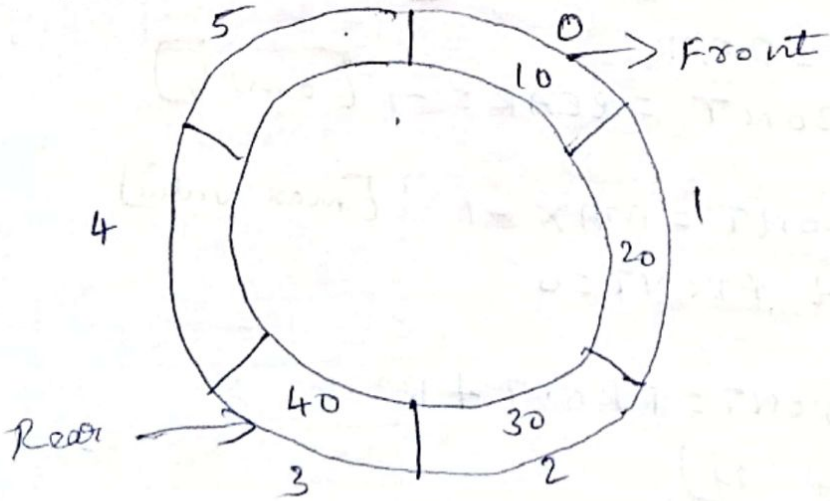
Algorithm for deletion:-

1) If FRONT = NULL
   Write "Underflow"
   Go to step-5
   [End of if]
2) Set PTR = FRONT
3) Set FRONT = FRONT → NEXT
4) FREE PTR
5) END



→ Types of queues :- (i) Simple (ii) Circular (iii) Dequeue (Head, Tail) (doubly linked list queue) (iv) Priority (v) Multiple

(ii) Circular queue :- It is a queue, in which all nodes are treated as circular such that the last node follows the 1st node.



* Algorithm of Insertion :-

1) If FRONT = 0 & REAR = MAX - 1 [Full]
   Write "overflow"
   go to step - 4
   [End of if]

2) If FRONT = -1 & REAR = -1 [empty]
   Set FRONT = REAR = 0
   Else if REAR = MAX - 1 & FRONT != 0
   Set REAR = 0
   Else
   set REAR = REAR + 1
   [End of if]

3) Set QUEUE[REAR] = VAL

4) EXIT.

* Algorithm of deletion :-

1) If front = -1 [Empty]
   Write "underflow"
   go to step - 4
   [End of if]

2) Set VAL = QUEUE[FRONT]

3) If FRONT = REAR

4) set FRONT = REAR = -1 [empty]

else

If FRONT = MAX - 1 [max value]
set FRONT = 0

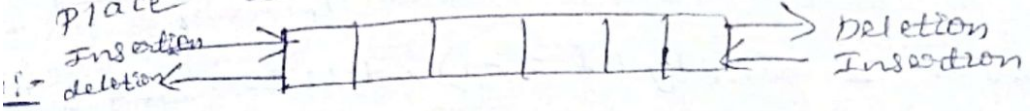
else

set FRONT = FRONT + 1
[End of if]

[End of if]

4) EXIT

(iii) **Dequeue :-** [Double-ended queue] is a queue in which insertion & deletion takes place at both ends.

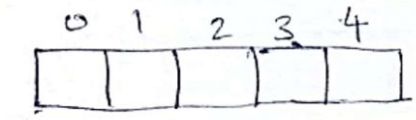


Types :-

[I/P restricted queue
  O/P restricted queue

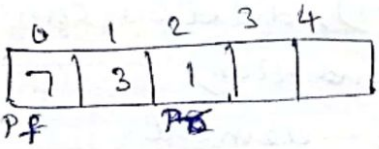
*** Operations of dequeue :-**

1) Take an array [dequeue] of size 'n'.
2) set 2 pts at first position & set front = -1 & rear = 0.

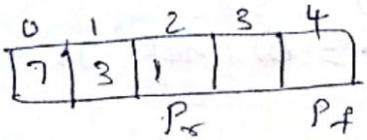


*** Insert at front :-**

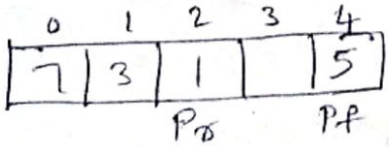
1) check position of front



2) If f < 1, reinitialise f = n - 1

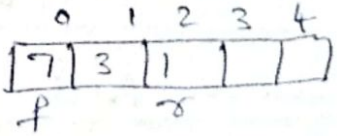


3) Else decrement front Pf.
4) Add new key [5] into array [front]

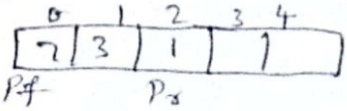


*** Insert at rear :-**

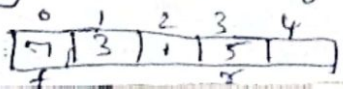
1) check if array is full.



2) If dequeue is f < 1, reinitialize rear = 0
3) Else, increment rear by 1:

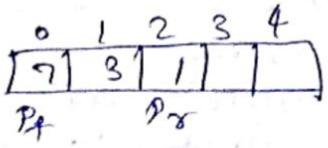


4) Add new key [5] into array [rear]

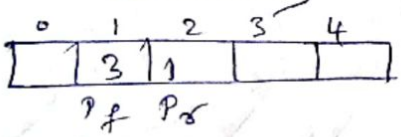


* Delete from Front :-

1) Check if deque is empty :

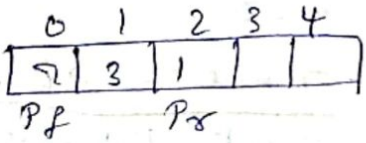


2) If deque is empty (f = -1), deletion can't be Performed (underflow condition)
3) If deque has only one element :
(f = r), set (f = -1 & r = -1)
4) Else if front is at end (f = n - 1) set go to front (f = 0)
5) Else, f = f + 1.

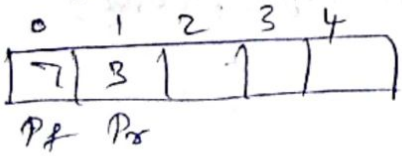


* Delete ~~oct~~ Rear :-

1) Check if deque is empty :



2) If deque is empty (f = -1), deletion can't be performed (underflow condition)
3) If deque has only one element
(f = r), set (f = -1 & r = -1)
4) If rear is at front (r = 0) set go to front (r = n - 1)
5) Else, r = r - 1



- **Check empty :-** f = -1
- **Check full :-** f = 0 & r = MAX - 1 (or)
f = r + 1

(iv) **Priority queue :-** It is a queue that contains items that have some preset priority.

- An element can be inserted or removed from any position depending on some priority.

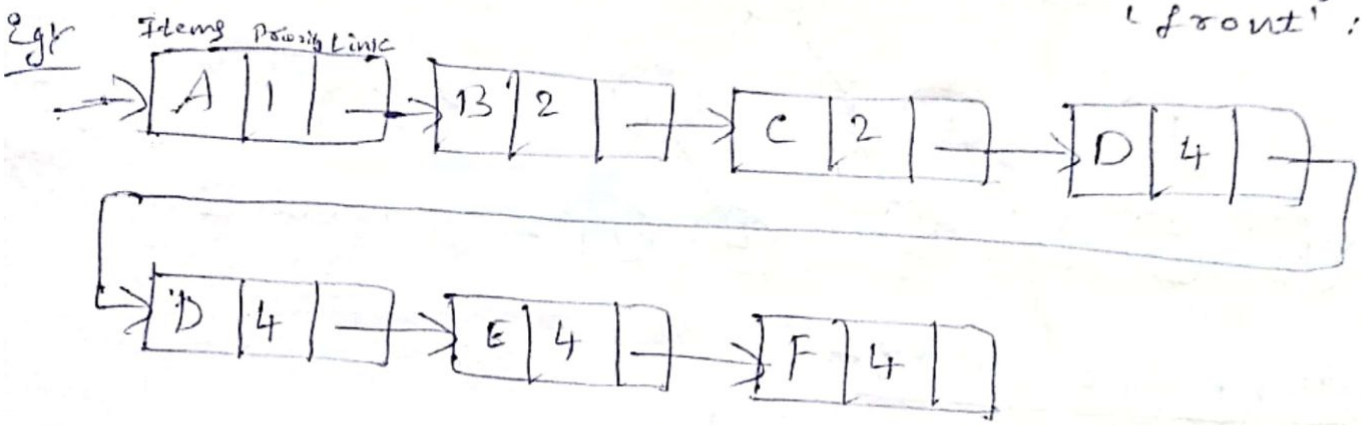
types : [Ascending Priority queue
Descending Priority queue

Eg: Items : A B G C D E F

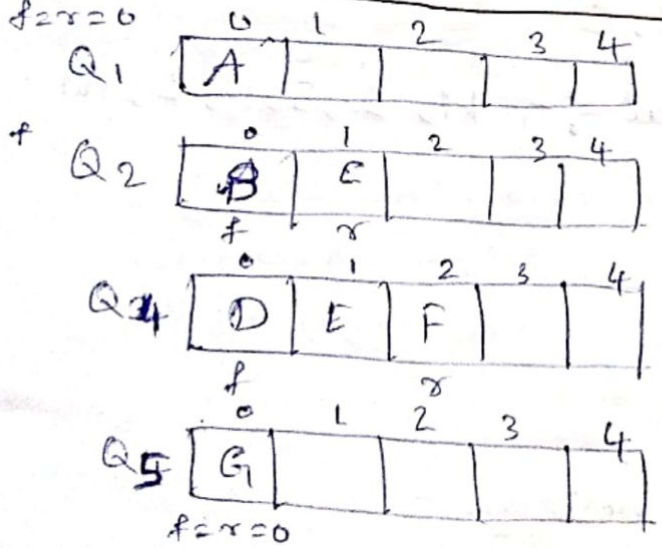
Priority : 1 2 5 2 4 4 4

* **Representing using Linked List :** In this, the last inserted node is always pointed by 'rear' & 1st node is always pointed by 'front'.

Eg:



* **Repreing using Array :** Array (multiple Q)



Rules of Priority queue :- An element of higher priority is processed before any element of lower priority.

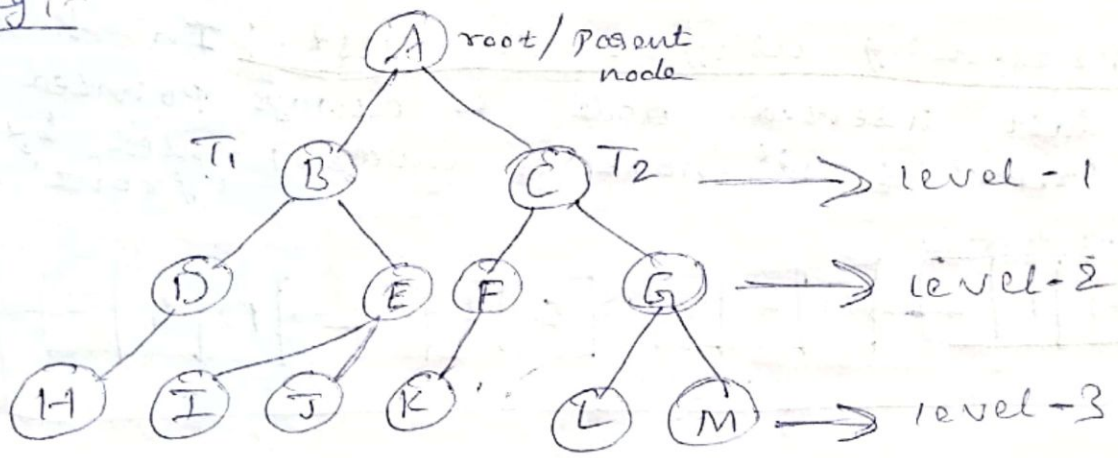
- Two element with same priority are processed according to the order in which they were added to the queue.

UNIT-3

BINARY TREES

- It is a data structure that is defined as collection of elements called nodes,
- In this, the top-most element is root node & each node has 0, 1 or atmost 2 children

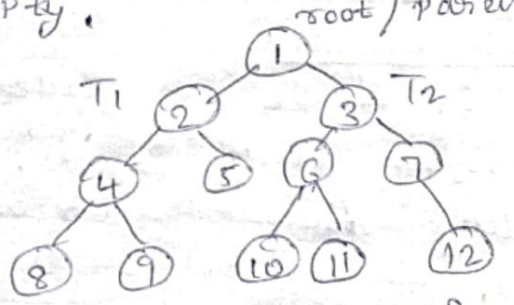
Eg:-



↳ Terminologies:-

1) Root node :- A (TOP most node)
2) sub-tree :- B, C (T1 & T2)
3) Leaf node / Terminal :- H, I, J, K, L, M (No children)
4) ~~Edge~~ Path :- Sequence of consecutive edges (A to M given as A, C, G & M)
5) ~~Path~~ Edge :- Line connecting a node 'N' to any of its successors.
6) Depth :- Length of path from roote 'R' to node 'N'. depth of root node is zero,
7) Height :- Total no of nodes on path from root to deepest node in tree. A tree with only a root node has a height of 1
8) Degree: InDegree :- Incoming node (no of edges arriving at that node)
9) OutDegree :- outgoing node (no of edges leaving that node)
10) Ancestor :- Predecessor node (A, B & D are ancestors of node H)
11) Descendent :- Successor node (B, D, H, E, I, J are descendants of node A)
12) Degree :- No of children that a node has. degree of leaf node is zero.

- A node that has 0 children is leaf / terminal node.
- Every node contains a data element, a left pointer which points to left child & right Pointer points to right child,
- The root Pointer is pointed by a root pointer, if a root = null, it means tree is empty.



Binary tree

- In above fig, 'R' is a root node & T1 & T2 are left & right sub-trees of 'R'.
- T1 is said to be the left successor of 'R', likewise, T2 is called right successor of 'R'.
- The left sub-tree of root node consists of nodes. 2, 4, 5, 8, 9. similarly, right sub-tree of root node consists of nodes, 3, 6, 7, 10, 11, 12,
- In a root node-1 has 2 successor: (2 & 3)
- Node-2 has 2 successor nodes: (5 & 4)
- Node-4 has 2 successor nodes: (8 & 9)
- Node-5 has 0 successor nodes:
- Node-3 has 2 successor nodes: (6, 7).
- Node-6 has 2 successor: (10, 11),
- Node-7 has 1 successor: (1, 2)
- A binary tree is recursive by definition as every node consists left & right subtree

even terminal contains empty left & empty right sub-trees,

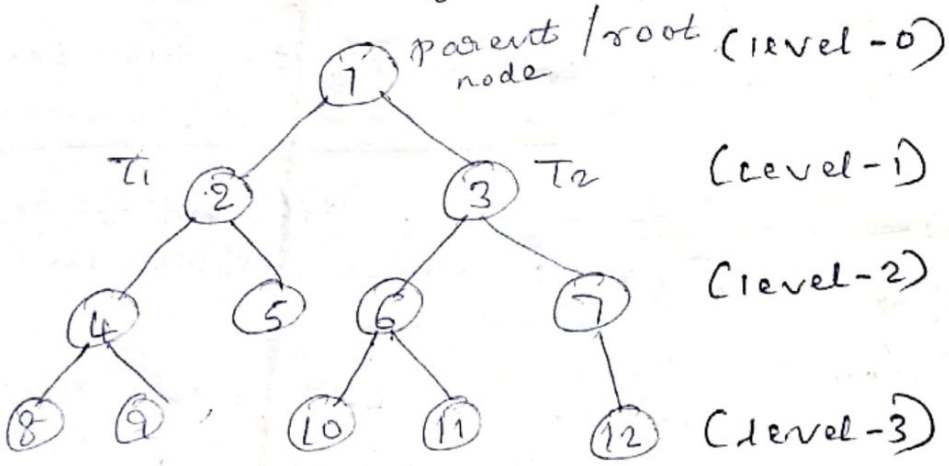
- Nodes 5, 8, 9, 10, 11, 12 has no successors & thus said to have empty sub-trees.

⇒ Terminologies :-

1) **parent** :- If 'N' is any node in 'T' and that has left successor S_1 & right successor S_2 then 'N' is called **parent** of S_1 & S_2. correspondingly S_1 & S_2 are called the left child & right child of 'N'.

- Every node other than the root has a parent node.

- Levels in binary trees :



- **Level no** :- Every node in binary tree is assigned a level no.
- The root node is defined to be at level-0.
- The left hand side & right hand side of node has level-1. Similarly, every node is at one level higher than its parent.
- So all child nodes are defined to have level no as parent level no + 1

2) **Degree of a node** :- It is equal to no of children that a node has.
- The degree of node is '0'.

Eg:- In the tree, degree of node-4 is 2, degree of node-5 is 0 & degree of node-7 is 1.

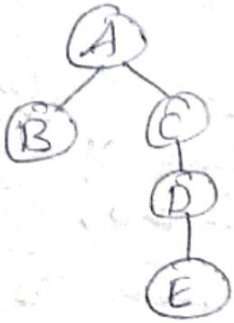
3) Siblings :- All nodes that are in same level & share same parents are called siblings (brothers).

Eg:- Nodes (2) & (3), nodes (4) & (5), nodes (6) & (5), nodes (8) & (9), nodes (10) & (11) are siblings.

4) Leaf node :- A node that has no children is called leaf node / terminal.

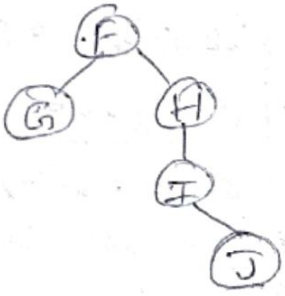
- The leaf nodes in the tree are: (8) (9) (5) (10) (11) (12).

Tree T

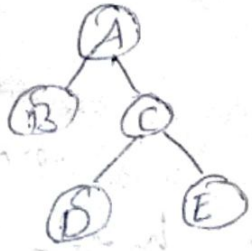


similar binary tree

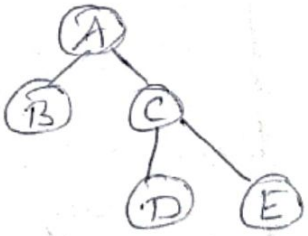
Tree T'



Tree T



Tree T'



'T' is copy of 'J'.

- 2 binary trees are copies, if they has similar structures & contents at corresponding nodes.

5) Edge :- It is a line connecting a node 'N' to any of its successors.

- A binary tree of 'n' nodes has exactly n-1 edges because every node except root node is connected to its parent via an edge.

6) Path :- A sequence of consecutive edges.
eg:- The Path from root node to node 8 is (1)(2)(4) & (8)

7) Depth :- The depth of 'n' is given as length of path from the root are to the node 'n', the depth of root node is '0'.

8) Height :- It is total no of nodes on the Path from root node to deepest node in the tree. The tree with root node is height of '1'.

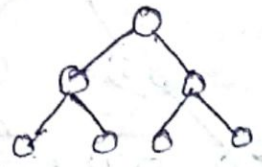
9) InDegree / Outdegree :- It is no of edges arriving at node, the root node is the only node that has in_degree = 0, similarly out_degree of a node is no of edges leaving that node.

10) Ancestor node :- It is any predecessor node on the Path from root to that node. The root node does not have any ancestors.

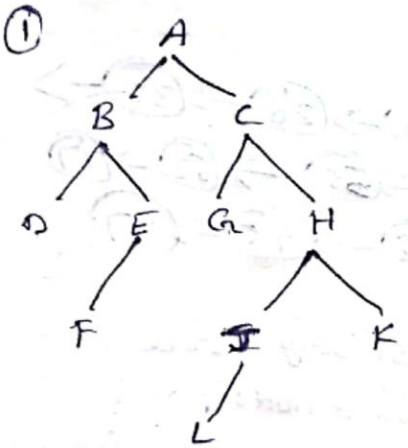
- Binary trees are commonly used to implement binary search trees, expression trees, tournament trees, binary heaps

11) Descendant node :- It is any successor node on any Path from the node to a leaf node. leaf nodes do not have any descendents.

↳ **complete binary trees :-** A binary tree is a full binary tree if every node has 0 or 2 children & when all the levels are completely filled.



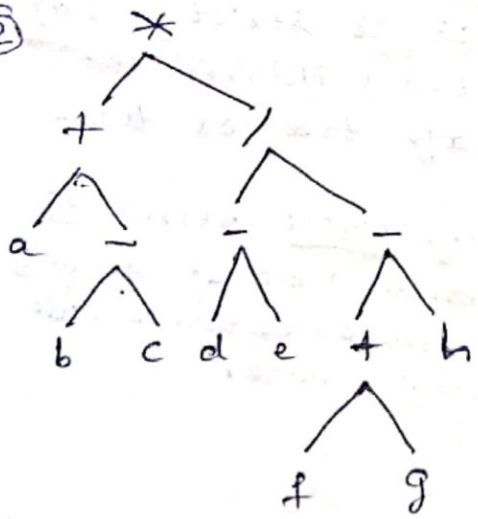
- To find childrens : $2 \times K \longrightarrow 2 \times K + 1$
- To find parentents : $\frac{K}{2} = \frac{4}{2} = 2$
- To find height : $Tn = \lceil \log_2(h+1) \rceil$
- Pre-order : Root → Left → right
- In-order : Left → root → right
- Post-order : Left → right → root

① 

preorder :- A → B → D → E → F → C → G → H → J → L → K

Inorder :- D → B → F → E → A → G → C → L → J → H → K

postorder :- D → F → E → B → G → L → J → K → H → C → A

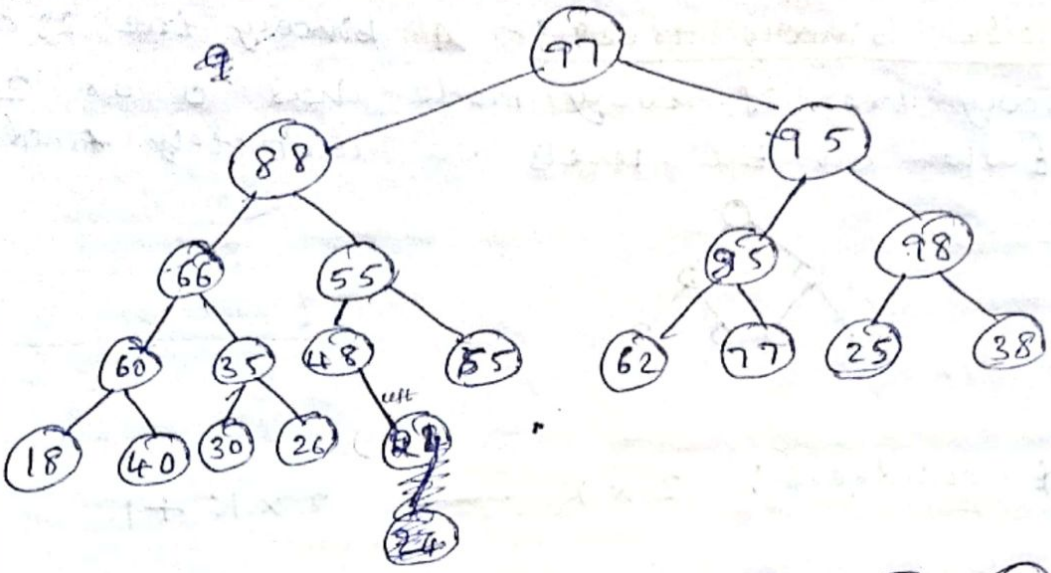
② 

pre-order :- * → + → a → - → b → c → / → - → d → e → - → + → f → h → g

Inorder :- a → + → b → - → c → * → d → - → e → / → f → + → g → + → h

postorder :- a → b → c → - → + → d → e → - → f → g → + → h → - → / → *

9.



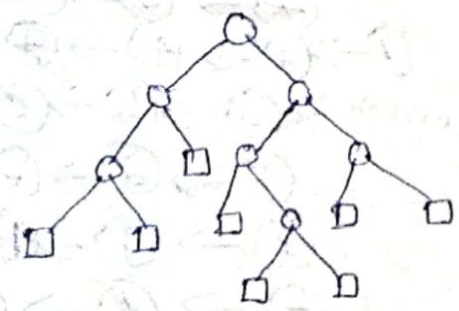
Pre-order :- 97 → 88 → 66 → 60 → 18 → 40 → 35 → 30 → 26 → 24 → 55 → 48 → 65 → 95 → 95 → 62 → 77 → 98 → 25 → 38

Inorder :- 18 → 60 → 40 → 66 → 30 → 35 → 26 → 24 → 48 → 55 → 65 → 97 → 62 → 85 → 77 → 95 → 25 → 98 → 38

Postorder :- 18 → 40 → 60 → 30 → 26 → 35 → 66 → 24 → 48 → 65 → 55 → 62 → 77 → 85 → 25 → 38 → 98 → 95 → 97

* Extended Binary tree :- A Binary tree which has atleast 0 or 2 childnode.
- The extended binary tree is a special form of a binary tree. If it is is strictly has either extended with dummy node to 0 or two children then it is called as strict binary tree or two tree.

Eg:-



Empty circle represents internal node & squares represents exctomal node (dummy node).

↳ Pre-order Traversal Algorithm :-

. PREORD(INFO, LEFT, RIGHT, ROOT).

- A binary-tree 'T' is in memory, the algorithm does a Pre-order traversal of 'T' applying an operation PROCESS to each of its nodes. & array STACK is used to temporarily hold address of nodes.

1) [Initially Push NULL onto STACK & initialise PTR]
   Set TOP := 1, STACK[1] = NULL and PTR = ROOT

2) Repeat step-3 to 5 while PTR != NULL.

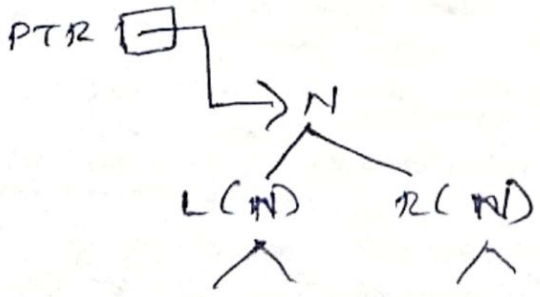
3) Apply PROCESS to INFO[PTR]

4) [Right child]
   If RIGHT[PTR] != NULL, then [Push on STACK]
   set TOP = TOP+1, and STACK[TOP] = RIGHT[PTR]
   [End of if structure]

5) [Left child]
   If LEFT[PTR] != NULL, then
   Set PTR = LEFT[PTR]
   else [POP from STACK]
   set PTR := STACK[TOP], set TOP = TOP-1
   [End of If structure]
   [End of step-2 loop]

6) Exit.

- Pre-order traversal algorithm uses a variable 'PTR' [Pointer] which will contain location of 'N' [node] currently being scanned.



- In the above diagram where L(N) denotes left child of node 'N' & R(N) denotes

right child.

- Algorithm also uses an array = STACK, which will hold addresses of nodes for further Processing.

Algorithm :

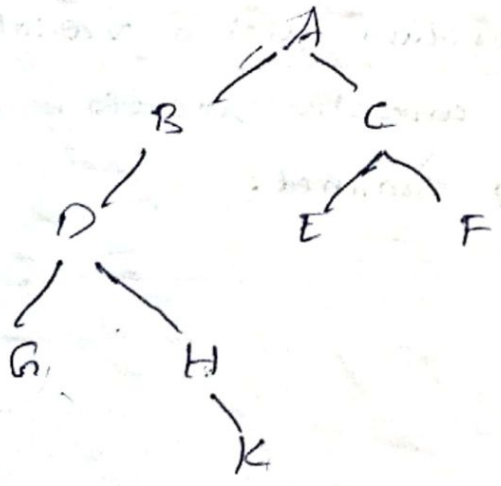
- Initially Push NULL on to stack & then set PTR = ROOT, then repeat following step until PTR = NULL or equivalently while PTR != NULL.

Step-1:- Proceed down the left most path rooted at PTR, Processing each node 'N' on Path & Pushing each right child R(N), if any, on to stack. Once traversing stop after a node N with no left child L(N) is Processed (Thus PTR is updated using assignment ~~'N' with no~~ PTR = LEFT [PTR] & traversing stops when LEFT [PTR] = NULL.

Step-2:- [Backtracking] Pop & assign PTR to the TOP element on STACK.

- If PTR = NULL then return to step-a, otherwise exit.

Eg:- consider a binary tree; will stimulate the above algorithm showing content of STACK at each time.



step-1:- Initially push null on to STACK

STACK : ϕ

then set PTR = A, the root of 'T'

step-2:- Proceed down left most path rooted at PTR = A as follows:

(i) Process A & push its right child 'C' on to stack.

STACK : ϕ, C

(ii) Process B (there is no right child)

(iii) Process D & push its right child 'H' on to STACK.

STACK : ϕ, C, H

(iv) Process 'G' (there is no right child)

step-3:- [Backtracking] Pop the top element 'H' & set PTR := H. This leaves:

STACK : ϕ, C

- Since PTR != NULL return to step-A of algo

step-4:- Proceed down the left most path rooted at PTR = H

PTR = H

step-5:- Process 'H' & push its right child 'K' on to stack.

STACK : ϕ, C, K,

- No other nodes is processed since (H) has no left child,

step-(6):- [Backtracking] Pop 'K' from STACK & PTR = K. This leaves:-

STACK : ϕ, C

- Since PTR != NULL, return to step-A of algorithm,

Step-6:- Proceed down the left most path rooted at PTR = K as follows:

- (VI) Process 'K' [No right child]

No other node is processed since 'K' has no left child,

Step-7:- [Backtracking] POP 'C' from stack & set PTR = C, this leaves;

STACK = ϕ.

- Since PTR != NULL returns to step-A of algorithm.

Step-8:- Proceed down the left most path rooted at PTR = C as follows;

- (VII) Process 'C' & Push its right child 'F' on on stack,

STACK = ϕ, F

(VIII) Process 'E' (There is no right child)

Step-9:- [Backtracking] POP 'F' from stack & set PTR = F, this leaves;

STACK = ϕ

- Since PTR != NULL, return to step-A of algorithm.

Step-10:- Proceed down the left most path rooted at PTR = F as follows:

- (IX) Process 'F' (No right child)
- No other node is processed since 'F' has no left child.

Step - 11:- [Backtracking] POP the top element NULL from stack set PTR = NULL. since PTR = NULL, algorithm is completed.

- from step- 2, 4, 6, 8 & 10, the nodes are processed in order, A, B, D, G, H, K, C, E, F.
- This is the required PRE-ORDER traversal of 'T'.

↳ In-order Traversal Algorithm:-

INORD(INFO, LEFT, RIGHT, ROOT)
This algorithm does an inorder traversal of 'T' by applying an operation PROCESS to each of its nodes.

1) [Push NULL onto STACK and Initialize PTR]
   Set TOP=1, STACK[1] = NULL and PTR = ROOT.
2) Repeat while PTR != NULL [Pushes left-most path onto STACK].
   a) Set TOP = TOP+1 and STACK[TOP] = PTR [Save node]
   b) Set PTR = LEFT[PTR] [Updates PTR] [End of LOOP].
3) Set PTR = STACK[TOP] and TOP = TOP-1 [POP node from STACK].
4) Repeat step-5 to 7 while PTR != NULL [Backtracking]
5) Apply PROCESS to INFO[PTR]
6) If RIGHT[PTR] != NULL then: [Right child]
   Set PTR = RIGHT[PTR].
   Go to step-3
   [End of if structure].
7) Set PTR = STACK[TOP] and TOP = TOP-1 [POP node]
   [End of LOOP]
8) Exit.

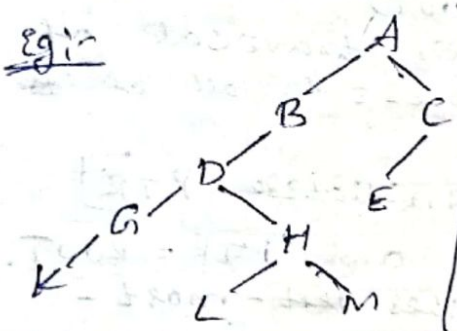
Algorithm:-

1) Initially push null on to stack & set PTR = root, then repeat following step until null is poped from stack.

a) Proceed down the left most path rooted at PTR pushing each node 'N' into STACK & stoping when a node 'N' with no left child is pushed on to stack.

Step-2:- [Backtracking] Pop & process the nodes on stack. If NULL is poped, then exit. If a node 'N' with right child R[N] is processed, set PTR = R[N]. & return to step-a.

Eg:-



1) Initially Push NULL on to STACK: STACK: ϕ then set PTR = A, the root of T,

2) Proceed down left most path rooted at PTR = A, pushing nodes A, B, D, G, K onto stack STACK: ϕ, A, B, D, G, K (No other node is pushed on to STACK, since 'K' has no left child)

3) [Backtracking] The nodes K, G and D are POPed & processed, leaving: STACK: ϕ, A, B (we stop processing at D, since D has a right child) Then set PTR = H, the right child of D.

4) Proceed down the left-most path rooted at PTR = H, pushing the nodes H and L on to stack: ϕ, A, B, H, L. (No other node is pushed on to stack, since L has no left ch

5) [Backtracking] The nodes L & H are POPed & Processed, leaving: STACK: ϕ, A, B (we stop Processing at H, since H has a right child) then set PTR = M, the right child of H.

6) Proceed down the left-most Path rooted at PTR = M, pushing node M on to stack: ϕ, A, B, M (No other node is Pushed on to stack, since 'M' has no left child)

7) [Backtracking] The nodes M, B & A are POPed & Processed, leaving: STACK: ϕ (No other element of stack is Poped, since A does have a right child). Set PTR = C, the right ch of A

8) Proceed down the left most path rooted at PTR = C, Pushing the nodes C & E on to STACK: ϕ, C, E

9) [Back tracking] Node 'E' is poped & processed. Since E has no right child, node C is POPed & processed. Since C has on right child, the next element, NULL is Poped from stack.

- The alg is now finished, Since Null is POPed from stack, As seen from steps 3, 5, 7, 9 nodes are Processed in order K, G, D, L, H, M, B, A, E, C.

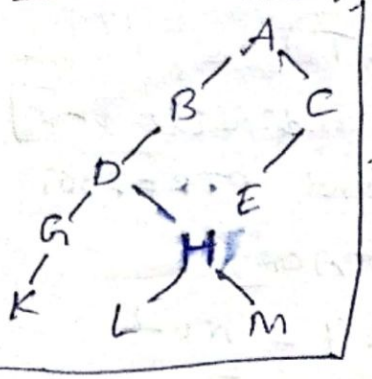
↳ Post-order Traversal algorithm:-

POSTORD(INFO, LEFT, RIGHT, ROOT)

1) [Push NULL onto STACK and initialize PTR]
   Set TOP = 1, STACK[1] = NULL and PTR = ROOT
2) [Push left most path into STACK]
   Repeat step 3 to 5 while PTR != NULL
3) set TOP = TOP + 1 and STACK[TOP] = PTR
   [pushes PTR on STACK]
4) If RIGHT[PTR] ≠ NULL then [push on STACK]
   Set TOP = TOP+1 and STACK[TOP] = -RIGHT[PTR]
   [End of If structure]
5) set PTR = LEFT[PTR] [updates pointer PTR]
   [End of step-2 loop]
6) set PTR = STACK[TOP] and TOP = TOP - 1.
   [POPS node from STACK].
7) Repeat while PTR > 0
   a) Apply PROCESS to INFO[PTR]
   b) set PTR = STACK[TOP] and TOP = TOP - 1
      [POPS node from STACK]
      [End of loop]
8) If PTR < 0 then;
   a) set PTR = - PTR
   b) Go to step-2
   [End of if structure]
9) Exit.

Algorithm:-

1) Initially push NULL on to stack & then set PTR = ROOT. Repeat the following step until null is poped from stack.
2) Proceed down the left most (push N on to STACK) path at PTR, at each node 'N' of path and if 'N' has a right child R[N] push -R[N] on to stack.
3) [Backtracking] Pop & process positive node on stack. If null is poped & then exit. If a negative node is poped, if PTR = -N for some node 'N' set PTR = N & return to step-2.



Consider again the binary tree T:

1) Initially push NULL onto STACK & set PTR = A, the root of T: STACK : ϕ

2) Proceed down the left most path rooted at PTR = A, pushing the nodes A, B, D, G & K onto stack, Furthermore, since A has right child C, push -C onto stack after A, but before B, & since D has a right child H, push -H onto stack after D but before G, This yields:

STACK : ϕ, A, -C, B, D, -H, G, K

3) [Backtracking] POP & process K & POP & process G. Since -H is negative, only POP -H, this leaves:

STACK : ϕ, A, -C, B, D Now PTR = -H, Repeat PTR = H & return to step-(a).

4) Proceed down left most path rooted at PTR = H, first push 'H' onto stack, since H has a right child M. Push -M on to stack, After H. Push L onto stack:

STACK : ϕ, A, -C, B, D, H, -M, L

5) [Backtracking] POP & Process L, but only POP -M, this leaves: STACK : ϕ, A, -C, B, D, H Now PTR = -M, reset PTR = M & return to step-a.

6) Proceed down left most path rooted at PTR = M. Now, only M is pushed onto STACK. This yields.:

STACK : ϕ, A, -C, B, D, H, M

7) [Backtracking] POP & Process M, H, D & B, but only POP -C, this leaves: STACK : ϕ, A Now: PTR = -C. Reset PTR = C & return to step-a.

8) Proceed down left most path rooted at PTR = C, first C is pushed onto STACK & then E, yielding:

STACK : ϕ, A, C, E

9) [Backtracking] POP & Process E, C & A, when NULL is poped, stack is empty & the algorithm is completed.

- As seen from 3, 5, 7 & 9, the nodes are processed in the order K, G, L, M, H, D, B, E, C, A. This is the required postorder traversal of binary tree T.

→ Binary Search Tree (BST):-

- BST is a tree in which 2 descendents taken like root left & right concept.
- If 'T' is a binary tree, then 'T' is called BST.
- If each node 'N' of 'T' has the following properties:

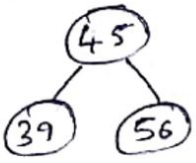
1) value of 'N' is greater than every value in the left subtree of 'N'.
2) value of 'N' is less than every value in the right subtree of 'N'.

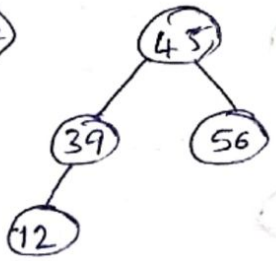
)Create a binary search tree (BST):

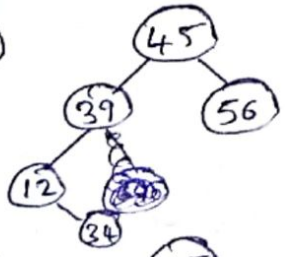
45, 39, 56, 12, 34, 78, 32, 10, 89, 54, 67, 81

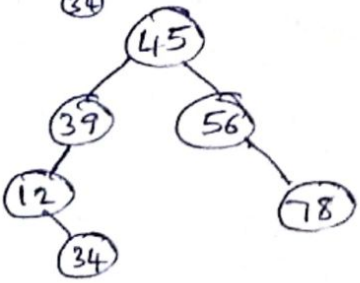
1)


2)


3)


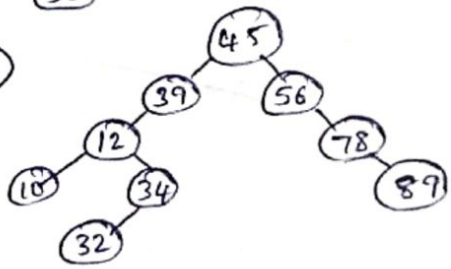
4)


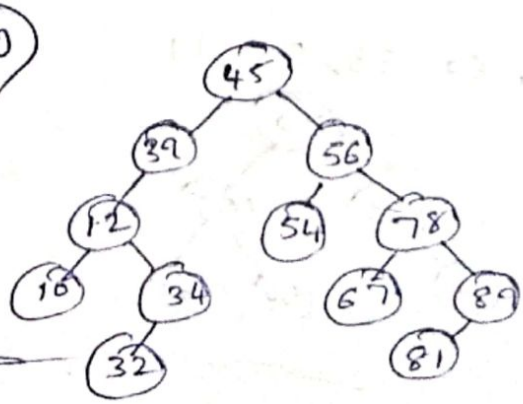
5)


6)


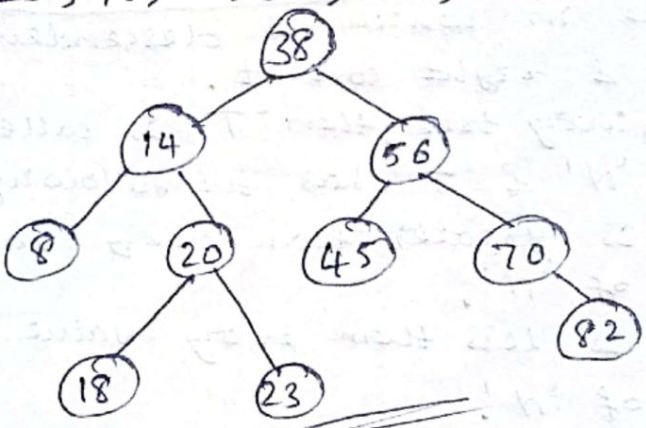
7)


8)

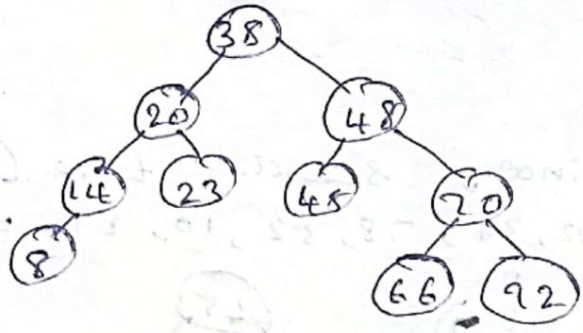

9)


10)


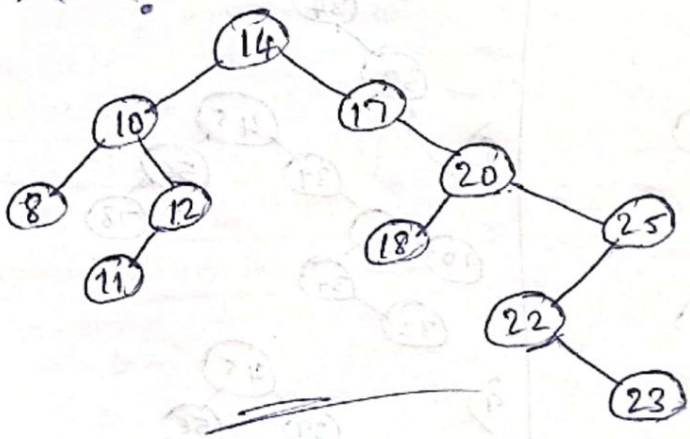
2) 8, 14, 23, 18, 20, 56, 45, 82, 70, 38.



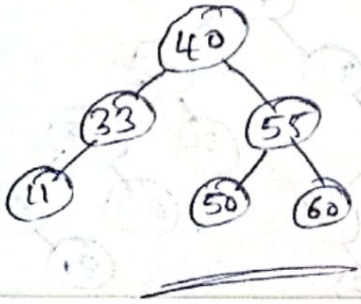
3) 8, 14, 23, 48, 20, 66, 45, 92, 70, 38



4) 14, 10, 17, 12, 10, 11, 20, 12, 18, 25, 20, 8, 22, 11, 23



5) 40, 60, 50, 33, 55, 11



→ Binary search Tree functions :-

(i) Searching / Finding a node.
(ii) Insertion
(iii) Deletion

(i) Searching / Finding a node :- [7.4]

FIND [INFO, LEFT, RIGHT, ROOT, ITEM, LOC, PAR]

1) [Tree empty ?]
If ROOT = NULL, then:
set LOC = NULL and PAR := NULL and Return

2) [ITEM at root?]
If ITEM = INFO[ROOT] then:
set LOC = ROOT and PAR = NULL; and Return.

3) [Initialize pointers PTR and SAVE]
If ITEM < INFO[ROOT], then:
set PTR = LEFT[ROOT] and SAVE = ROOT
else
set PTR = RIGHT[ROOT] and SAVE = ROOT
[End of if]

4) Repeat steps - 5 and 6 while PTR ≠ NULL

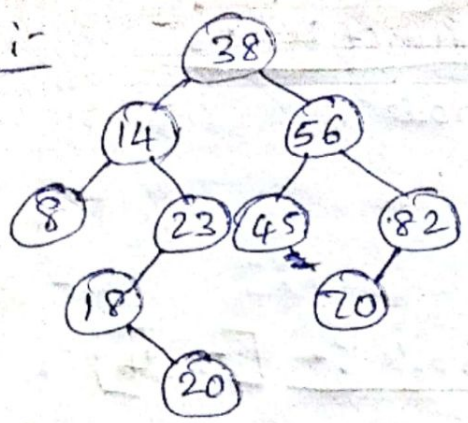
5) [ITEM Found ?]
If ITEM = INFO[PTR], then: set LOC = PTR and
PAR = SAVE, and Return.

6) If ITEM < INFO[PTR], then
set SAVE = PTR and PTR = LEFT[PTR]
else
set SAVE = PTR and PTR = RIGHT[PTR]
[End of if]
[End of step-4 loop]

7) [Search unsuccessful]
set LOC = NULL and PAR = SAVE

8) EXIT.

Eg :-



Item = 20

- Item < INFO[ROOT]
  20 < 38 ✓ then

set PTR = left[root]

Ptr = 14

- Save = root
  Save = 38

⊙ Item ≠ INFO[Ptr]
20 ≠ 14 ⊗

- ITEM < INFO[PTR]
  20 < 14 ⊗ then

set Save = Ptr & Ptr = right[Ptr]
Save = 14 & Ptr = 23

⊙ Item = INFO[Ptr]
20 ≠ 23 ⊗

- Item < info[Ptr]
  20 < 23 ✓

set Save = Ptr & Ptr = left[Ptr]
Save = 23 & Ptr = 18

⊙ Item = INFO[Ptr]
20 ≠ 18 ⊗

- Item < info[Ptr]
  20 < 18 ⊗

set Save = Ptr & Ptr = right[Ptr]
Save = 18 & Ptr = 20

⊙ Item = INFO[Ptr]
20 = 20 ✓

loc = Ptr = 20
pos = Save = 18

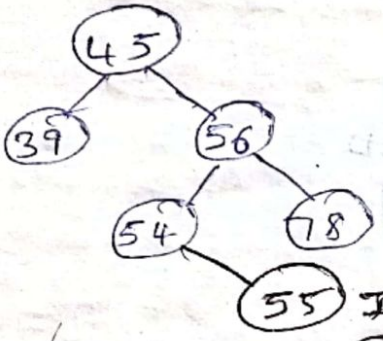
(ii) Insertion algorithm:-

INSERT (TREE, VAL)
(root)

1) If TREE = NULL
Allocate memory for TREE
Set TREE → DATA = VAL
Set TREE → LEFT = TREE → RIGHT = NULL
else
If VAL < TREE → DATA
INSERT (TREE → LEFT, VAL)
else
INSERT (TREE → RIGHT, VAL)
[End of if]
[End of if]

2) EXIT.

Eg:-

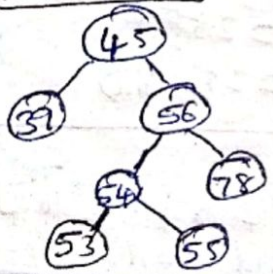


VAL = 55

Inserted node

(5) Tree = Null (X)
(val)
Set Tree → data = 55
Set tree → left = TREE → right = Null

VAL = 53



If val < tree → data
53 < 54 ✓
Insert (tree → left, val)
(53)

(4) Tree = Null ✓
(45)
Set tree → data = Val
Set tree → Left = Tree → right = Null
(both left & right side of tree is null

45 → root node (first node)

(iii) Deletion algorithm :- [no children]

Deleting a child :- [leaf node] [Procedure - 7.6]

a) CASE A (INFO, LEFT, RIGHT, ROOT, LOC, PAR)

1) [Initialize child]

if LEFT [LOC] = NULL and RIGHT [LOC] = NULL

then:

set CHILD = NULL

Else

if LEFT [LOC] ≠ NULL, then

set CHILD = LEFT [LOC].

Else

set CHILD = RIGHT [LOC]

[End if structure]

2) If PAR ≠ NULL then

if LOC = LEFT [PAR], then

set LEFT [PAR] = CHILD

else

set RIGHT [PAR] = CHILD

[End of if structure]

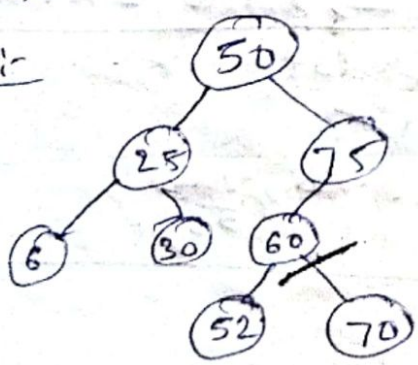
Else

set Root = CHILD

[End of if structure]

3) Return

Eg :-



Item = 75 (root which has (NULL) 2 child)

If left [loc] ≠ Null (not empty)

set child = 60

If par ≠ Null (not empty)

if loc = left [par] = 25

set left [par] = 25 (child)

Item = 70

Left [loc] = Null & right [loc] = Null

then, set child = Null (delete)

b) Deleting a node with 2-children: (7.7)

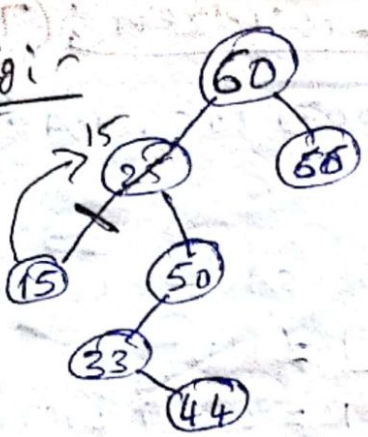
CASE B (INFO, LEFT, RIGHT, ROOT, LOC, PAR)

1) [FIND SUC and PARSUC]
   a) Set PTR = RIGHT[LOC] and SAVE = LOC
   b) Repeat while LEFT[PTR] ≠ NULL
      Set SAVE = PTR and PTR = LEFT[PTR]
      [End of loop]
   c) set SUC = PTR and PARSUC = SAVE

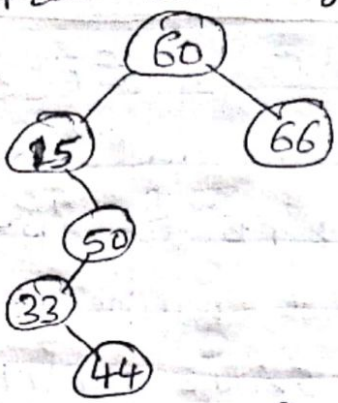
2) [DELETE inorder successor, using procedure 7.6]
   Call CASEA (INFO, LEFT, RIGHT, ROOT, SUC, PARSUC)

3) [Replace with N by inorder successor]
   a) If PAR ≠ NULL, then
      If LOC = LEFT[PAR], then
      set LEFT[PAR] := SUC
      Else
      set RIGHT[PAR] = SUC
      [End of if structure]
   Else
      set ROOT = SUC
      [End of if structure]
   b) set LEFT[SUC] = LEFT[LOC] and
      RIGHT[SUC] = RIGHT[LOC] & return.

Eg:-



After deleting - 25



Item = 25 (which has 2 children)

- Ptr = right[loc] & save = loc
  ptr = 50 & save = 25
- while left[Ptr] ≠ Null
  33 ≠ Null ✓
- set save = ptr & ptr = left[Ptr]
  save = 50 & Ptr = 33
  set suc = ptr & Parsuc = save
  suc = 33 & Parsuc = 50

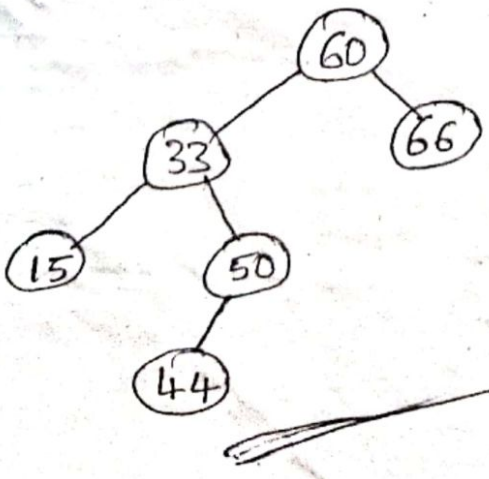
CALL CASE-A

- If left[loc] ≠ Null
  15 ≠ Null ✓, then
  set child = left[loc]
  child = 15
- If Par ≠ Null
  60 ≠ Null, then
  if loc = left[Par]
  25 = 25, then
  set left[Par] = Child
  left[Par] = 15

case - B (3)

If Par ≠ Null
60 ≠ Null, then
if loc = left[Par]
25 = 25, then
set left[Par] = Succ
left[Par] = 33
set left[suc] = left[loc]
& right[suc] = right[loc]
- left[suc] = 15 &
  right[suc] = 50

∴ After replacing N by its inorder successor;



c) Deleting a node which has only one child:-

Algorithm:-

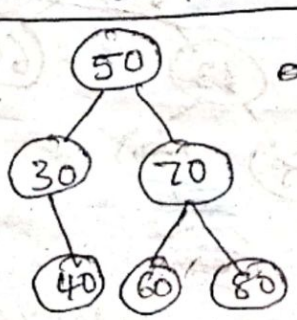
DEL (INFO, LEFT, RIGHT, ROOT, AVAIL, ITEM)

1) [Find location of ITEM and its Parents, using Procedure 7.4] (Searching alg)
   Call FIND (INFO, LEFT, RIGHT, ROOT, ITEM, LOC, PAR)
2) [ITEM in tree?]
   If LOC = NULL, then:
   write: ITEM not in tree and Exit.

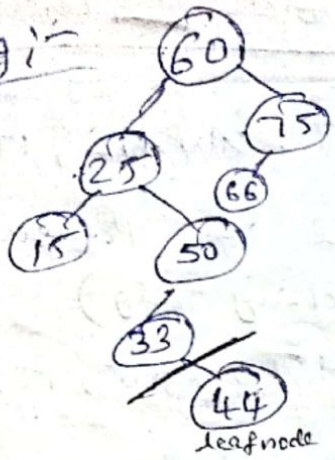
3) [Delete node containing ITEM]
   If RIGHT[LOC] ≠ NULL and LEFT[LOC] ≠ NULL, then
   CALL CASE B (INFO, LEFT, RIGHT, ROOT, LOC, PAR)
   Else
   CALL CASE A (INFO, LEFT, RIGHT, ROOT, LOC, PAR)
   [End of if structure]
4) [Return deleted node to the AVAIL list]
   Set LEFT[LOC] := AVAIL and AVAIL = LOC
5) EXIT.

⇒ SUC → gives loc of inorder successor.
⇒ PARSUC → gives location of parent of inorder successor
⇒ N → Having 2-children.

Eg:- delete(30)



Eg:-



ROOT [3]

AVAIL 7

	INFO	LEFT	RIGHT
1	33	8	10
2	2	8 0	10
3	60	2 1	4
4	66	0	0
5		0	
6		0	
7	75	5	0
8	15	0	0
9	44	50	0
10	50	0	0

delete node - 44 [case - A] [44] Direct

delete - (75)

Search loc :

Root = 60

PTR = Rightroot

PTR = 75, Save = 60

75 = 75, loc = ptr → [loc = 75, par = save = 60]

Deleting node with 1 - child :

A

child = Left [loc] = 66

if loc = left [par] = 25

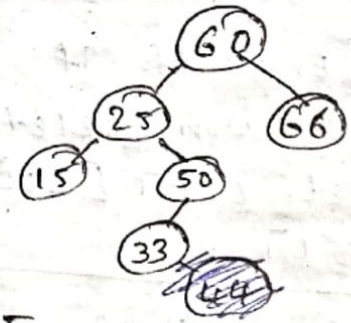
set Left [par] = 25 (child)

set Left [loc] = Avail

Avail = loc

Avail = 75 (deleted)

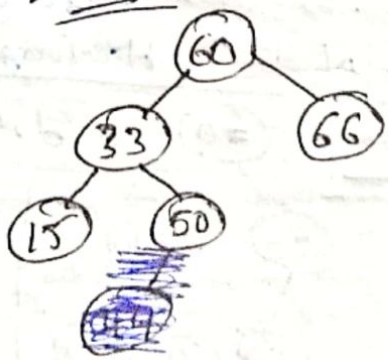
(Free)



[After deleting -75 [which has one child] [CASE - A & C]

[Finding location of node] 7.4

- ITEM ∈ INFO [ROOT]
  25 < 60 ✓ then
- Ptr = Left [root]
  ptr = 25
- Save = root
  save = 60
- Item = INFO [Ptr]
  25 = 25 ✓
- set loc = ptr = 25
  Par = save = 60



[After deleting - 25 [which has 2 child] [case - B]

→ Tree Sort :- It is a online sorting algorithm.

- It uses a binary search tree data structure to store elements
- It uses inorder traversal for retrival in sorted order of binary search tree.
- Since it is a online sorting algorithm elements inserting are always maintained in sorted order.

↳ Tree sort algoritum :-

- Let us assume that we have unsorted array : A[ ] containing 'n' elements.

* Tree sort :-

1) Build the BST by inserting element from the array in BST.
2) Perform inorder traversal on the tree to get the elements back in sorting order.

* insert ( ) :- Create BST node with a value equal to the array element A[i]

① Insert (node, key)

- If root = NULL, return newly formed node.
- If root → data < key,
  root → right = insert(root → right, key)
- If root → data > key, root → left = insert(root → left, key)

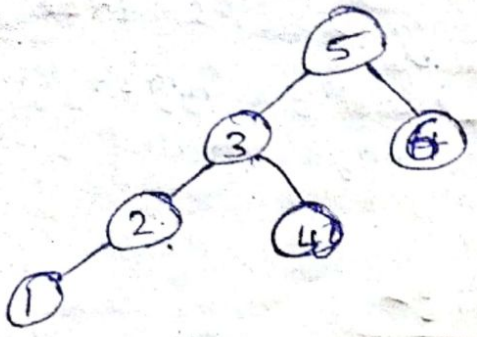
② Return the PTR to original root.

* Inorder ( ) :- Traverse left subtree.

- Visit root.
- Traverse right subtree.

* Tree sort example :-

- Suppose we have array {5, 3, 4, 2, 1, 6} we will sort it using insertion sort algorithm



- 1st we initialize BST by creating root node - 5,
- 3 is smaller than 5 so it is inserted in left of 5.
- 4 < 5 but larger than 3, so it is inserted in right of 3, but left of 4
- 2 is the smallest element in current tree so its get inserted at left most position
- 1 is the smallest element in current tree so its get inserted at left most position
- 6 is largest element in current tree so its get inserted at right most position
- After BST has been built we perform inorder traversal on tree to get final sorted array : {1, 2, 3, 4, 5, 6}

* Tree sort algorithm Complexity :-

1) Average case :- In this, time complexity inserting 'n' nodes in BST is of the order of $O(n \log n)$.

- It occurs when BST is form is a balanced BST. hence time complexity is of the order of $O(n \log n)$.

2) Worst case :- It occurs when array is sorted & an unbalanced BST having a maximum height of O(n) formed.
- It requires O(n) time for traversal & $O(n^2)$ for insertion.
- The worst case time complexity is : $O(n^2)$

3) Best case :- It occurs when BST formed is balanced.
- The best case time complexity is $O(n \log n)$
- It is same as average case time complexity.

* Space complexity :-
- Space complexity for this algorithm is : $O(n)$ because n-nodes have to be created for each element inside BST.

Eg:-

```
#include <bits/stdc++.h>
using namespace std;
class Node
{
  int key;
  Node *left, *right;
};
Node * NewNode (int item)
{
  Node *temp = new Node; Inserting new node
  temp -> key = item;
  temp -> left = temp -> right = NULL;
  return temp;
}
void inorder (Node *root, int arr[], int &i)
{
```

```
if(root != NULL)
{
 inorder(root -> left , arr, i);
 arr[i++] = root -> key;
 inorder (root -> right, arr, i);
}
Node * insertintoBST(Node * node, int key)
{
if(node == NULL) return new Node(key);
if(key < node -> key)
node -> Left = insertintoBST(node -> left, key);
else if(key > node -> key)
node -> right = insertintoBST(node -> right,
                              key);
return node;
}
void treeSort(int arr[], int n)
{
 Node *root = NULL;
 root = insertintoBST(root, arr[0]);
 for(int i=1; i<n; i++)
 root = insertintoBST(root, arr[i])
 int i=0;
 inorder(root, arr, i);
}
int main()
{
  int n=6;
  int arr[6] = {5, 3, 4, 2, 1, 6};
  cout << "input array";
  for(int i=0; i<n; i++);
    cout << arr[i] << " ";
}
cout << "\n";
treeSort(arr, n);
cout << "output array";
```

```
for(int i=0; i<n; i++)
{
    cout<<arr[i]<<" ";
}
cout<<"\n";
}
```

o/p :- 1 2 3 4 5 6.

→ Heap Sort :-

Atmost complete Binary Tree

All leads 'r' completely filled except leaf node

the node 'r' Inserted as left as possible

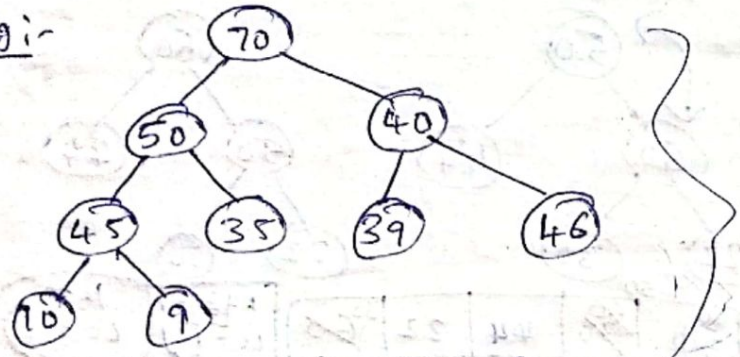
2 types :-

(1) max heap → Insertion, deletion

For every node i, the value of node is less than or equal to its parent value

$A[parent[i]] \geq A[i]$

eg :-



complete binary tree.

1	2	3	4	5	6	7	8	9
70	50	40	45	35	39	16	10	9

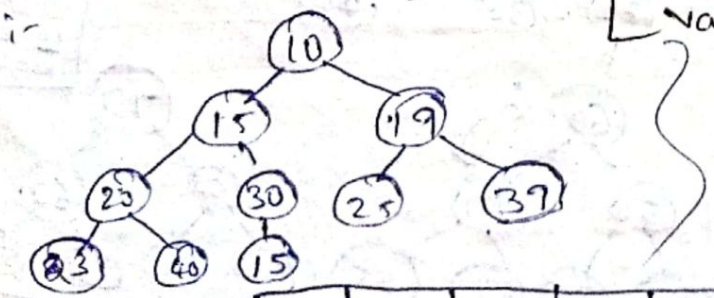
→ Arr =

(2) min heap →

$A[parent[i]] \leq A[i]$

For every node i, the value of node is greater than or equal to its parent value {except root node}

eg :-



complete binary tree

1	2	3	4	5	6	7	8	9	10
10	15	19	20	30	25	39	23	40	15

→ Arr =

Eg:- Inserting a node in a heap:

44, 30, 50, 22, 60, 55, 77, 55 → Build heap 'H'.

1) 

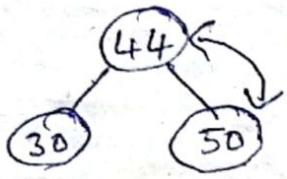
Item = 44

A = | 44 |
1

2) 

Item = 30

A = | 44 | 30 |
1 2

3) 

Item = 50

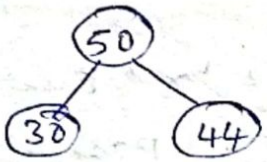
[After filling left, then we should insert into right]

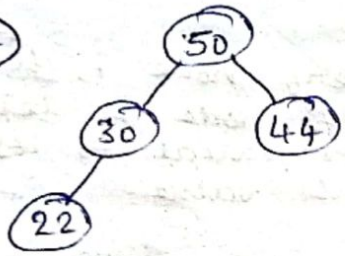
A = | 50 | 30 | 44 |
1 2 3

P = ⌊3/2⌋ = 1

LC = 2×K = 2

RC = 2×K+1 = 3



4) 

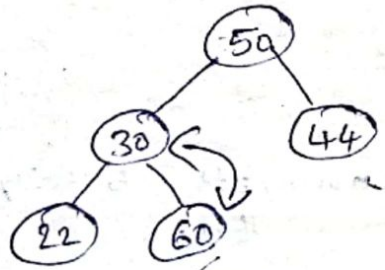
Item = 22

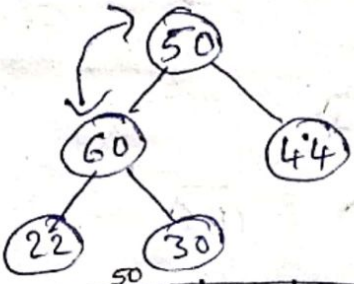
A = | 50 | 30 | 44 | 22 |
1 2 3 4

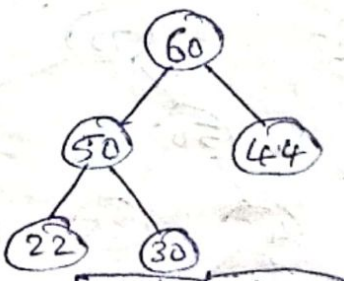
P = ⌊4/2⌋ = 2

LC = 2×K = 4

RC = 2×K+1 = 5

5) 



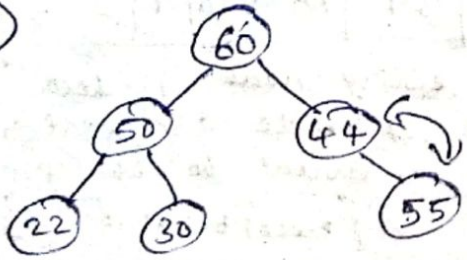


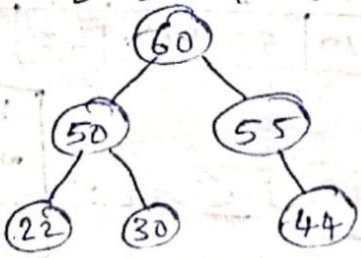
Item = 60

A = | 50 | 60 | 44 | 22 | 30 |
1 2 3 4 5

K = ⌊2/2⌋ = 1, LC = 1, RC = 2

K = 2, LC = 4, RC = 5

6) 



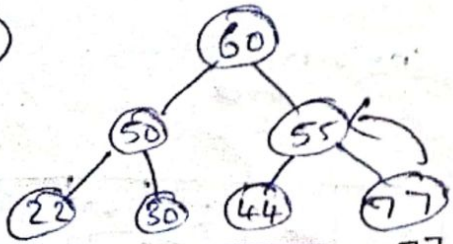
Item = 55

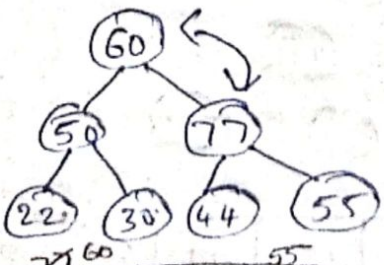
A = | 60 | 50 | 55 | 44 | 22 | 30 |
1 2 3 4 5 6

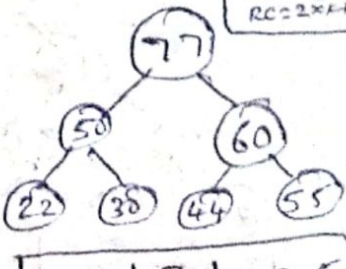
P = ⌊6/2⌋ = 3

LC = 2×K = 6

RC = 2×K+1 = 7

7) 





Item = 77

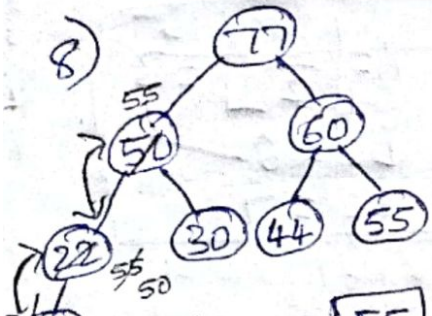
A = | 60 | 50 | 55 | 22 | 30 | 44 | 77 |
1 2 3 4 5 6 7

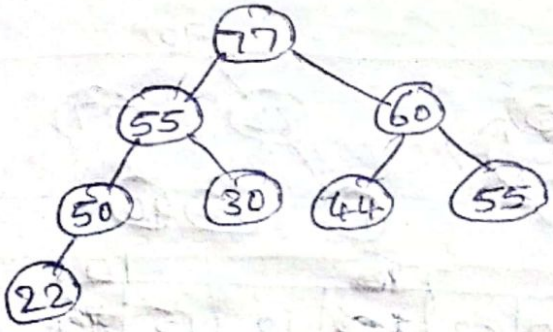
P = ⌊7/2⌋ = 3

LC = 6

RC = 7

8)





Item = 55

	55	50		22			
A=	77	~~50~~	60	~~22~~	30	44	55	~~55~~
	1	2	3	4	5	6	7	8

$P = \lfloor \frac{8}{2} \rfloor = 4$

$LC = 2 \times K = 2 \times 4 = 8$

$RC = 2 \times K + 1 = 9$

$P = \lfloor \frac{4}{2} \rfloor = 2$

$LC = 4$

$RC = 5$

Eg:- 82, 90, 10, 12, 15, 77, 55, 23

1) 82

A = | 82 |

1

2) 90

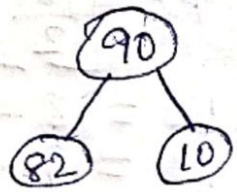
	90	82
A=	~~82~~	~~90~~
	1	2

$P = \lfloor \frac{2}{2} \rfloor = 1$

$LC = 2 \times 1 = 2$

$RC = 3$

3) 10



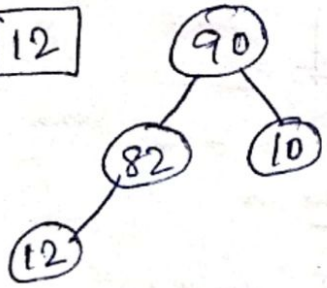
	90	82	10
A=			
	1	2	3

$P = \lfloor \frac{3}{2} \rfloor = 1.5$

$LC = 2$

$RC = 3$

4) 12



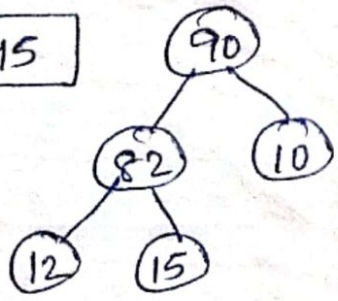
	90	82	10	12
A=				
	1	2	3	4

$P = \lfloor \frac{4}{2} \rfloor = 2$

$LC = 4$

$RC = 5$

5) 15



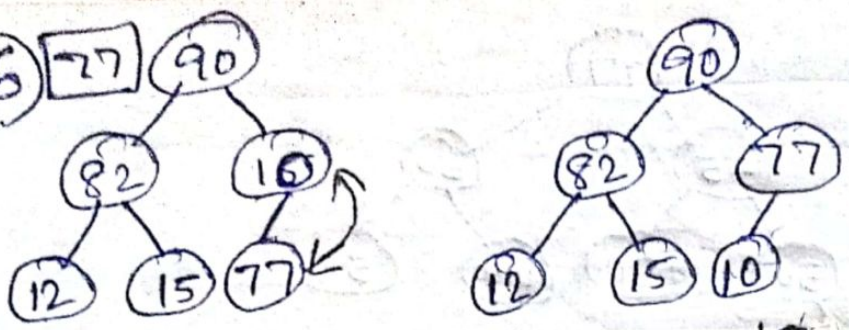
A = | 90 | 82 | 10 | 12 | 15 |

$P = \lfloor \frac{5}{2} \rfloor = 2.5$

$LC = 4$

$RC = 5$

6) 77



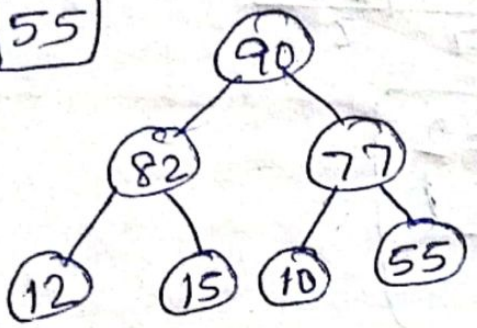
$P = \lfloor \frac{6}{2} \rfloor = 3$

$LC = 6$

$RC = 7$

	1	2	3	4	5	6
A =	90	82	~~10~~ 77	12	15	~~77~~ 10

7) 55



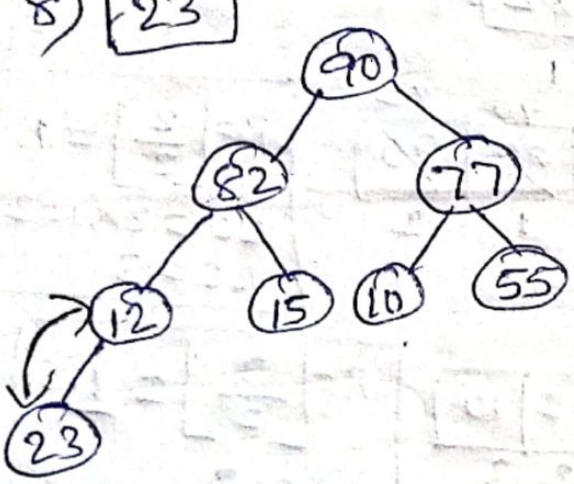
$P = \lfloor \frac{7}{2} \rfloor = 3.5$

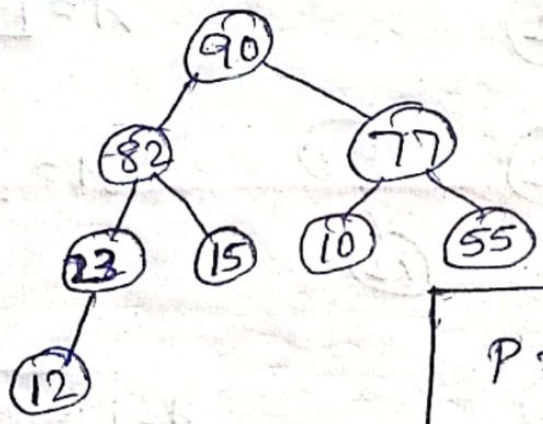
$LC = 6$

$RC = 7$

	1	2	3	4	5	6	7
A =	90	82	77	12	15	10	55

8) 23





$P = \lfloor \frac{8}{2} \rfloor = 4$

$LC = 8$

$RC = 9$

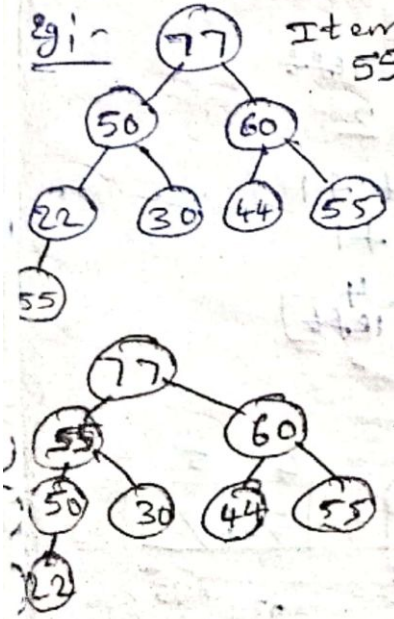
	1	2	3	4	5	6	7	8
A =	90	82	77	~~12~~ 23	15	10	55	~~23~~ 12

Algorithm :-

INSHEAP (TREE, N, ITEM)
A heap H with N elements is stored in the array TREE and an ITEM of information is given. This procedure inserts ITEM as a new element of H. PTR gives the location of ITEM as it is given in the tree, and PAR denotes the location of the parent of ITEM.

1) [Add new node to H and initialize PTR]
   Set N = N + 1 and PTR = N.
2) [Find location to insert ITEM]
   Repeat steps 3 to 6 while PTR > 1.
3) Set PAR = [PTR/2]. [Location of Parent node]
4) If ITEM ≤ TREE[PAR], then:
   Set TREE[PTR] = ITEM, and return.
   [End of if structure]
5) Set TREE[PTR] = TREE[PAR] [Moves node down]
6) Set PTR = PAR. [Updates PTR].
   [End of step-2 loop].
7) [Assign ITEM as the root of H]
   Set TREE[1] = ITEM.
8) Return.

Eg:-



Item = 55

1	2	3	4	5	6	7
77	50	60	22	30	44	55

N = 7
N + 1 = 8

P = 7/2 = 3.5 = 3
LC = 2×P = 6
RC = 2×P + 1 = 7

PTR = 8
8 > 1 ✓
Par = 8/2 = 4
55 ≤ 22 ✗
T[8] = 22
Ptr = Par = 4
Ptr = 4

PTR = 4
4 > 1 ✓
Par = 4/2 = 2
55 ≤ 50 ✗
T[4] = 50
Ptr = 2

2 > 1 ✓
Par = 2/2 = 1
55 ≤ 77 ✓
T[2] = 55
T[2] = 77
Ptr = 1

1	2	3	4	5	6	7	8
77	55	60	22	30	44	55	50

⇨ Deleting a node using heap sort

$A[P] \geq [i]$

	1	2	3	4	5
A=	95	85	70	55	33

$K = 1$

$LC = 2 \times K = 2$

$RC = 2 \times K + 1 = 3$

Item = 95
last = 33
33 ≥ 85 ✗
33 ≥ 70 ✗

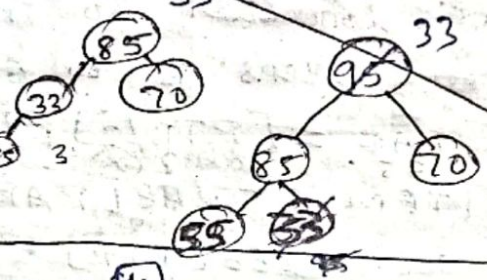
N = N − 1
= 5 − 1
= 4

ptr = 1
left = 2
right = 3
70 ≤ 85 ✓

~~left = N~~
~~left = 4~~
~~last = tree[left]~~
~~33~~

Tree[ptr] = 85 ; ptr = 3
~~Tree[ptr] = tree[right]~~
tree[ptr] = 33

	1	2	3	4	5
A=	95	85	70	55	33
	33			95	55 3
A=	33	85	70	55	95



1) item = Tree[1]
item = 95

2) last = tree[N]
last = 33

3) N = N − 1
N = 5 − 1
= 4

4) ptr = 1,
left = 2
Right = 3
3 ≤ 4 ✓
right ≤ N (3)

5) Last ≥ Tree[left (2)] & Last ≥ tree[right]
33 ≥ 85 ✗ & 33 ≥ 70 ✗

6) If Tree[right (3)] ≤ tree[left (2)]
70 ≤ 85 ✓
set Tree[ptr] = tree[left (2)] & ptr = left
Tree[ptr] = 85 & ptr = 2

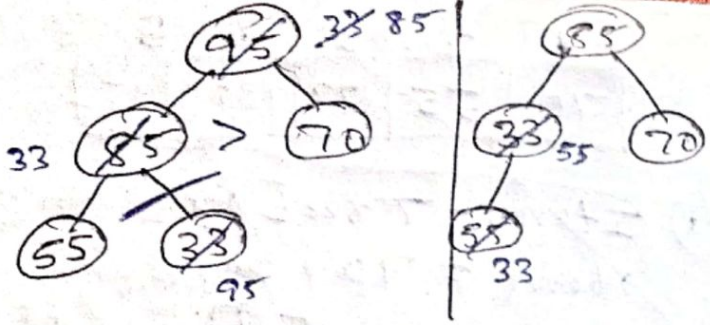
7) set left = 2 * ptr
= 2 * 2 = 4
Right = left + 1
= 4 + 1
= 5

8) If left = N & if last < tree[left (4)]
4 = 4 ✓ & 33 < 55 ✓
set ptr = left
ptr = 4

9) set Tree[ptr] = last
Tree[ptr] = 33

1	2	3	4	5
95	85	70	55	33
33	33		33	95
85	55			

85	55	70	33	95
1	2	3	4	5



1) Item = Tree[1]
   item = 85
2) Last = Tree[N]
   Last = 33
3) N = N - 1
   = 4 - 1
   = 3

4) ptr = 1
   left = 2
   right = 3
   right ≤ N
   3 ≤ 3 ✓

5) Last ≥ Tree[left] & last ≥ Tree[right]
   (33 ≥ 55) ✗ & (33 ≥ 70) ✗

6) If Tree[right] ≤ Tree[left]
   (70 ≤ 55) ✗

   set Tree[ptr] = Tree[right] & Ptr = right
   Tree[ptr] = 70 & ptr = 3

7) set left = 2 * ptr
   = 2 * 3 = 6

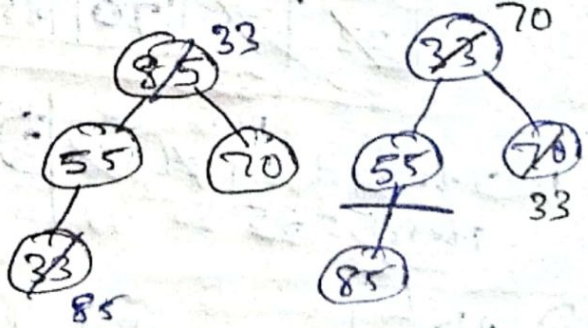
   Right = left + 1
   = 6 + 1
   = 7

8) If left = N & if last < tree[left]
   6 ≠ 3 & (33 < 55) ✓
   set Ptr = left → ptr = 2

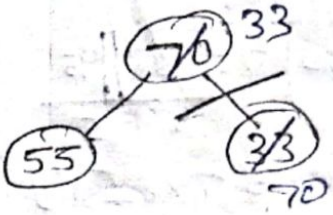
9) set tree[ptr] = Last
   tree[ptr] = 33

1	2	3	4	5
~~85~~	55	70	~~33~~	95
33		ptr	85	

1	2	3	4	5
~~33~~	55	~~70~~	85	95
70	left (ptr)	33 right ptr		
5	6	7		

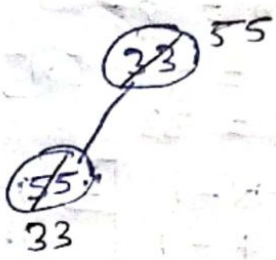


1	2	3	4	5
70	55	33	85	95



1) Item = Tree[1]
   Item = 70

2) Last = Tree[N]
   Last = 33



3) N = N-1
   N = 3-1
   = 2

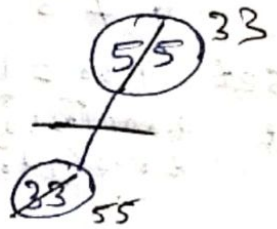
4) Ptr = 1
   Left = 2
   Right = 3
   Right ≤ N
   3 ≤ 2 (X)

8) If Left = N & if Last < Tree[Left]
   2 = 2 ✓ & 33 < 55 ✓
   Set Ptr = Left
   Ptr = 2

9) Set Tree[Ptr] = Last
   Tree[Ptr] = 33

1	2	3	4	5
~~70~~	55	~~33~~	85	95
33	Ptr	70		

1	2	3	4	5
~~33~~	~~55~~	70	85	95
55	33			
	Ptr			



1) Item = Tree[1]
   Item = 55

2) Last = Tree[N]
   Last = 33

3) N = N-1
   = 2-1
   = 1

4) Ptr = 1
   left = 2

8) If left = N & if last < Tree[left]
   2 ≠ 1 (X) & 33 < 33 (X)

9) Set Tree[Ptr] = last
   Tree[Ptr] = 33

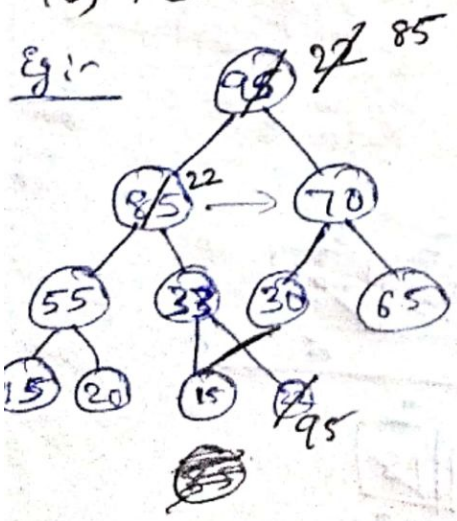
1	2	3	4	5
~~55~~	~~33~~	70	85	95
33	55			

1	2	3	4	5
33	55	70	85	95

DELHEAP (TREE, N, ITEM) :-

1) Set ITEM = TREE[1] [Removes root of H]
2) Set LAST = TREE[N] and N = N - 1 [Removes last node of H]
3) Set PTR = 1, LEFT = 2 and RIGHT = 3
   [Initialises pointers]
4) Repeat step - 5 to 7 while RIGHT ≤ N
5) If LAST ≥ TREE[LEFT] and LAST ≥ TREE[RIGHT]
   then: Set TREE[PTR] = LAST and Return
   [End of if]
6) If TREE[RIGHT] ≤ TREE[LEFT] then:
   Set TREE[PTR] = TREE[LEFT] and PTR = LEFT
   Else
   Set TREE[PTR] = TREE[RIGHT] and PTR = RIGHT
   [End of if]
7) Set LEFT = 2 * PTR and RIGHT = LEFT + 1
   [End of step-4 loop]
8) If LEFT = N and if LAST < TREE[LEFT], then
   set PTR = LEFT
9) Set TREE[PTR] = LAST.
10) Return.

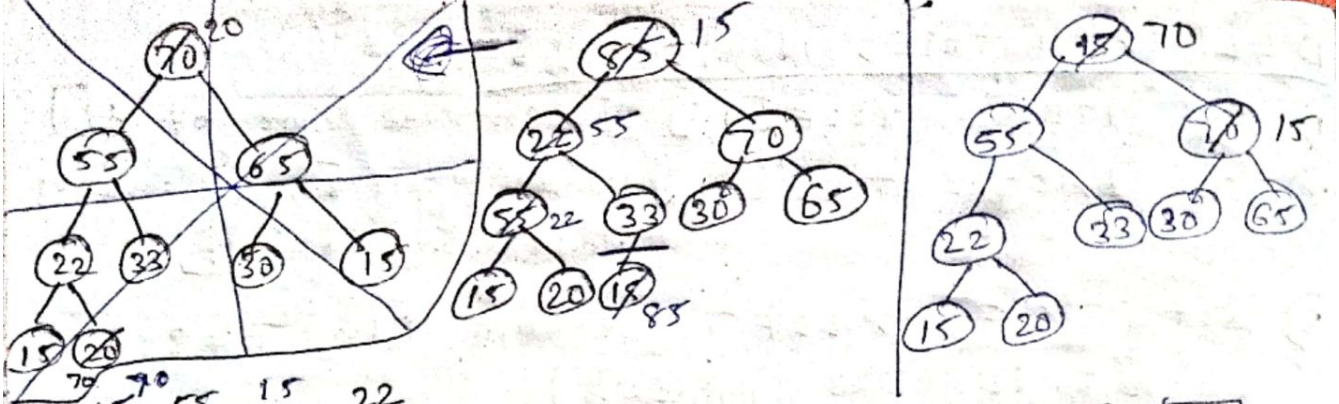
Eg:-



	95	85	70	55	33	30	65	15
	1	2	3	4	5	6	7	8

20	15	95
9	10	

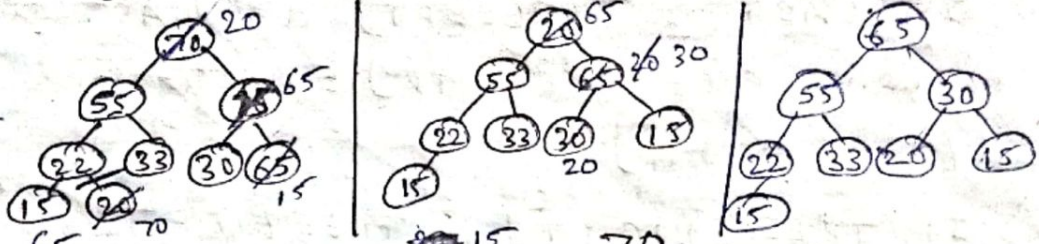
K = 1
LC = 2 × K = 2
RC = 2 × K + 1 = 3



1	2	3	4	5	6	7	8	9	10	11
85	55	70	22	33	30	65	15	20	85	95

85

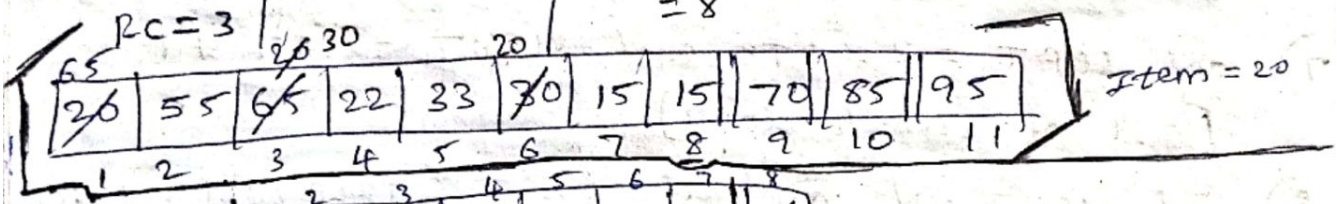
K = 1
Lc = 2
Rc = 3



1	2	3	4	5	6	7	8	9
70	55	15	22	33	30	65	15	20	85	95

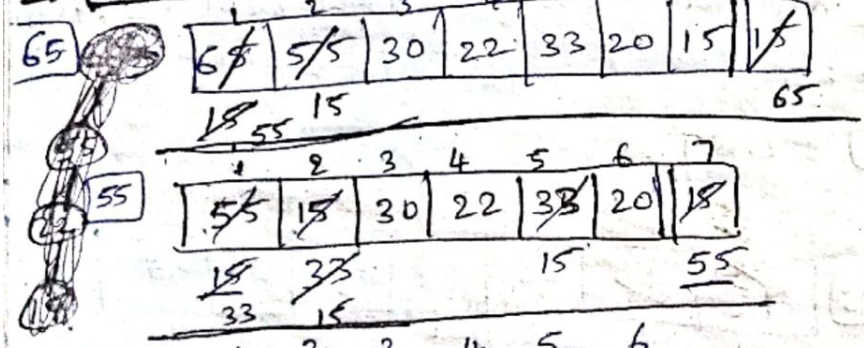
70

K = 1 | Item = 70 | N = N-1
Lc = 2 | Last = 20 | N = 9-1 = 8
Rc = 3



1	2	3	4	5	6	7	8	9	10	11
20	55	65	22	33	30	15	15	70	85	95

Item = 20



1	2	3	4	5	6	7	8
65	55	30	22	33	20	15	15

65

1	2	3	4	5	6	7
55	15	30	22	33	20	15

55

1	2	3	4	5	6
33	15	30	22	15	20

33

1	2	3	4	5
30	22	20	15	15

30

1	2	3
22	15	15

22

1	2
15	22

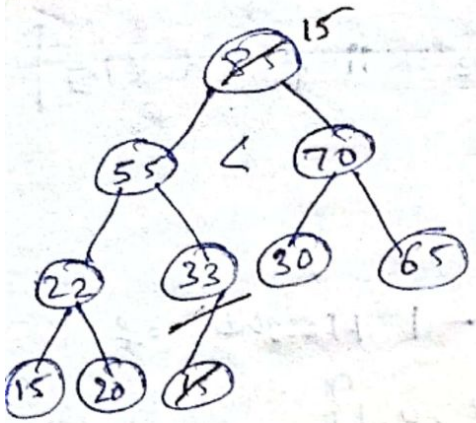
20

1	2	3	4
22	15	20	15

15

1
15

Empty



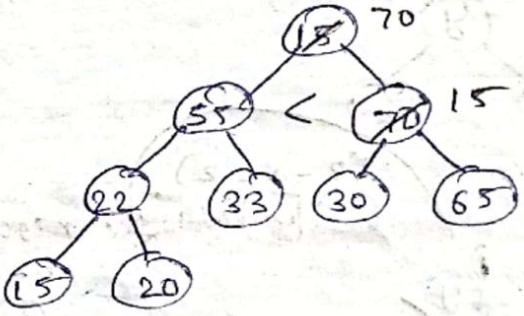
85	55	70	22	33	30	65	15	20	15
1	2	3	4	5	6	7	8	9	10

Item = 85 | N = N − 1 | ptr = 1
Last = 15, | N = 10 − 1 = 9 | Left = 2
| | Right = 3

3 ≤ 9 ✓

15 ≥ 55 ✗

70 ≤ 55 ✗



i.e., T[1] = 70, ptr = 3

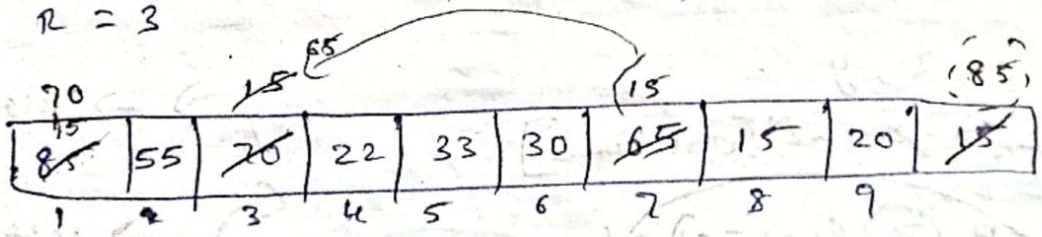
Left = 2 × 3 = 6,

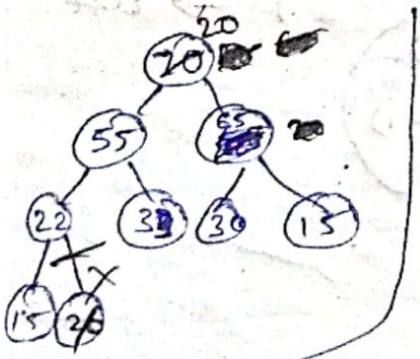
Right = 7

ptr = 1

L = 2 × 1 = 2

R = 3



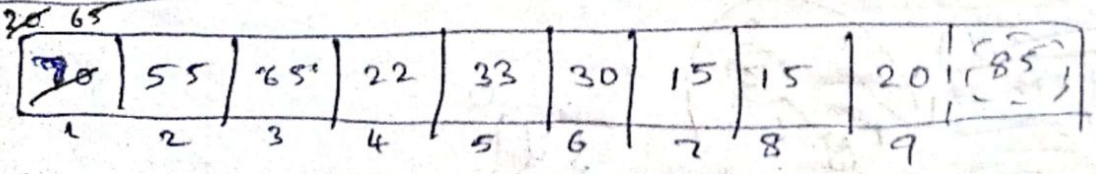


Item = 70 | N = 9 − 1 | 3 ≤ 8 ✓

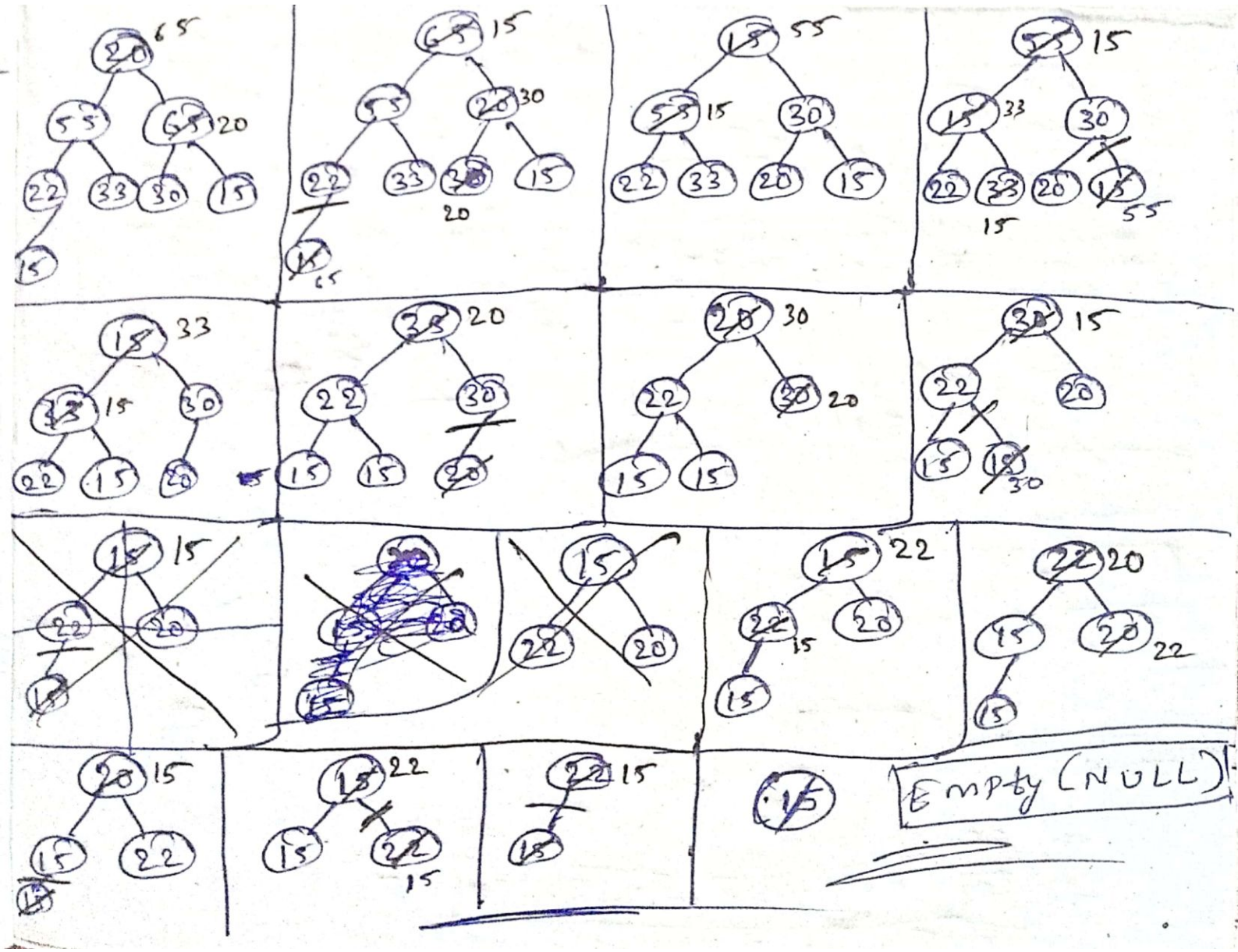
Last = 20, | N = 8.

20 ≥ 55 ✗ | T[1] = 65, ptr = 3

65 ≤ 55 ✗ | last = 6, R = 7.



→ until 7 iteration.



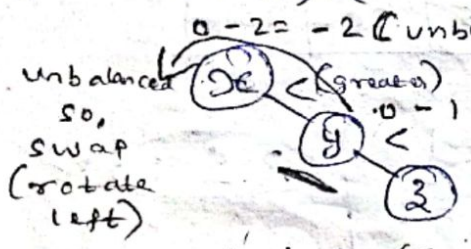
→ [Height balanced Tree] AVL Trees :- [Adelson velskii & Landis]

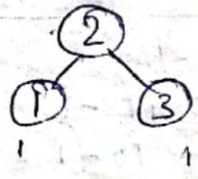
[Balanced Tree]

1) BST TREE

2) |Height of left subtree| − |Height of right subtree| = {−1, 0, 1}

Eg :- x, y, z

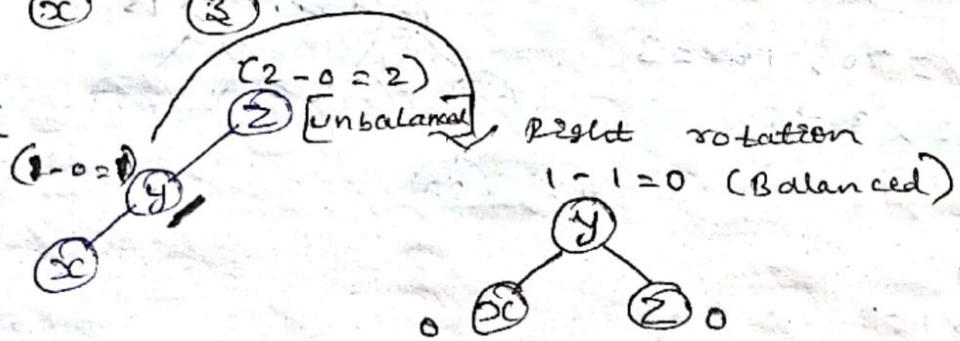




1 − 1 = 0 (Balanced)

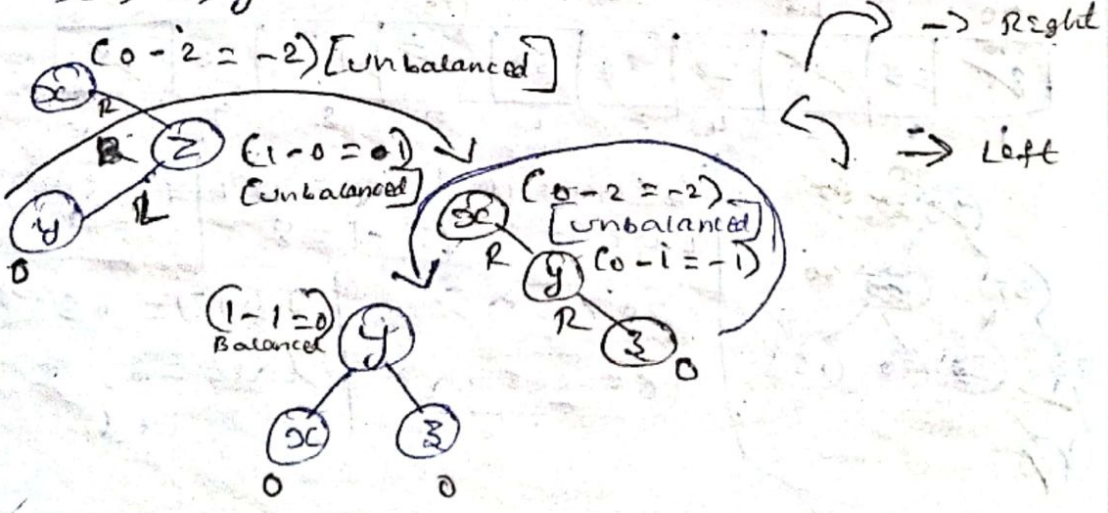
1 − 1 = 0 (Balanced)

Eg :-

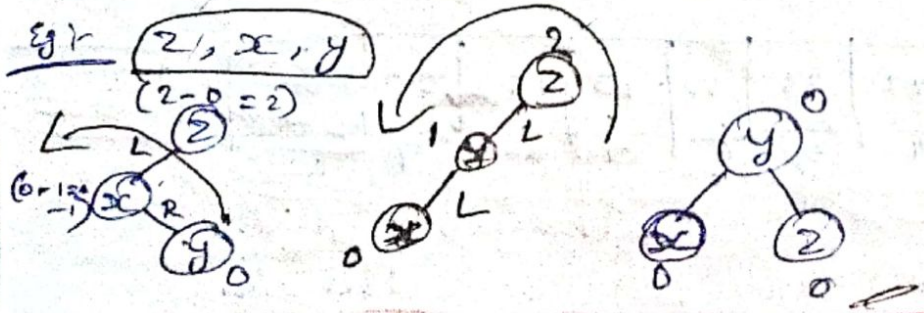


Eg :-

x, z, y



Eg :- z, x, y



- AVL tree can be defined as height balanced binary search tree in which each node is associated with a balance factor which is calculated by subtracting the height of its right sub-tree from that of its left sub-tree.
- Tree is balanced, if balance factor of each node is in b/w -1 to 1, otherwise the tree will be unbalanced & need to be balanced.

1 → The left sub-tree level is higher than the right sub-tree.

0 → The left sub-tree & right sub-tree contain equal height.

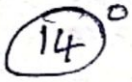
-1 → the left-subtree level is lower than the right sub-tree.

* 4 situation to make unbalanced tree to balanced tree:

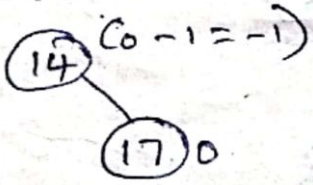
1) LL rotation:- Inserted node is in left subtree of left subtree.
2) RR rotation:- Inserted node is in right subtree of right subtree.
3) LR rotation:- Inserted node is in right subtree of left subtree
4) RL rotation:- Inserted node is in left subtree of right subtree

Eg:- 14, 17, 11, 7, 53, 4, 13, 12, 8, 60, 19, 16, 20

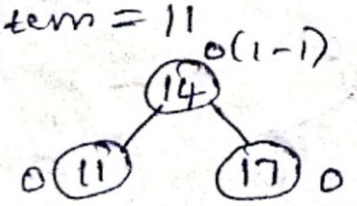
1) Item = 14



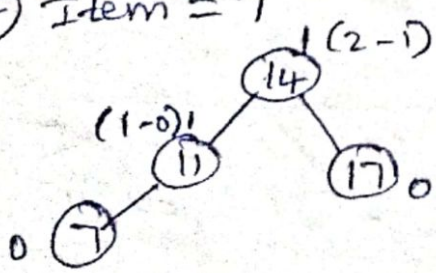
2) Item = 17



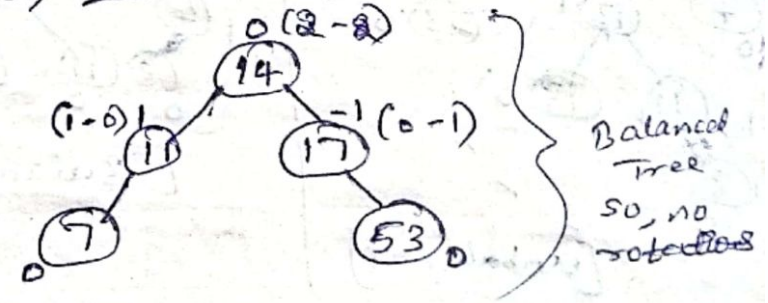
3) Item = 11



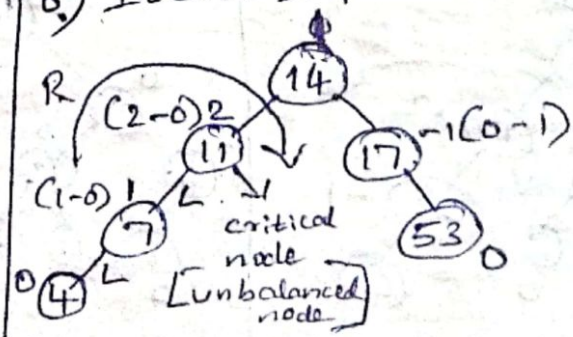
4) Item = 7



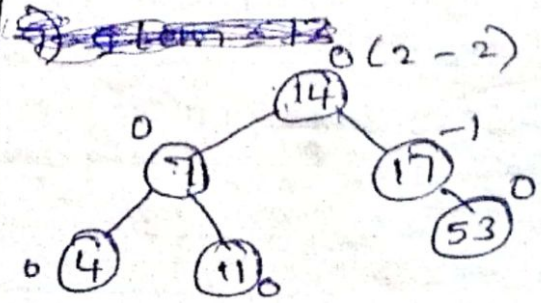
5) Item = 53



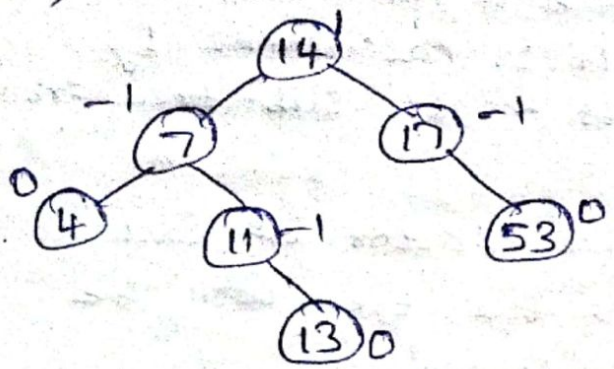
6) Item = 4



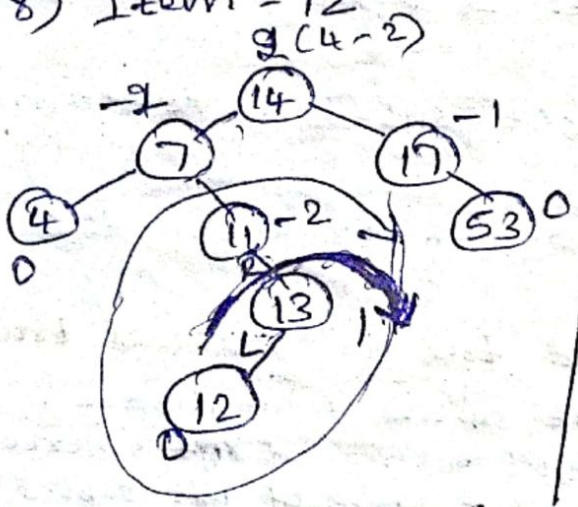
7) Item = 13

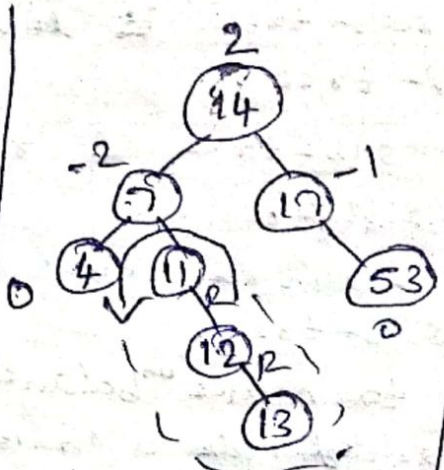


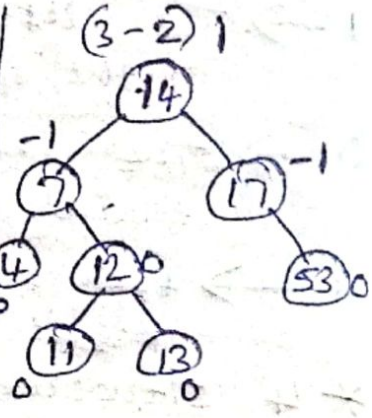
7) Item = 13



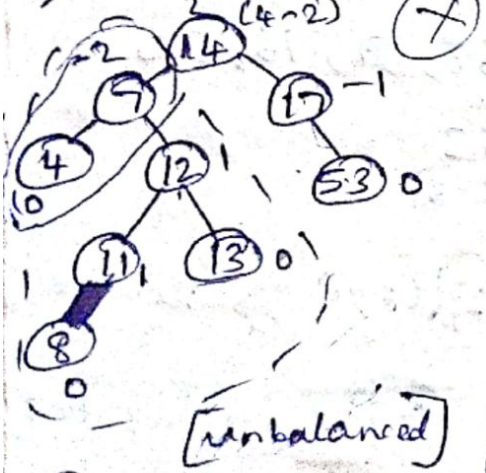
8) Item = 12





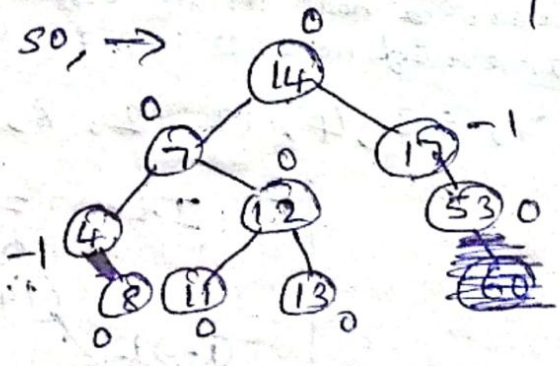


9) Item = 8



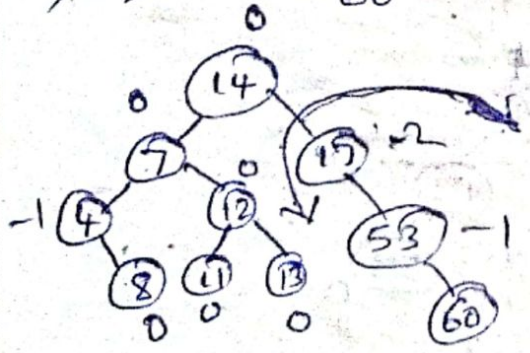
[unbalanced]

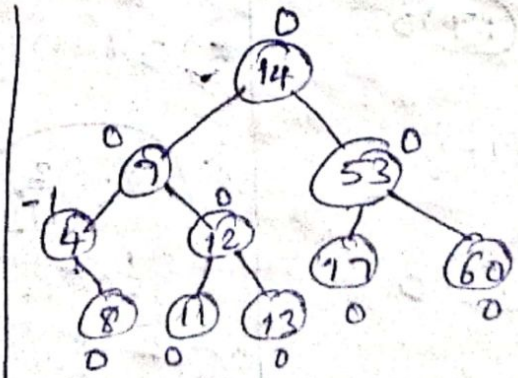
So, →



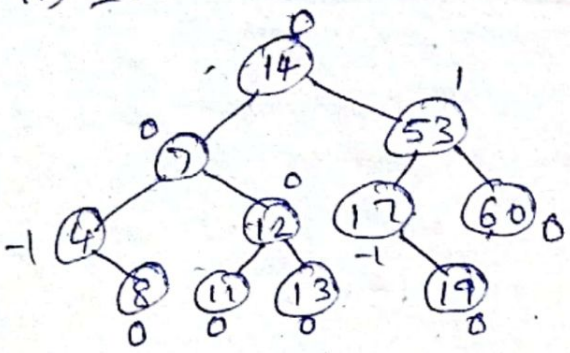
[Balanced]

10) Item = 60

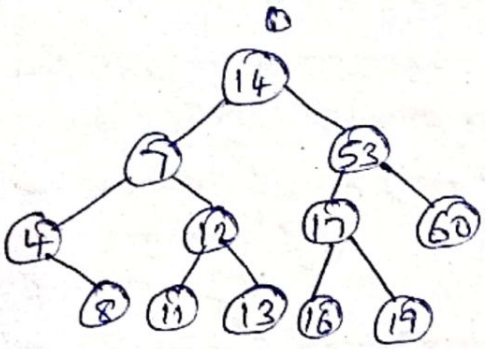




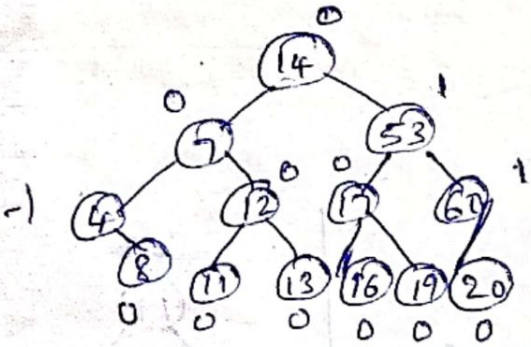
11) Item = 19



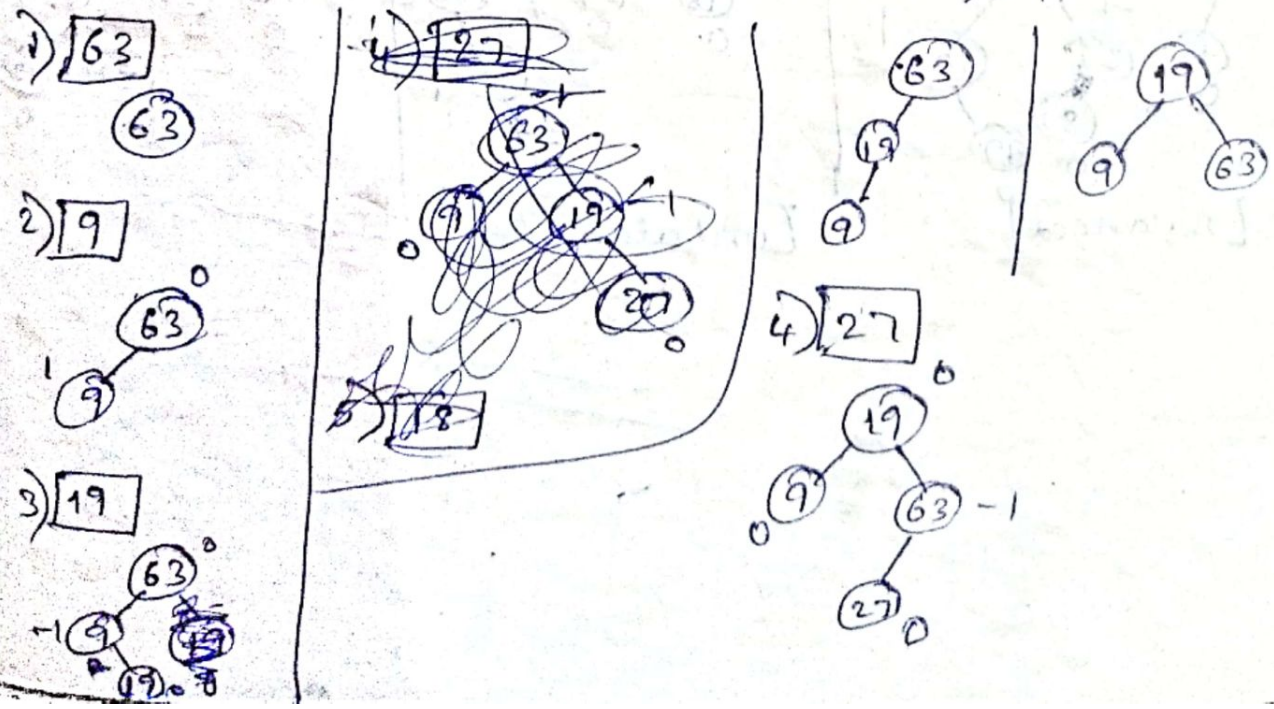
12) Item = 16



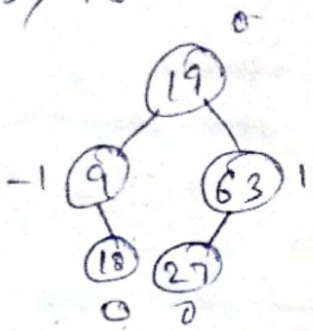
13) Item = 20



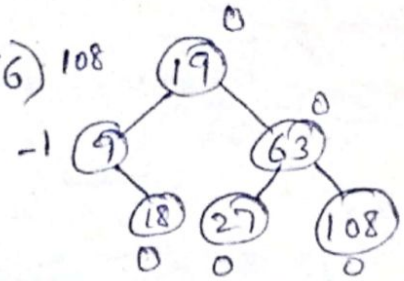
Eg:- 63, 9, 19, 27, 18, 108, 99, 81



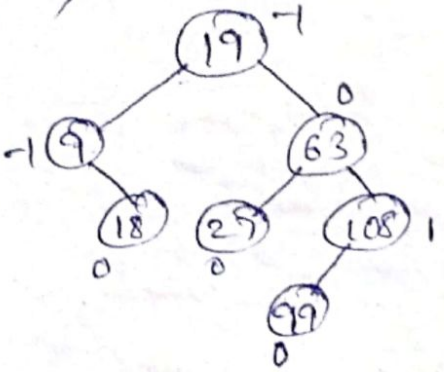
5) 18



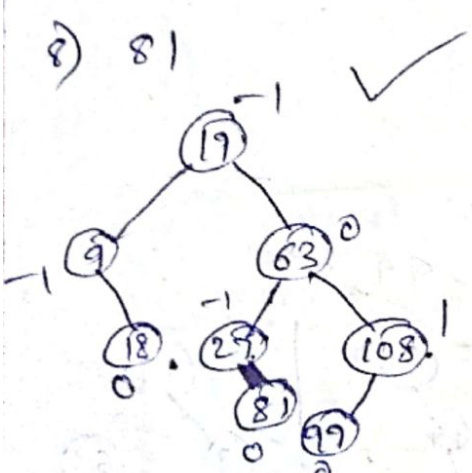
6) 108



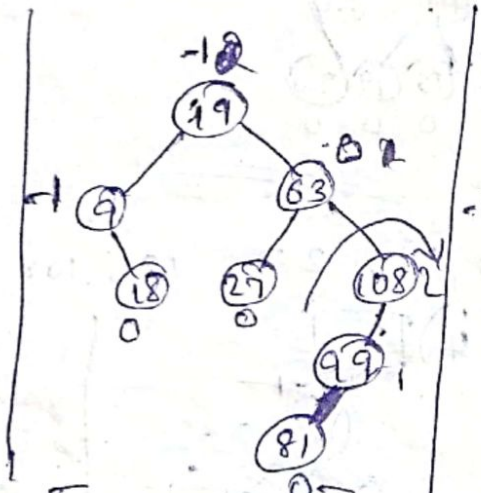
7) 99



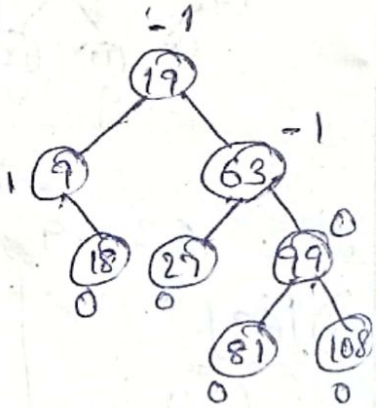
8) 81



[Balanced]



[unbalanced]

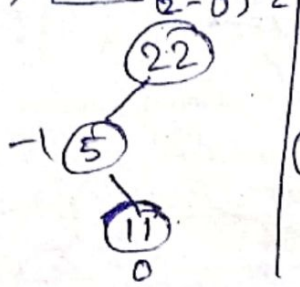


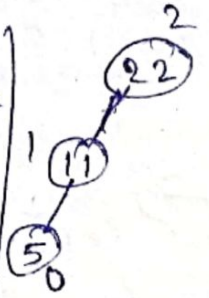
Eg:- 22, 5, 11, 19, 2, 25, 7, 31, 44, 56, 100

1) 22

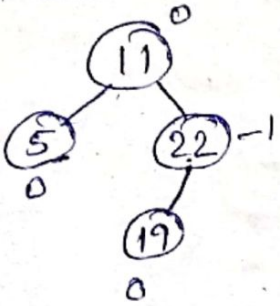
2) 5

3) 11

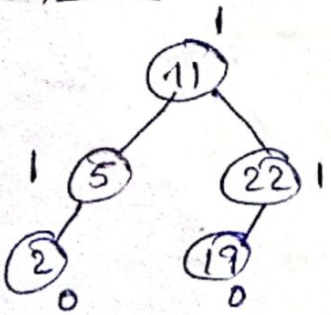




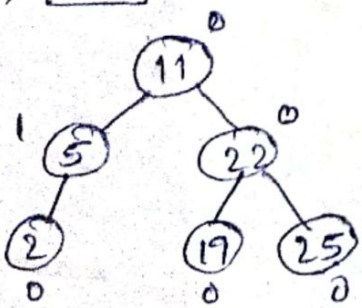
4) 19



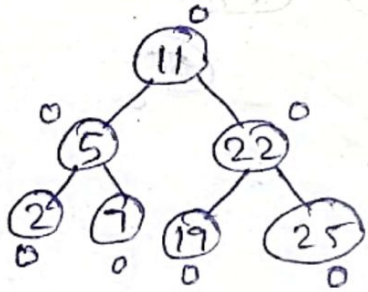
5) 2



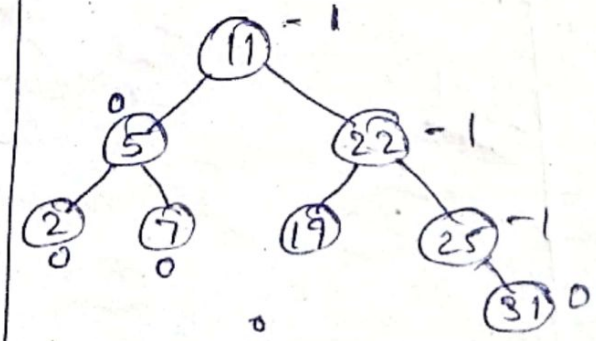
6) 25

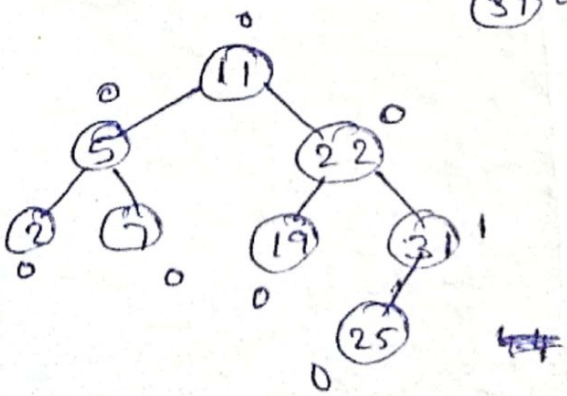


7) 7

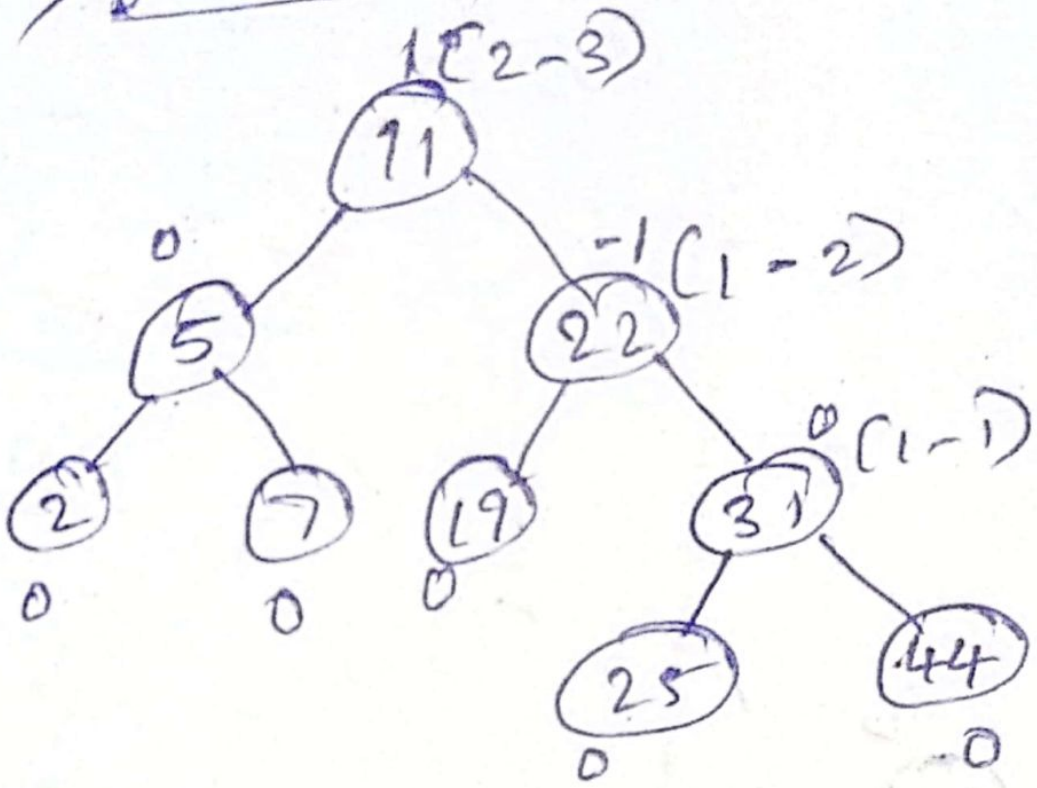


8) 31

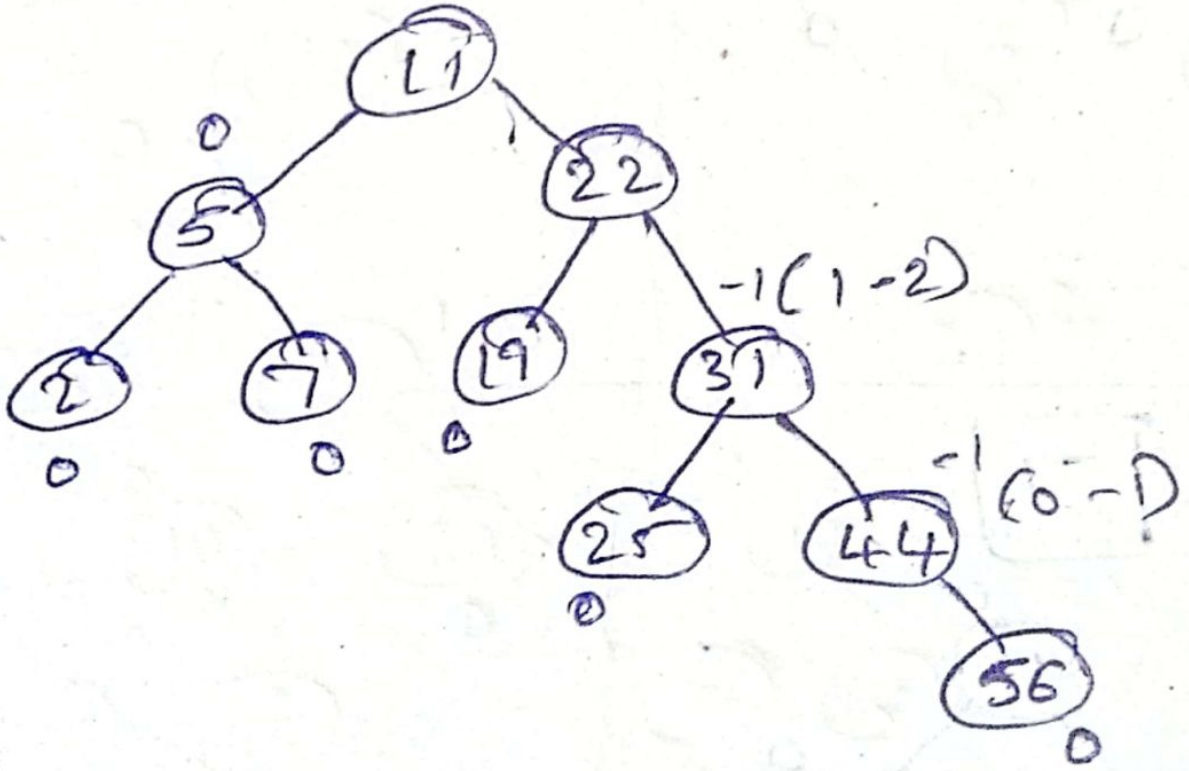




9) 44

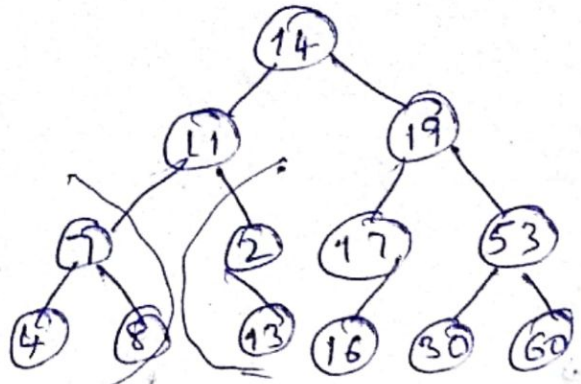


10) 56

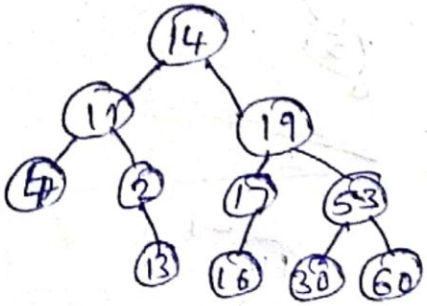


✗ : Deletion in AVL tree :-

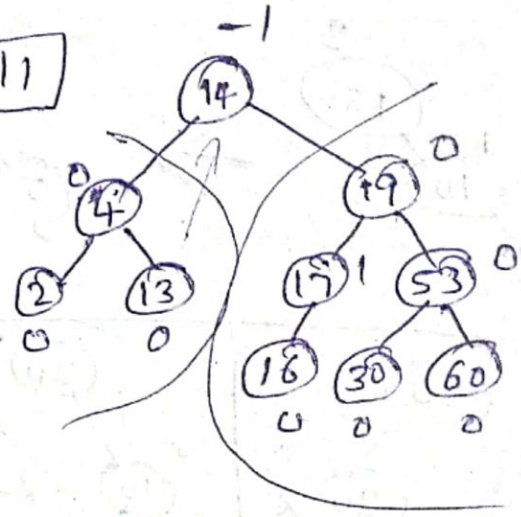
Eg :- 8, 7, 11, 14, 17



1) 8 | 7



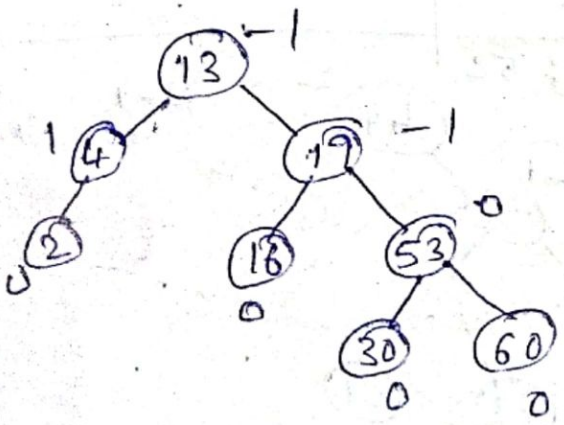
2) 11



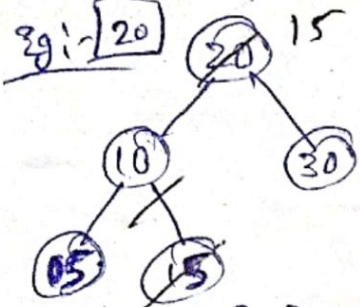
3) Inord traversal : 14 | 17

2, 4, 13, 16, 14, 17, 19, 20, 53, 60

(14 is the root; 2, 4, 13, 16 are its predecessors; 17, 19, 20, 53, 60 are its successors)

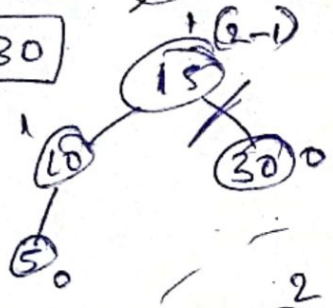


Eg:- 20

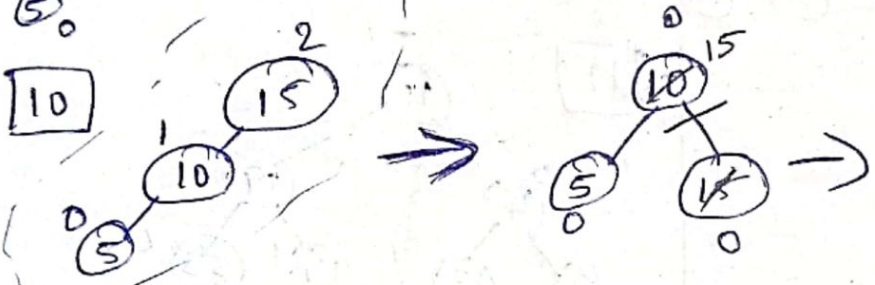


Delete : 20, 30, 10

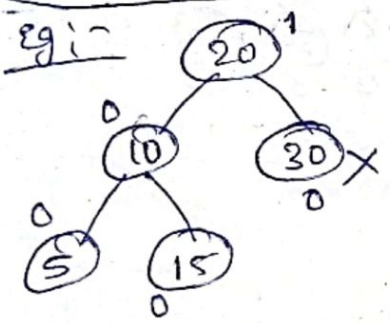
30

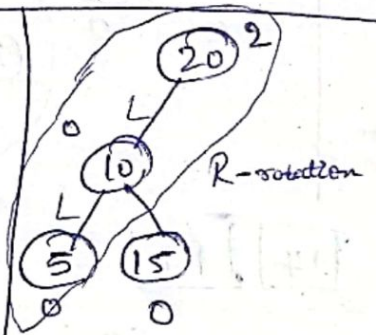


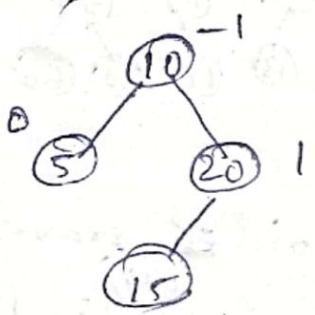
10



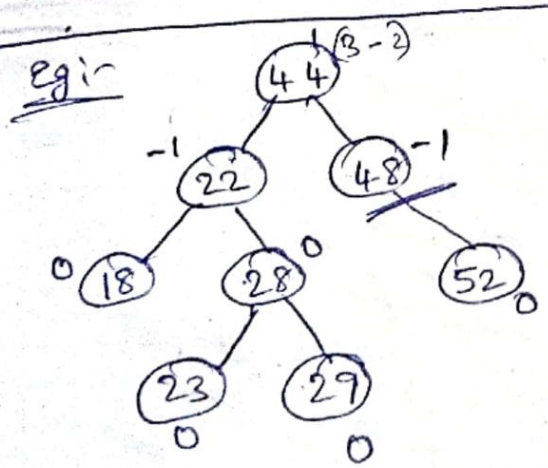
Eg:-



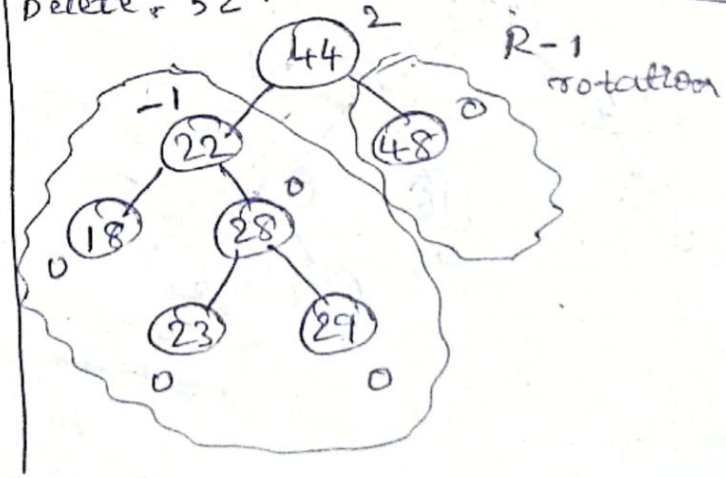


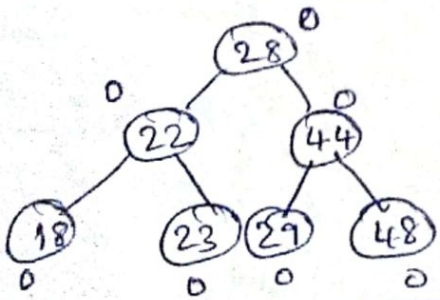


Eg:-



Delete : 52





→ Lexicographic Search Tree Tries :-

Insert: a b c d
a b g l
c d f
a b c d e
l m n

→ Dictionary form

aide
aoed

i, 0

alphabets : 26 - tries

aide →

Insertion into Trie :-

```
TrieNode
{
  MAP < character, TrieNode > children;   (root)
  boolean end of word;
}
```

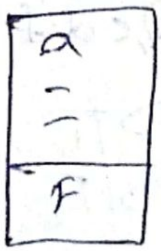
2 ways :-

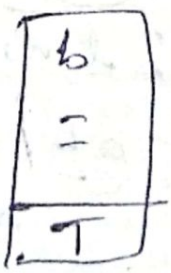
1) Prefix based search
2) whole word search

Delete :-

1) Prefix based search
2) whole word search.

ab →





* Inserting :-

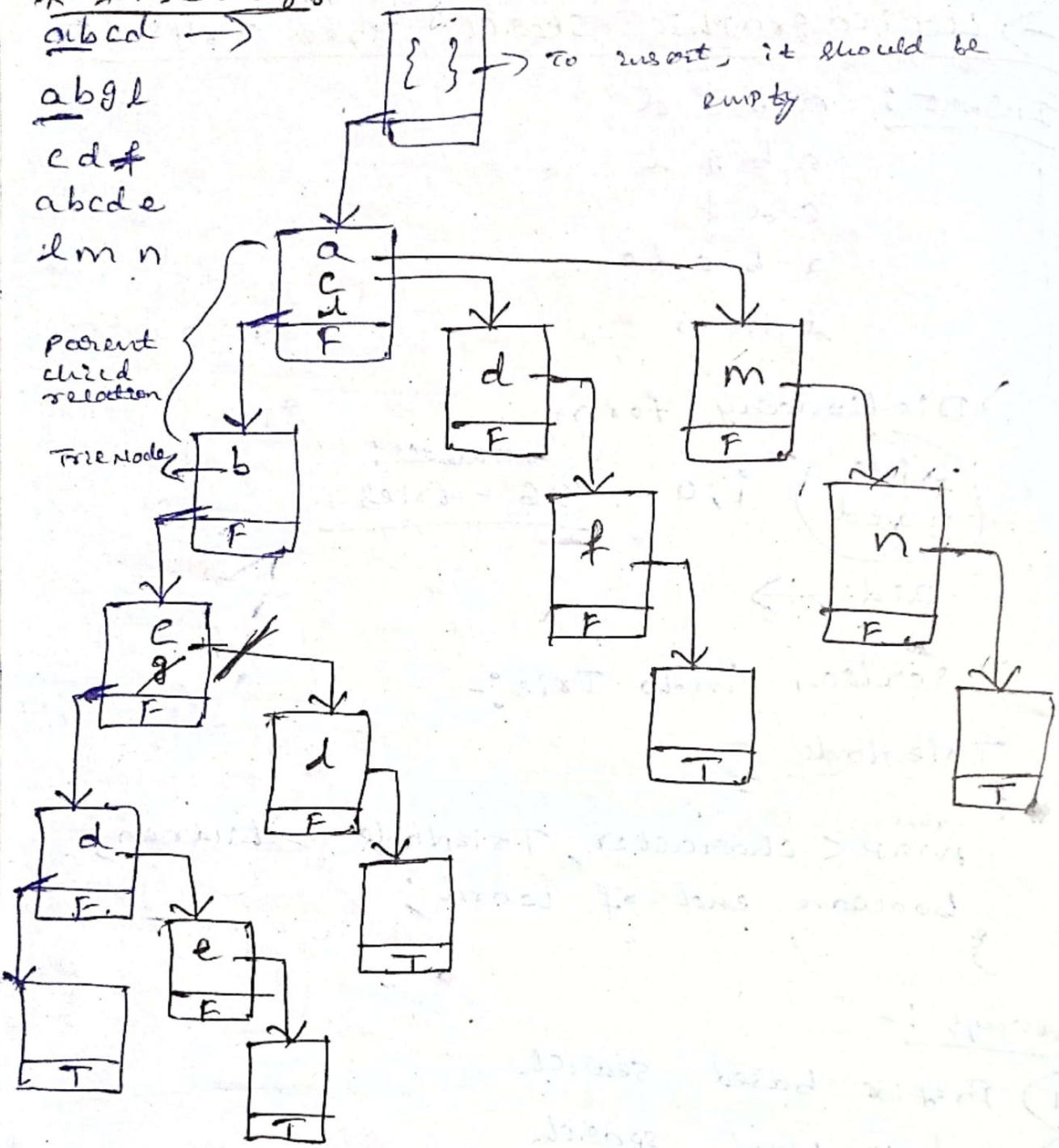
abcd →

abgl

cdef

abcde

lmn



Time complexity :- $O(n^2)$ / $O(n \times n)$

* Searching :-

1) Prefix : ab, (xd) → F

2) whole word : lmn → T, (ab) → F, cdf → T, (ghe) → F

ab / c → T

prefix ← ab

abc → F

* Delete :-

1) Delete entire word.
2) Delete all the word which starts with given prefix.

- a b c → (F)
- a b d l → (T) [End of word should be T]
- a b c d → (F) [End of word is 'F' (false) & it contains reference 'e']

→ B-tree :- [External searching tree]

- Balanced 'm' ~~order~~ way tree.
- Generalization of BST in which a node can have more than one key and more than 2 children.
- Maintains sorted data.
- All leaf node must be at same level.
- B-tree of order 'n' has following properties.
  - Every node has max - 'n' children.
  - Min children :- leaf → 0
    root → 2
    internal nodes → $\left\lceil \frac{n}{2} \right\rceil$ $\frac{3}{2} = (2)$
  - Every node has max (n-1) keys
    min keys :- root node → 1
    All other nodes → $\left\lceil \frac{n}{2} \right\rceil - 1$

Eg :- Insert in B-tree :

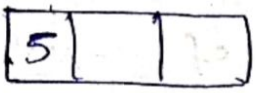
construct a B-tree of order 4 with following set of data;

5, 3, 21, 9, 1, 13, 2, 7, 10, 12, 4, 8

order = 4, keys = (n - 1)

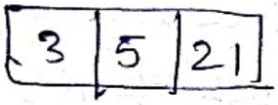
= 4 - 1 = 3

1) (5)

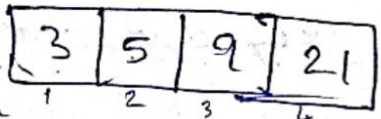


2) (3)

3) (21)



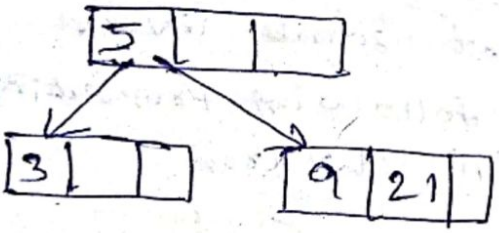
4) (9)



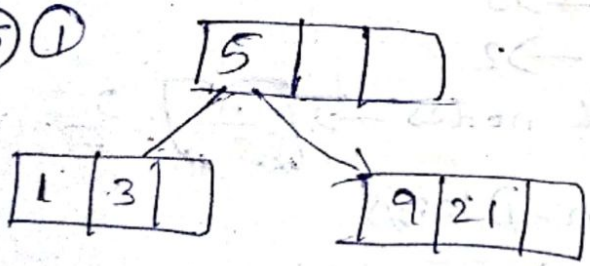
min 1 2 3 4 max

order = 4 , $\frac{4}{2} = 2$

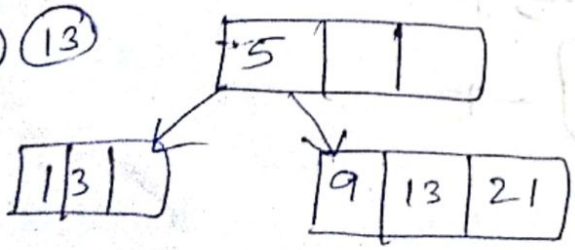
5 is root



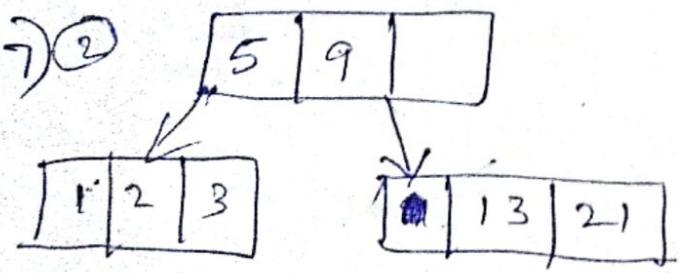
5) (1)



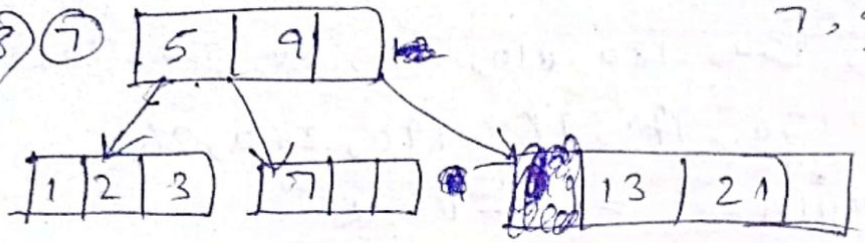
6) (13)



7) (2)



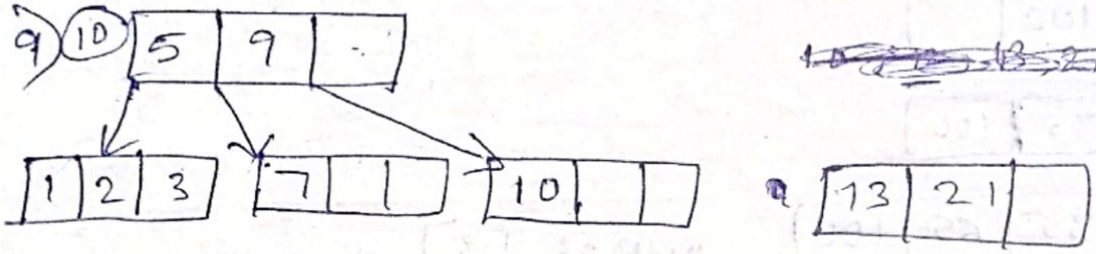
8) (7)



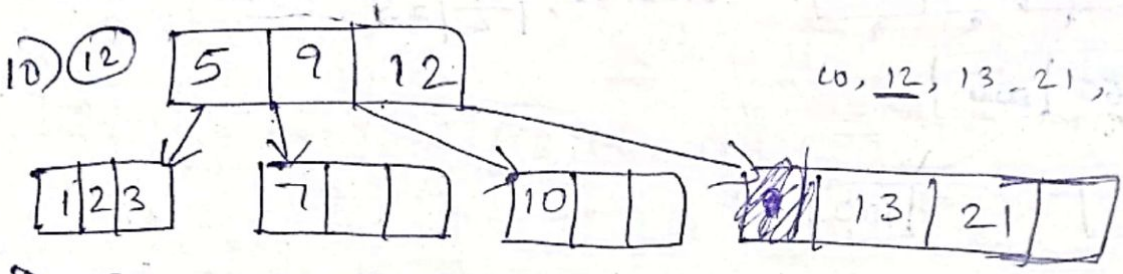
7, 9, 13, 21

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

9) (10)

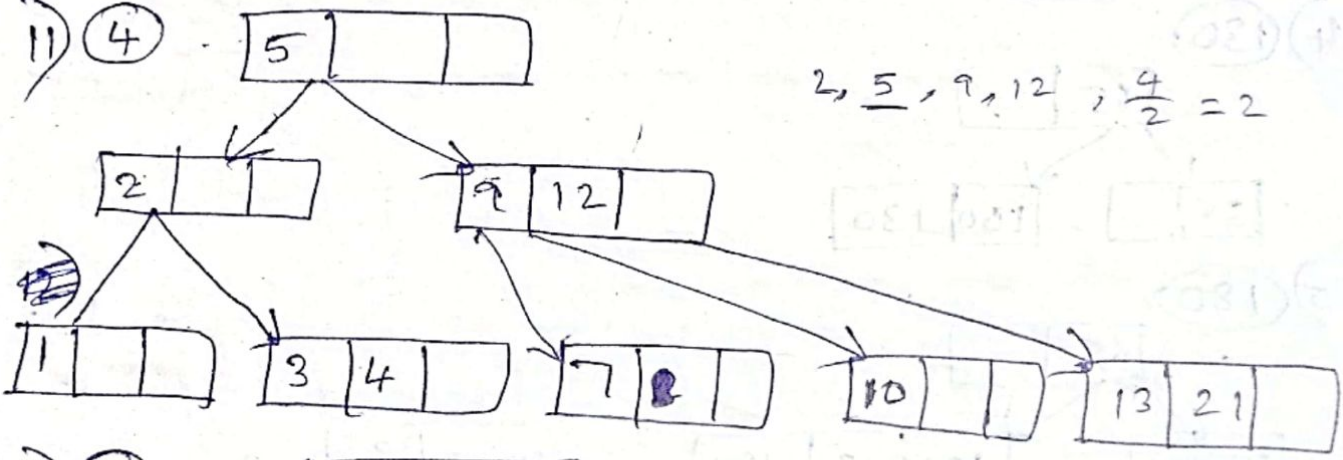


10) (12)



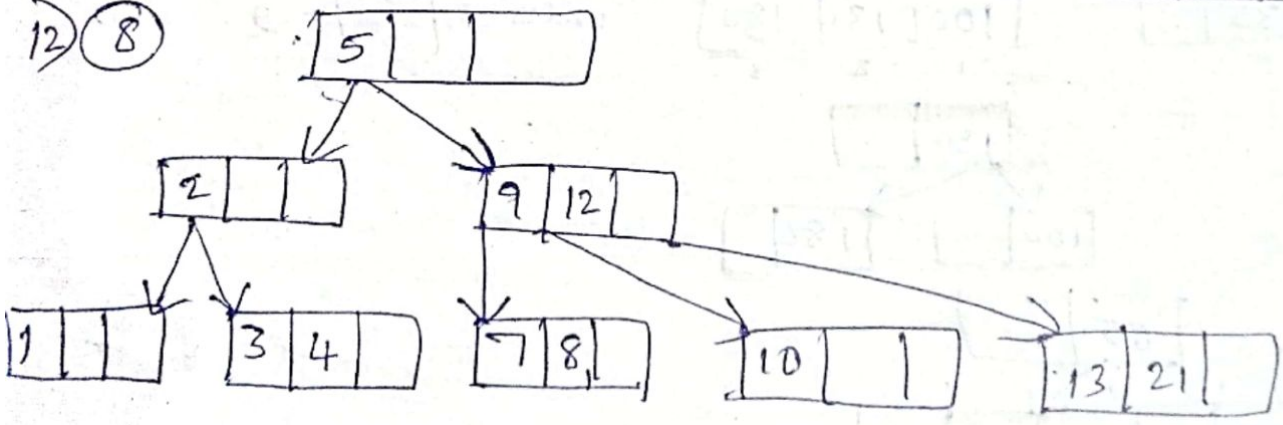
10, 12, 13, 21, $\frac{4}{2} = 2$

11) (4)



2, 5, 9, 12, $\frac{4}{2} = 2$

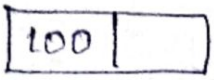
12) (8)



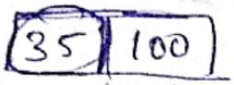
Eg: 100, 35, 65, 130, 180, 10, 20, 40, 50, 70, 80, 90, 110, 120, 140, 160, 190, 240, 260.

order = 3, Key(n-1) = 3 - 1 = 2

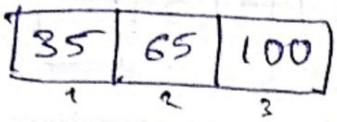
1) 100



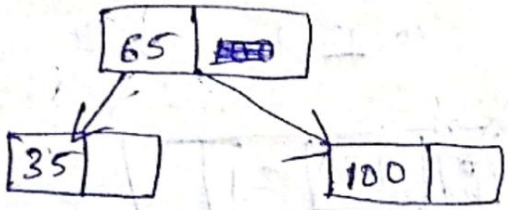
2) 35



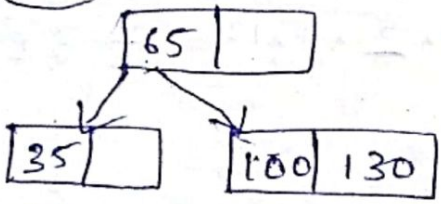
3) 65



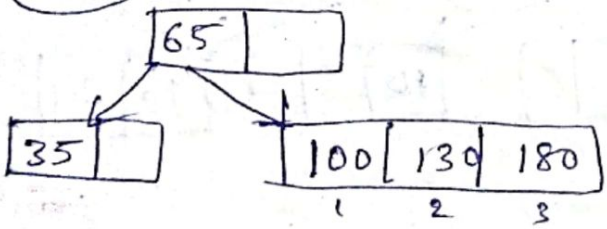
order = 3, $\lceil \frac{3}{2} \rceil = 2$



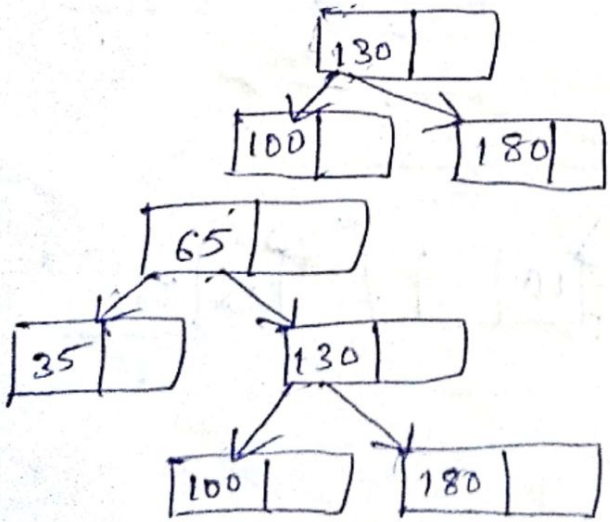
4) 130



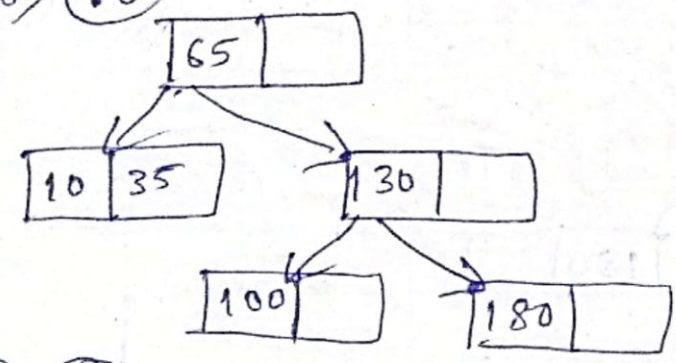
5) 180



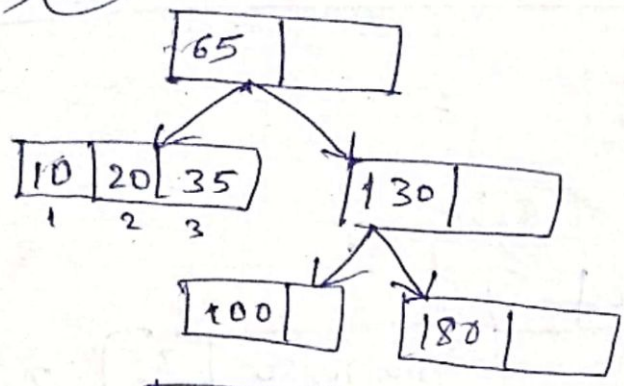
order = 3, $\lceil \frac{3}{2} \rceil = 2$



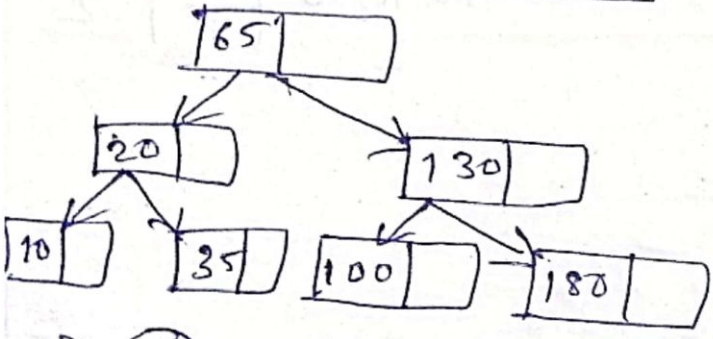
6) (10)



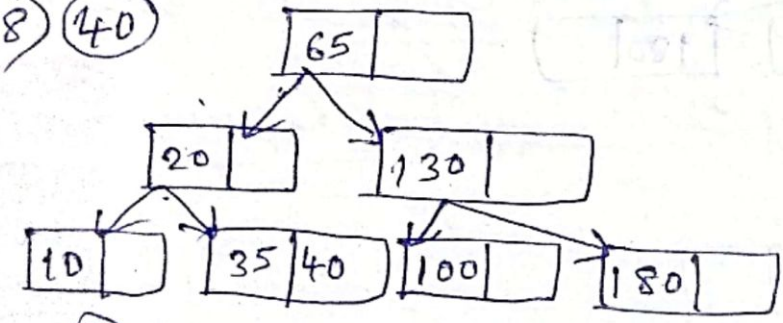
7) (20)



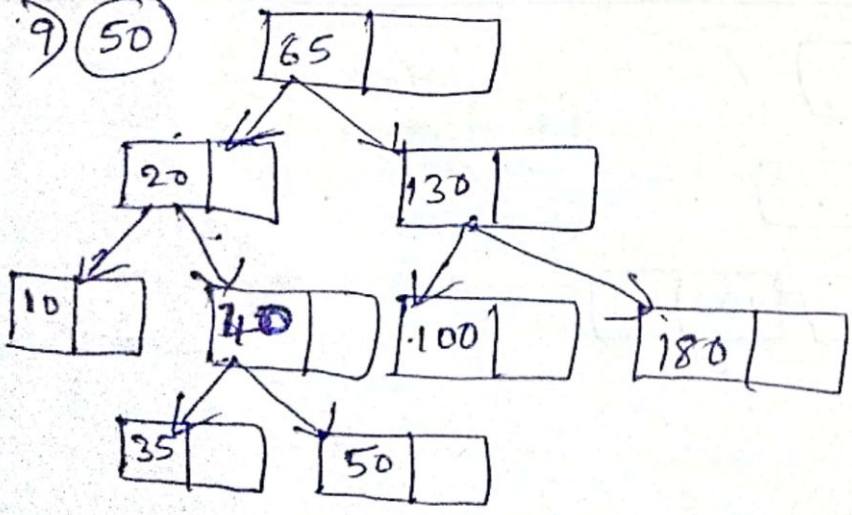
10, 20, 35, $\lceil \frac{3}{2} \rceil = 2$



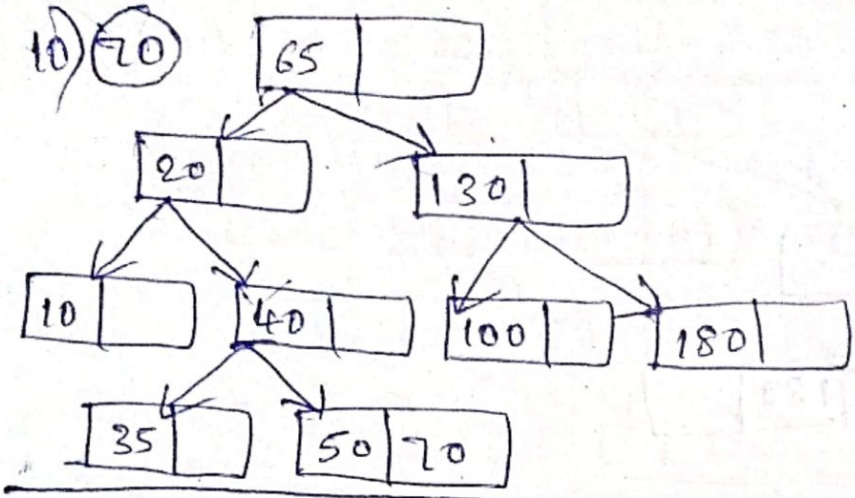
8) (40)



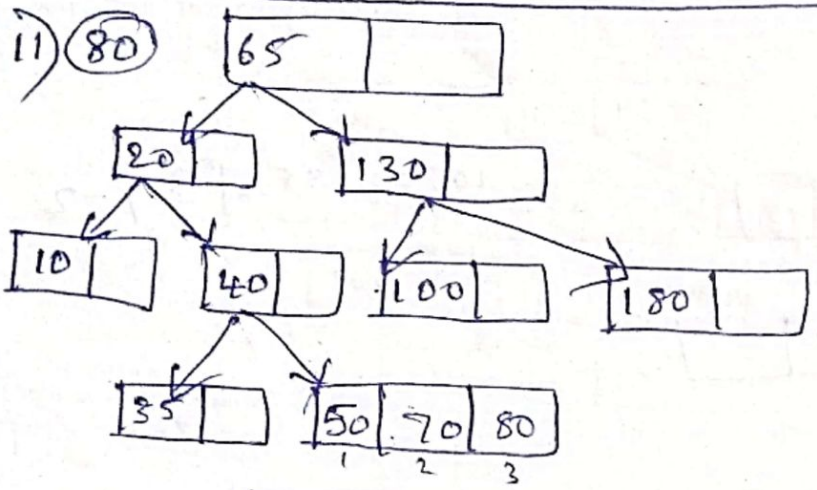
9) (50)



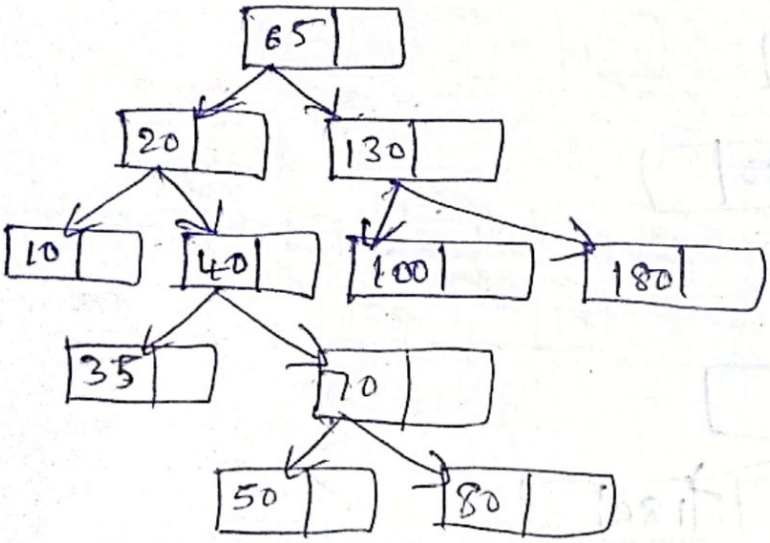
10) 70



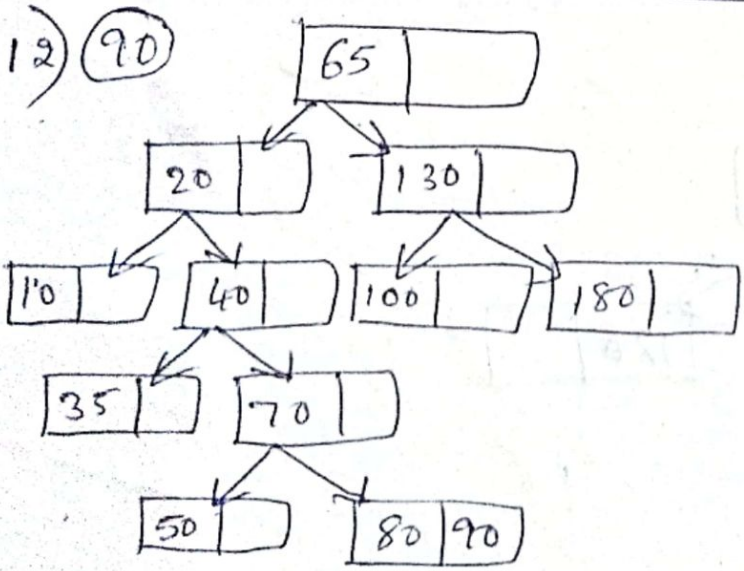
11) 80



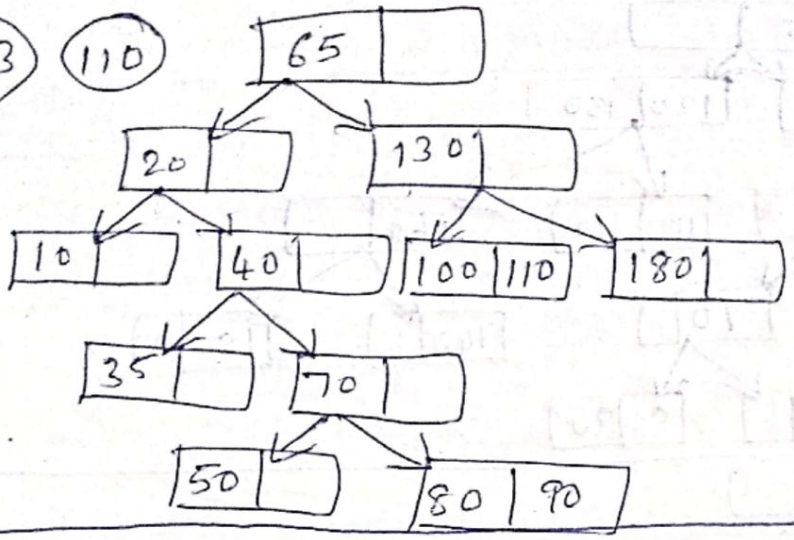
50, 70, 80 $\therefore \left\lceil \frac{3}{2} \right\rceil = 2$



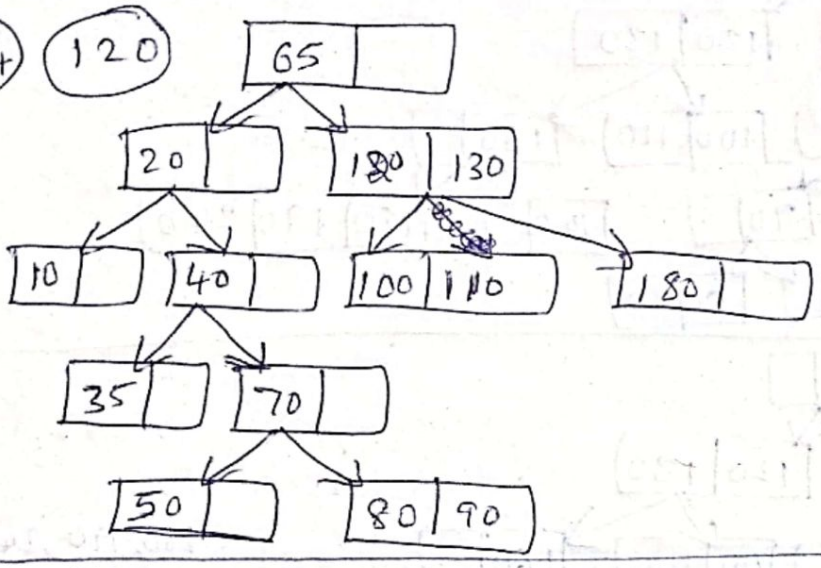
12) 90



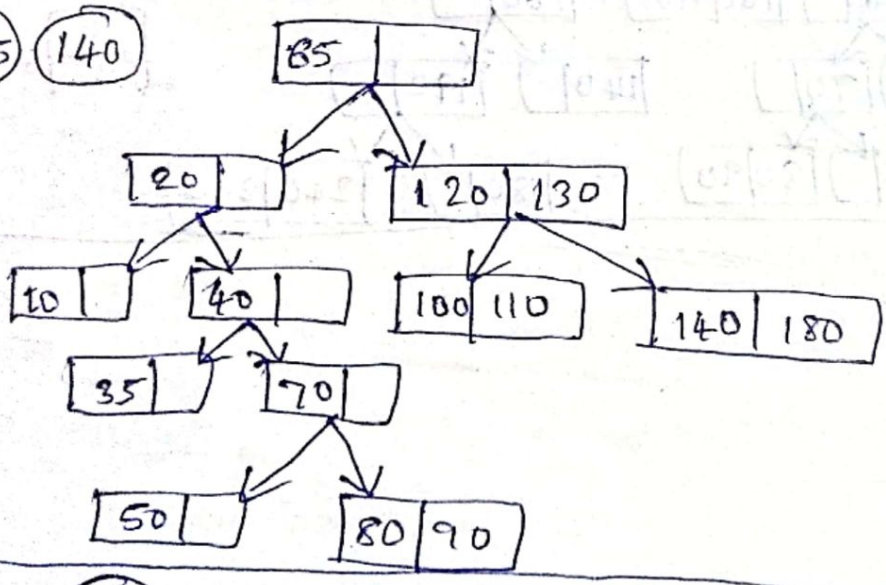
13) 110



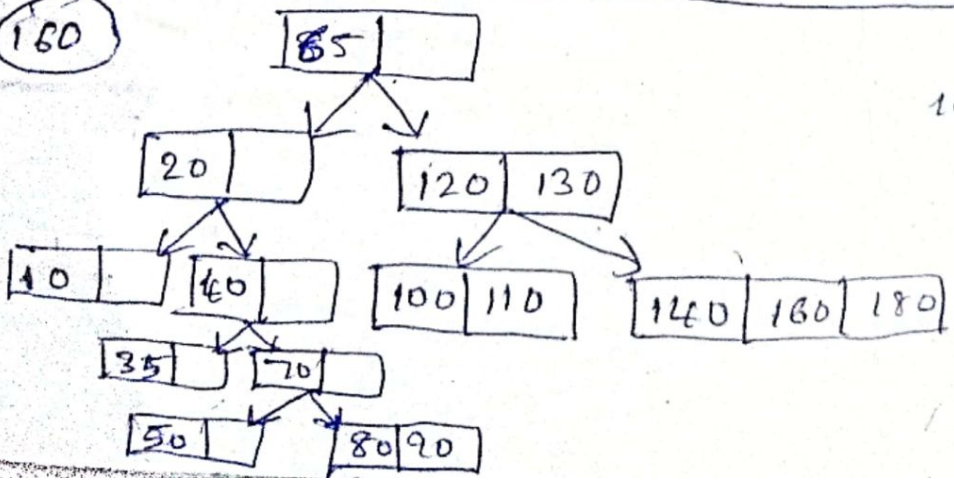
14) 120



15) 140



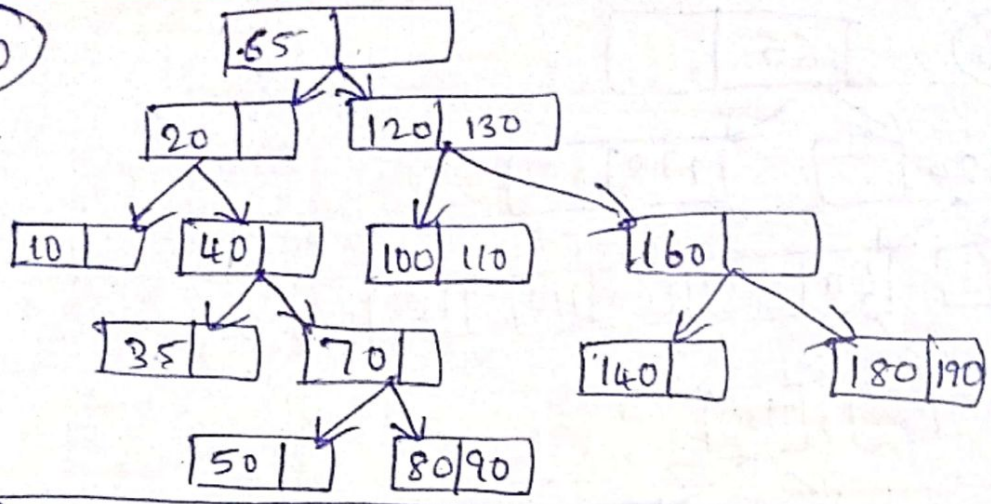
16) 160



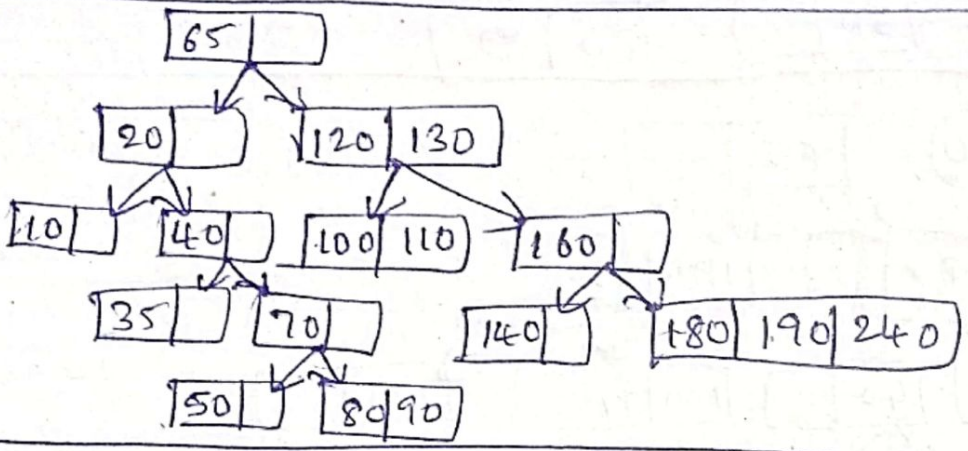
140, 160, 180

$\left\lceil \frac{3}{2} \right\rceil = 2$

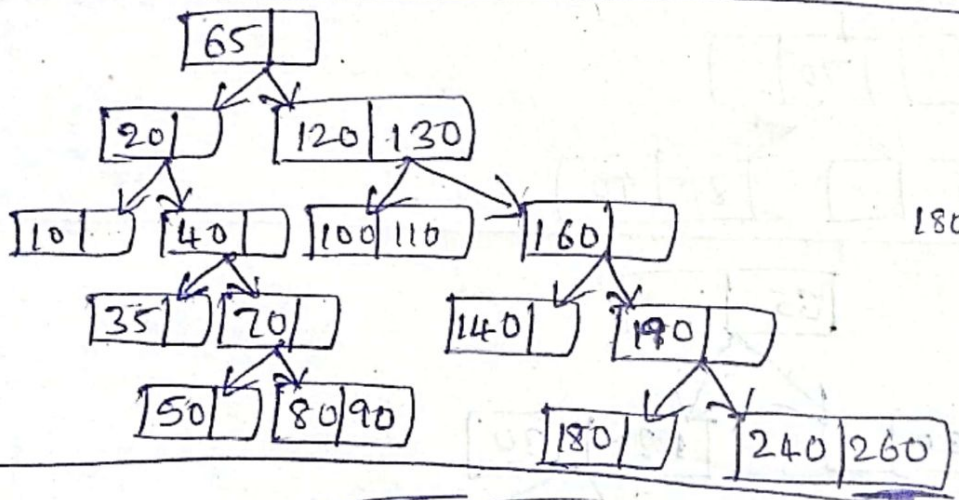
17) (190)



18) (240)



19) (260)



180, 190, 240

$\left\lceil \frac{3}{2} \right\rceil = 2$

↳ Deleting a node in B-Tree :-

2 cases :

1) If target key is in leaf node.
2) If target key is in internal node.

order(m) = 5

min children = $\lceil \frac{m}{2} \rceil = 3$

max children = 5

min keys = $\left(\lceil \frac{m}{2} \rceil - 1 \right) = 3 - 1 = 2$

max keys = 4 $(m - 1 = 5 - 1 = 4)$

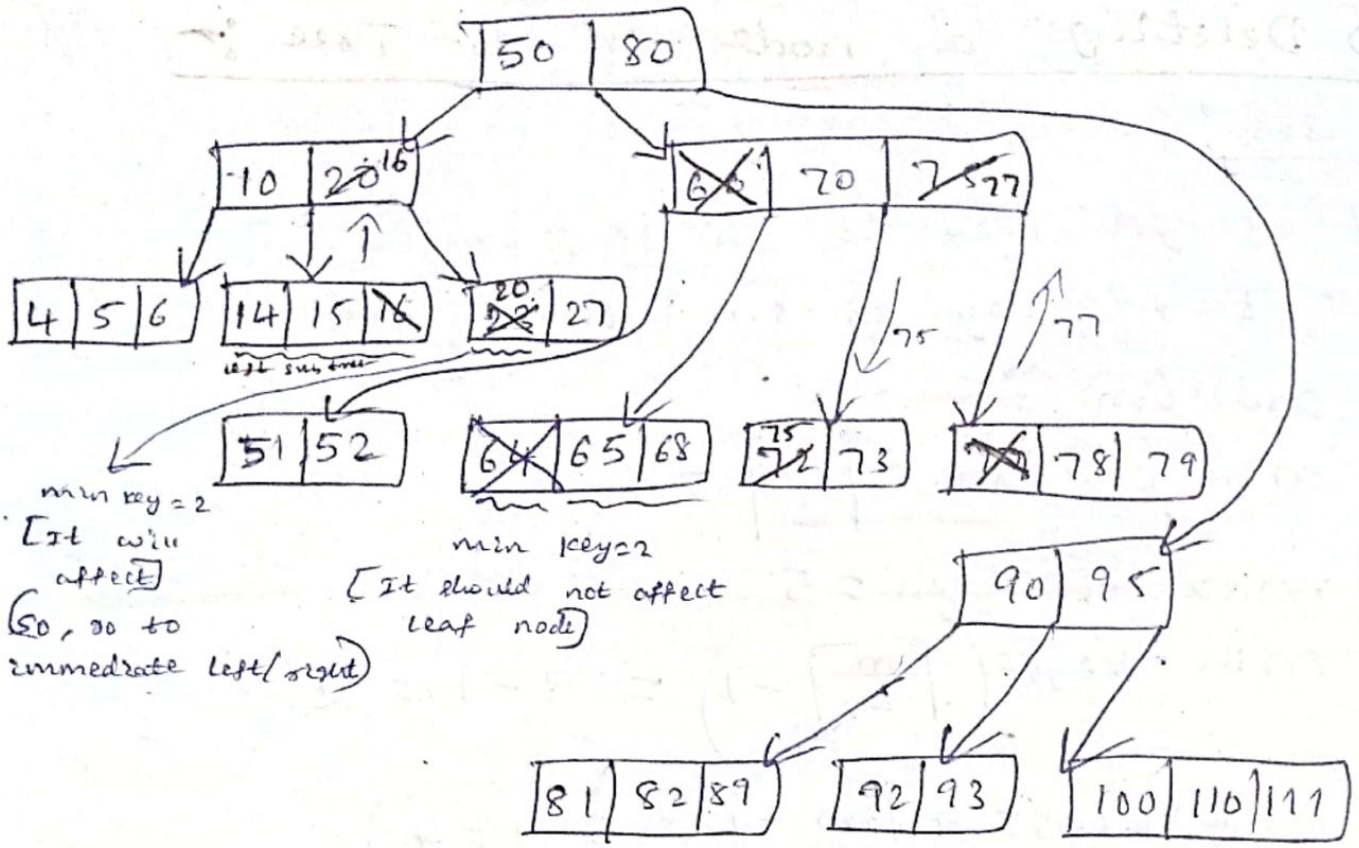
leaf node -

2 conditions :

1) That leaf node contains more Than min no of keys
2) That leaf node contains min no of keys.

↳ 3 keys :

1) borrow key from its immediate left child (Sibling)
2) borrow key from its immediate right child (Siblings)
3) merge left/right node with target node with Parent.



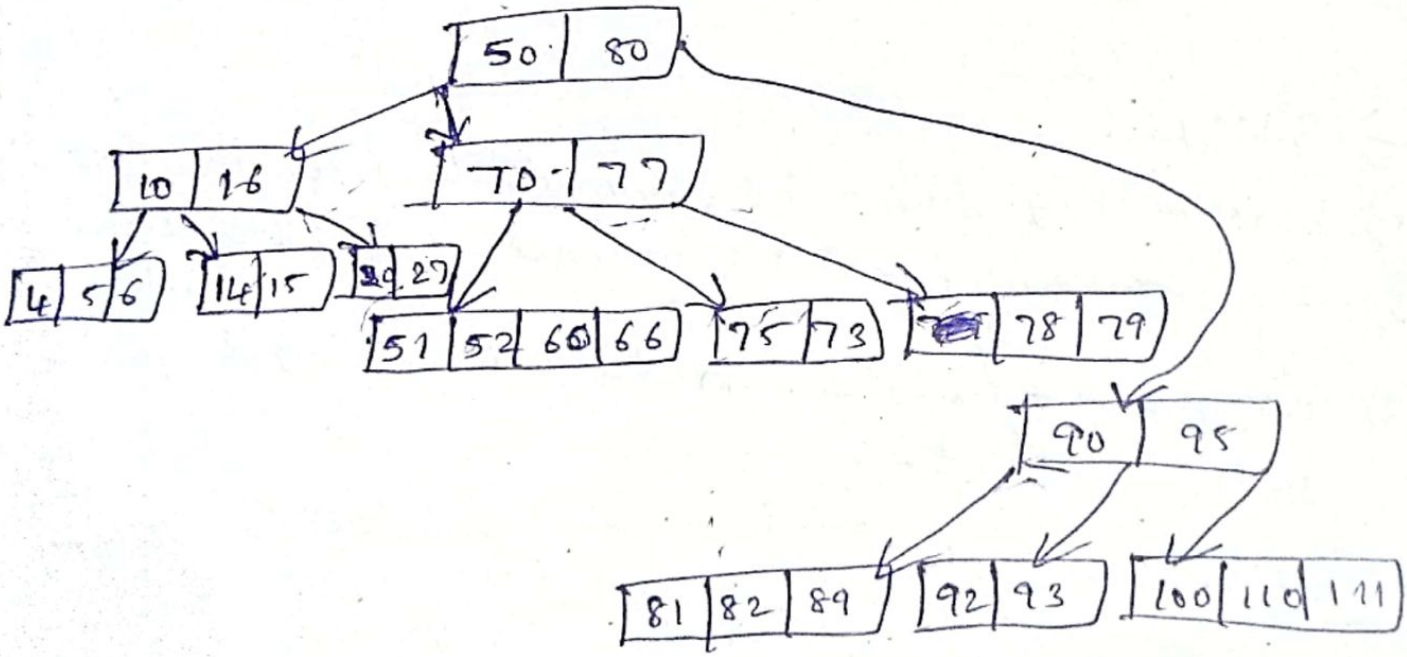
Delete :-

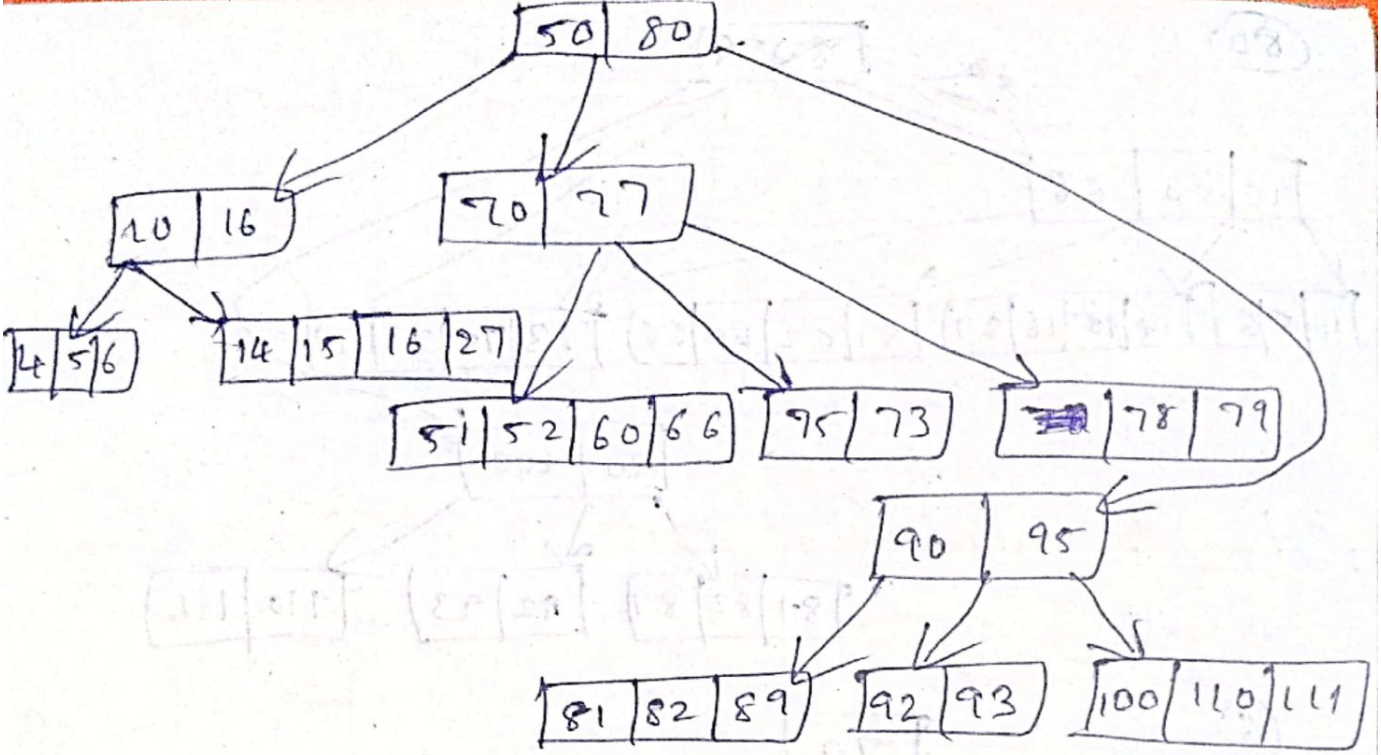
64, 28, 72, 65, 20, 70, 95, 77, 80, 100, 6, 27, 60, 16, 50.

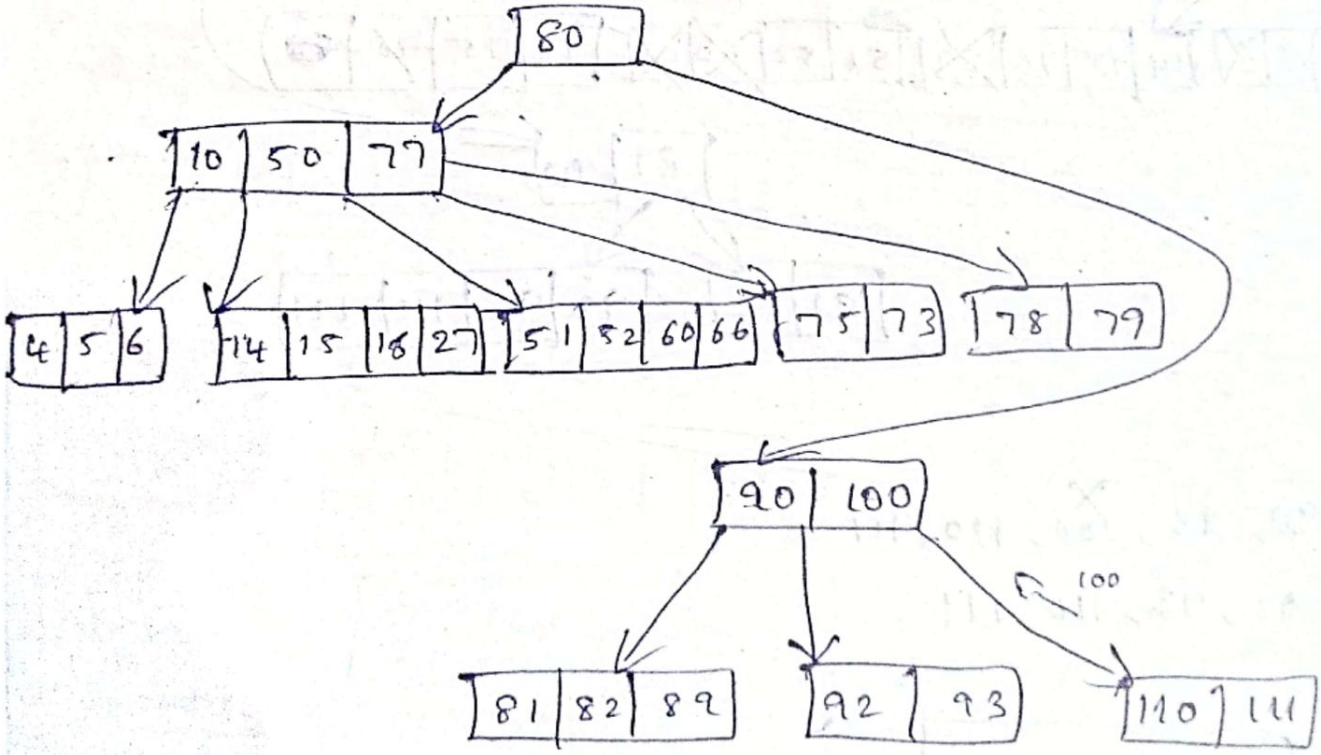
root

⇒ 51, 52, (60) ~~64~~, 68

⇒ [51, 52, 60, 68]



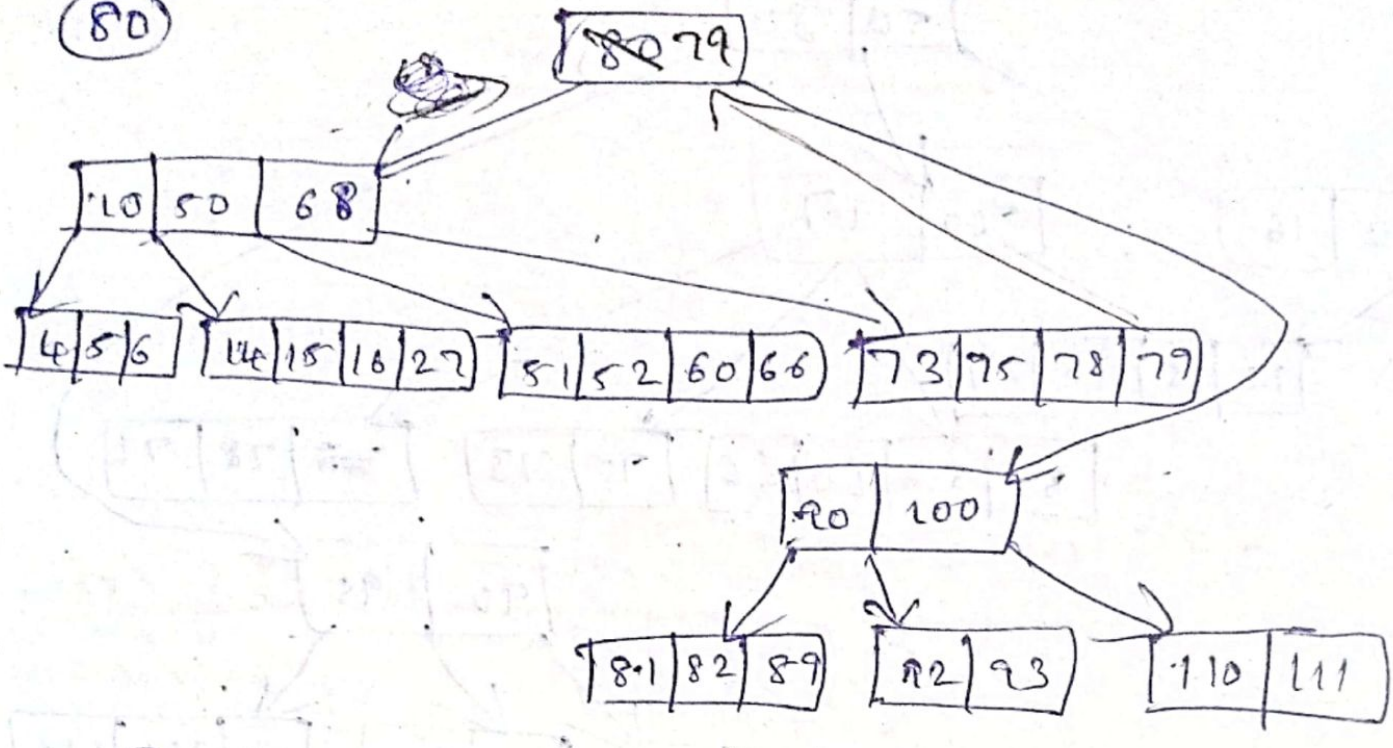




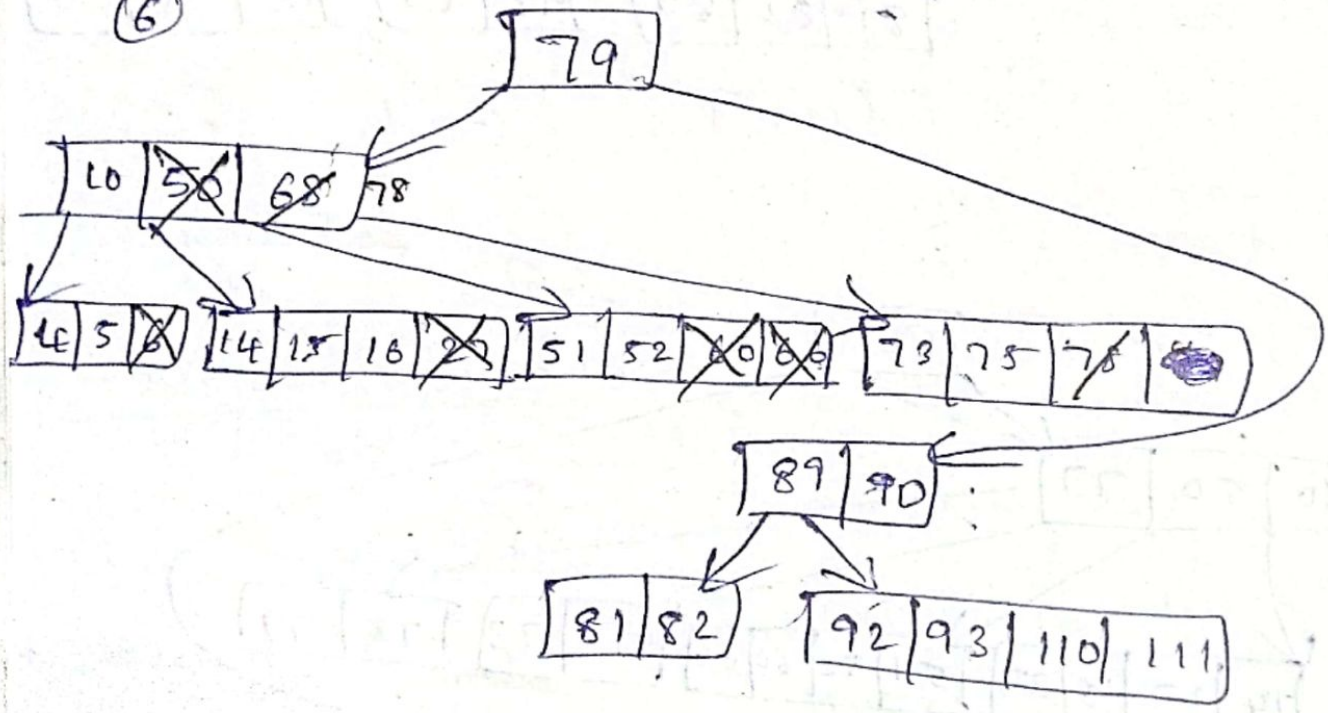
merge : 75, 73, ~~77~~, 78, 79

73, 75, 78, 79

(80)



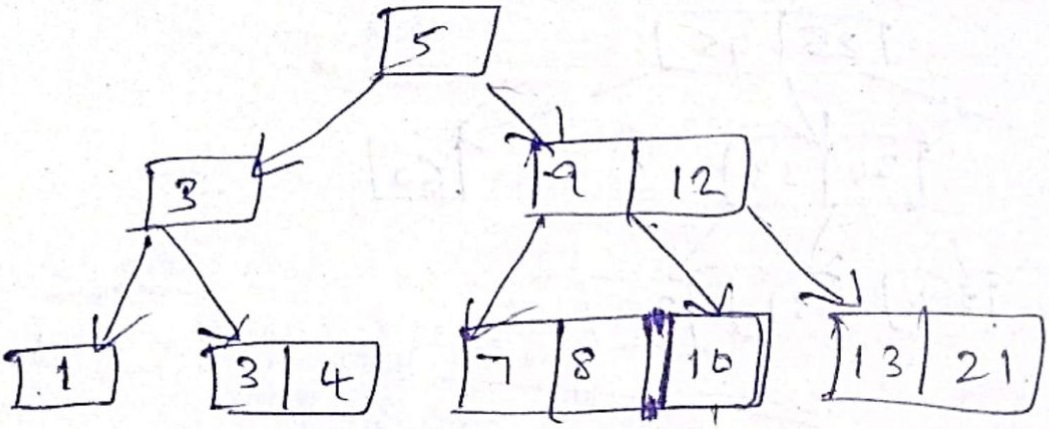
(6)



92, 93, 100, 110, 111

92, 93, 110, 111

10, 78, 89, 90

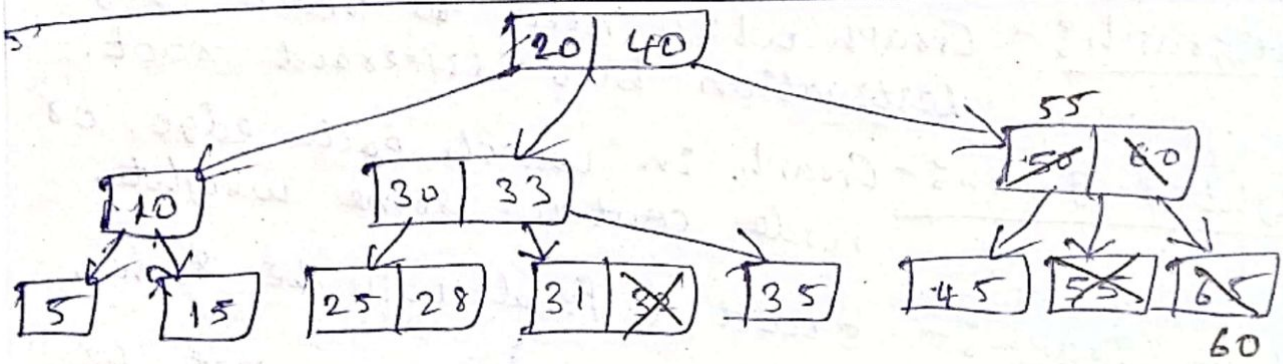


order = 4

$(m - 1) = 3$

min $\left\lceil \frac{m}{2} \right\rceil = \lceil 1.5 \rceil = 2$

min keys $= \left\lceil \frac{m}{2} \right\rceil - 1 = 1$



order = 3

max keys = 2

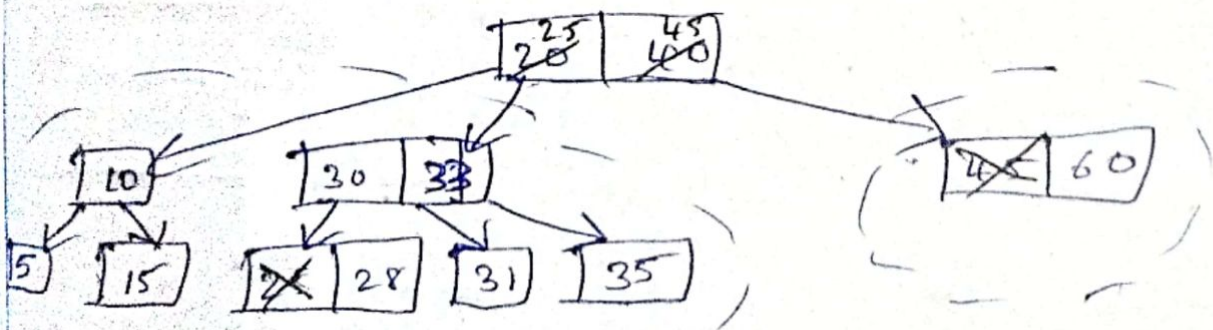
min children $= \left\lceil \frac{m}{2} \right\rceil = \lceil 1.5 \rceil = 2$

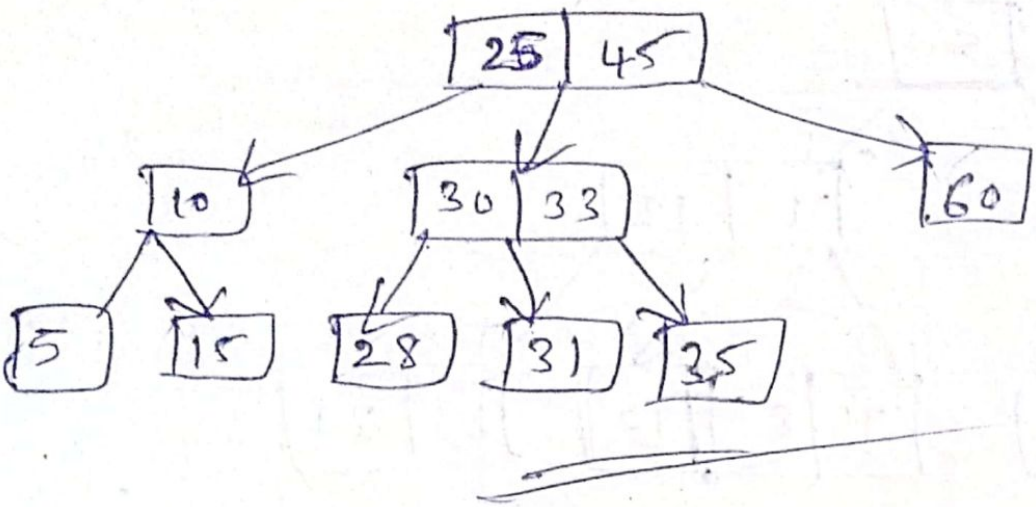
min keys $\left\lceil \frac{m}{2} \right\rceil - 1 = 1$

55 60 65

45 55 60

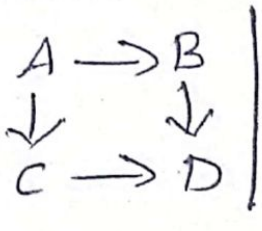
45, 60





Graph:- Collection of edges & vertex.

Degree:-



A (B, C)
B (D)
C (D)

Loop:-

→ B (A)(n)

Multigraph:- Graph which need to reach single destination but different root.

Weighted graph:- Graph in which each edge or node contain some weight.

Cyclic graph:- start & final will be same.

Acyclic graph:- Initial & final will not be the same.

→ Graphs :-

- A graph 'G' consist of 2 things :

i) A set V of ~~of~~ elements called nodes (points / vertices)

ii) A set 'E' of edges such that each edge e in E is identified with a unit ~~(an~~ (unordered pair $[u, v]$ of nodes in V, denoted by $E = [u, v]$),

- Sometimes we indicate the parts of a graph by writing $G = (V, E)$
- Suppose $E = [u, v]$ then the nodes 'u' & 'v' are called the end points of V & 'u' & 'v' are said to be adjacent nodes ~~of~~ neighbour nodes.
- The degree of a node 'u' is written as $\deg(u)$, is the no of edges containing 'u'. If $\deg(u) = 0$ i.e, if 'u' does not belong to any edge then 'u' is called as Isolated node.
- A Path 'P' of length 'n' from a node 'u' to a node 'v' is defined as a sequence of n+1 nodes :

$P = (v_0, v_1, v_2 - - v_n)$

such that $u = v_0$, v_{i-1} is adjacent to v_i for $i = 1, - - n$ ⟶ & $v_n = v$

- A Path 'P' is closed; if $v_0 = v_1$.
- A Path 'P' is said to be simple, if all nodes are distinct with exception that v_0 may $= v_n$ i.e, P is simple, if nodes $v_0, v_1 -- v_{n-1}$ are distinct & nodes $v_1, v_2 -- v_n$ are distinct.
- Cycle :- It is a closed simple path with length 3 or more.
- A cycle of length 'k' is called k-cycle.
- A graph G is said to be connected if there is a path b/w any 2 of its nodes.
- A complete graph with n-nodes will have $\left[\frac{n(n-1)}{2}\right]$ edges.
- A connected graph with (or) without any cycle is tree graph or simply a tree. (or) Pre-tree.
- A graph 'G' is said to be labeled if its edges are assigned data.
- In particular G is said to be weighted if each edge E in 'G' is assigned a non-negative numerical value $w(E)$ called the weight of length of E.

- **Multiple edges :-** distinct edges e & e' are called multiple edges, if it connect the same end points i.e, if $e = [u, v]$ & $e' = [u, v]$

- **Loops :-** An edge E is called a loop if it has identical endpoints, that is, if $e = [u, v]$.

- **multi-graph :-** Multigraph 'M' is said to be finite if it has a finite no of nodes & finite no of edges.

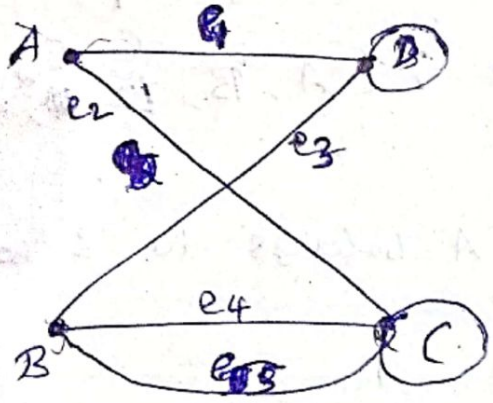


fig (b)
[multigraph]

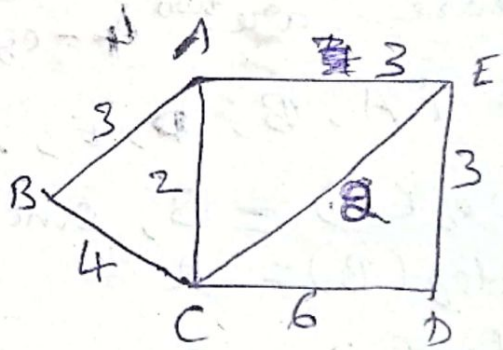


fig (d)
[weighted graph]

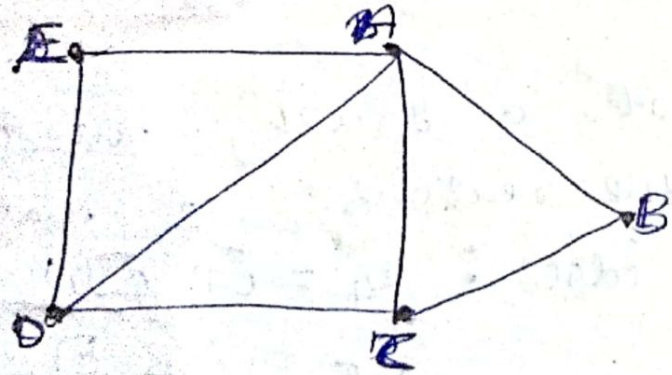


fig-(a)
[graph]

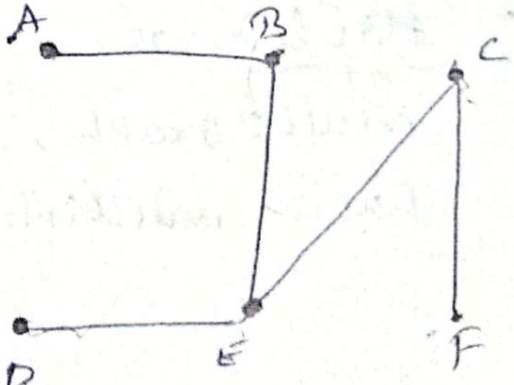


fig (c)
[tree]

- fig(a) is a picture of connected graph with 5 nodes A, B, C, D & E & 7 edges [A, B] [B, C] [C, D] [D, E] [A, E] [C, E] [A, C]. there are 2 simple path of length 2 from B to E [B, D, E] & [B, C, D, E].

There is 1 simple path ~~from~~ of length 2 from B to D [B, C, D]

- we note that B, A, D is not a path since [A, D] is not a edge.

- There are two 4-cycle : [A, B, C, E, A] & [A, B, D, E, A]

- Deg(A) = 3, since A belongs to 3 edges,
  deg(B) = 2
  deg(C) = 3
  deg(D) = 3
  deg(E) = 2.

- fig(b) is not a graph but a multigraph, the reasons is that it has multiple edges : $e_4 = [B, C]$, $e_5 = [B, C]$ & it has a loop $e_6 = [D, D]$, the definition of a graph usually does not allow either multiple edges — or — multiple loops.

- fig(c) is a tree-graph with:
  $m = 6$ nodes & consequently edges $= (m-1) = 5$.

- fig(d) is the graph except the graph is weighted.
- consider $P_1 = [B, C, D]$
  $P_2 = [B, A, E, D]$
  are both Paths from node B to D. although P_2 contains more edges than P_1, the weight $w(P_2) = 9$. is less than weight P_1. $w(P_1) = 10$.

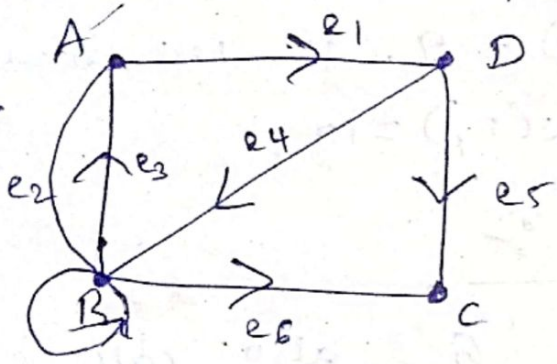
↳ **Directed graph :-**

- A directed graph 'G' also called a di-graph or graph is same as a multi-graph except that each edge 'e' in 'G' is assigned a direction or in other word, each edge 'e' is identified with an ordered pair, $[u, v]$ of nodes in 'G' rather than an unordered pair $[u, v]$.

- The terminologies are used in directed graph:
  e - begins at 'u' & ends at 'v'
  u - is the origin or initial Point of V.
  v - is the destination or terminal Point of 'e'
  u - is a Predesesor of 'v' & 'v' is a successor (neighbour) of 'u'.

- u is adjacent to 'v' & 'v' is adjacent to 'u'.
- The outdegree of a node 'u' in 'G' written as: outdeg(u) is the no of edges begining at u similarly indeg~~(u)~~ of 'u' is writen as: indeg(u) is no of edges end at 'u'.
- A node 'u' is called a ~~node~~ source, if it has a positive outdegree but zero indegree.

Eg:-



↳ Sequential representation of Graphs: Adjacent matrix ; Sparse matrix:-

- There are 2 standard ways of maintaining a graph 'G' in a memory of computer, one way called the sequential representation of graph 'G' is by means of its adjacency matrix 'A'.

A =

	A	B	C	D
A	0	0	0	1
B	2	1	1	0
C	0	0	0	0
D	0	1	1	0

4×4

Adjacency matrix

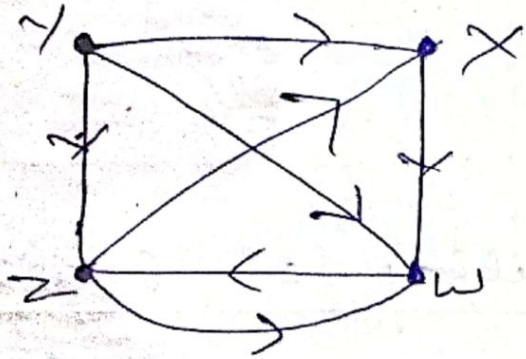
- The other way called the linked list is by the means of Linked list of neighbours.

Adjacency matrix:- Suppose 'G' is a simple directed graph with m-nodes, suppose the nodes of G have been ordered & are called $(v_1, v_2 - v_m)$ then the adjacency matrix $A = (a_{ij})$ of graph of G is the m x m matrix defined as follows:

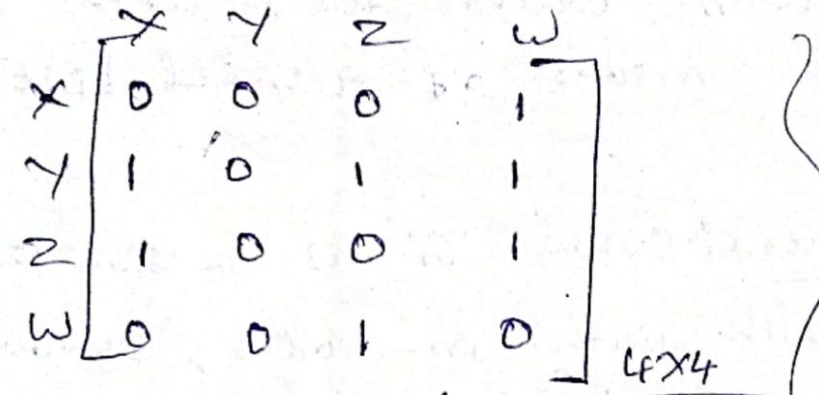
$$a_{ij} = \begin{cases} 1 & \text{if } v_i \text{ is adjacent to } v_j, \text{ there is an edge } [v_i, v_j] \\ 0 & \text{otherwise.} \end{cases}$$

Such a matrix 'A' which contains entries of only 0 & 1 is called a bit matrix or a boolean matrix.

Eg:-



- Suppose the nodes are stored in memory in a linear array DATA as follows:
  DATA : X Y Z W.
- then we assume that ordering of 'G' is as follows,
  $v_1 = X$, $v_2 = Y$, $v_3 = Z$, $v_4 = W$.
- Adjacency matrix of A[;] is as follows:



~~↳ Linked representation of graphs~~

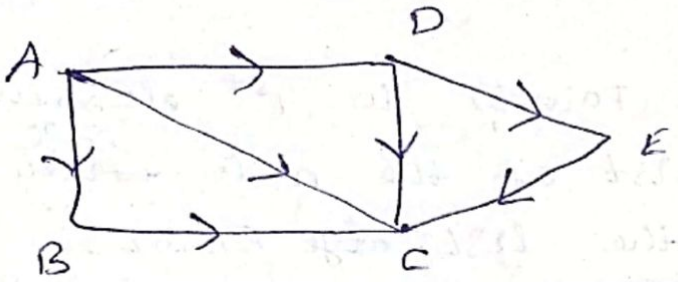
↳ Linked representation of graphs:-

- Let 'G' be a directed graph with 'N' nodes. The sequential representation of G in memory i.e, representation of G by its adjacency matrix. A - has a no of major drawbacks.

- First of all:
it may be difficult to insert & delete in G, because the size of a may need to be changed & the nodes may need to be reorder

so there is a many changes in matrice 'A'.

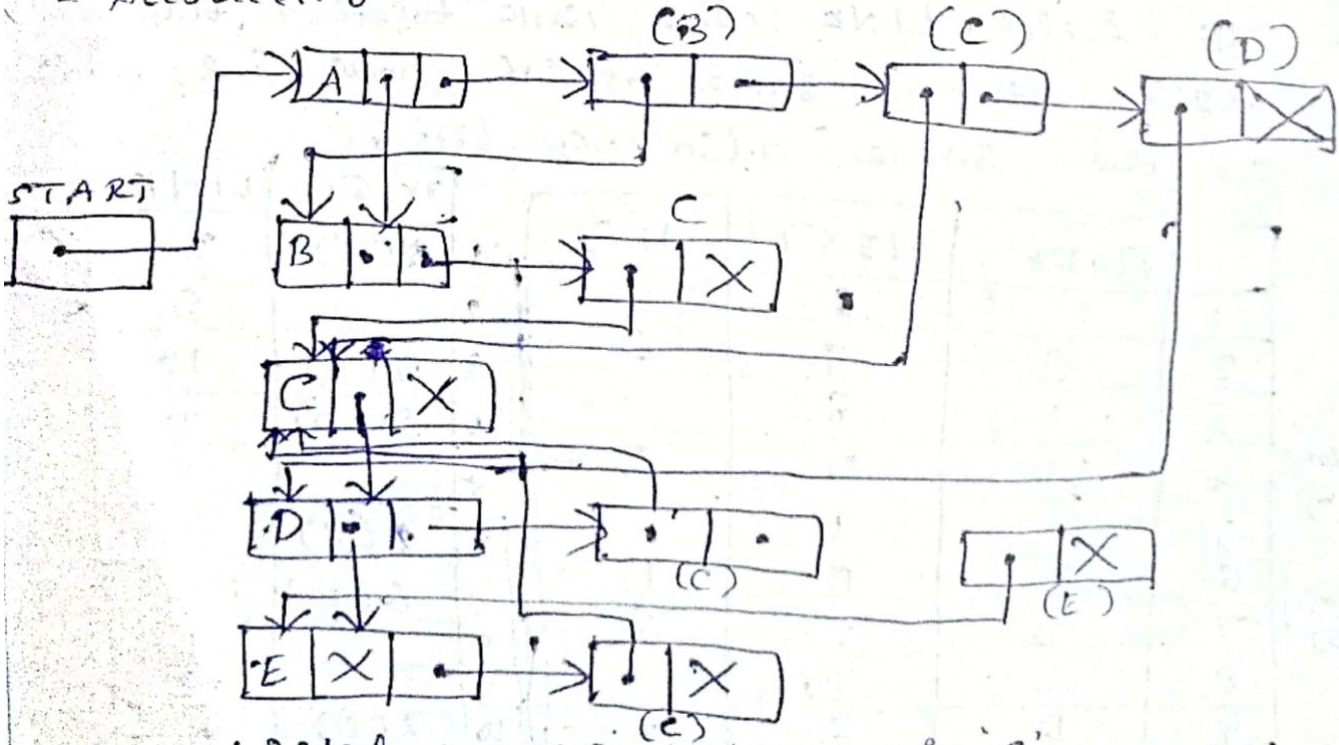
- G is usually represented in memory by a linked representation as a called as an adjacency structure.
- consider the graph as shown below: table shows each node followed by its adjacency list which is its list of adj nodes also called its successors or neighbours



node	Adjacency list
A	B, C, D
B	C
C	—
D	C, E
E	C

	A	B	C	D	E
A	0	1	1	1	0
B	0	0	1	0	0
C	0	0	0	0	0
D	0	0	1	0	1
E	0	0	1	0	0

- Adjacency list of 'G':-



Listed representation of 'G'.

↳ Node list :- Each element in list will correspond to a node in G & it will be record of the form:

NODE	NEXT	ADJ

- Here NODE will be the name / key value of node,
- NEXT will be a pointer to next node in the list.
- ADJ will be a pointer to 1st element in adjacency list of the node which is maintained in the list EDGE.

↳ Edge list :- Each element in list will correspond to an edge of 'G' & will be record of the form.

DEST	LINK

- The field DEST will point to location of destination or terminal node of the edge,
- The field LINK will link together the edges with a same initial node i.e, node in the same adjacency list.

	NODE	NEXT	ADJ
1		3	
2	C	9	0
3		8	
4	A	7	3
5		1	
6	E	0	11
7	B	2	6
8		10	
9	D	6	1
10		0	

Start = 4
Avail = 5

	DEST	LINK
1	2(C)	7
2		5
3	7(B)	10
4	9(D)	0
5		8
6	2(C)	0
7	6(E)	0
8		9
9		12
10	2(C)	4
11	2(C)	0
12		6

AVAILN = 2

↳ Warshall's algorithm:- used to find path matrix

- A directed graph 'G' with 'M' nodes is maintained in memory by its adjacency matrix 'A', this algorithm finds (boolean) path matrix 'P' of graph 'G'.

1) Repeat for I, J = 1, 2, -- M [initialises P]
   if A[I, J] = 0, then, set P[I, J] = 0
   ELSE
   set P[I, J] = 1
   [End of loop]

Path matrix holding a value = 0.

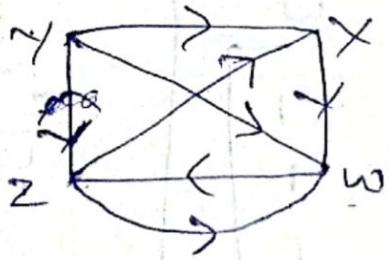
2) Repeat step-3 & 4 for K = 1, 2, -- M
   [Update P]
3) Repeat step-4 for I = 1, 2, -- M
4) Repeat for J = 1, 2, -- M:
   set P[I, J] = P[I, J] (or) (P[I, K] (AND) P[K, J])

   (1) (1) = 1

   (End of loop)
   (End of step-3)
   (End of step-3 loop)
5) Exit

(Direct path)
X → to → Z
↓
Intermediate node 'K'

FAILN 2



* Adjacency matrix:

A = P =

	X	Y	Z	W
X	0	0	0	1
Y	1	0	1	1
Z	1	0	0	1
W	0	0	1	0

* P_X, P_Y, P_Z, P_W [Path matrix]

P_X =

	X	Y	Z	W
X	0	0	0	1
Y	1	0	0	1
Z	2	0	0	1
W	0	0	1	0

P_Y =

	X	Y	Z	W
X	0	0	0	1
Y	1	0	1	1
Z	1	0	0	1
W	0	0	1	0

P_Z =

	X	Y	Z	W
X	0	0	0	1
Y	1	0	1	1
Z	1	0	0	1
W	1	0	1	1

$W \to X$

$W \to W$

$W \to Z \to (\wedge) X$ = 1

$W \to X$

$W \to Z \to X$

0 1 = 1

P_W =

	X	Y	Z	W
X	0	0	0	1
Y	1	0	1	1
Z	1	0	0	1
W	1	0	1	1

$Z \to Z$

$Z \to W \to Z$

1 $\wedge$ 0 = 0

$X \to X$

$X - W - Z - X$

$X \to Y$

$X \to W \to Y$

1 $\wedge$ 0

= 0

$X \to Z$

$X \to W \to Z$

1 $\wedge$ 1 = 1

x → w, w, 1, 1	y → x, w, 1, 1	y → y, w, 1, 0
$1 \wedge 1 = 1$	$1 \wedge 1 = 1$	$1 \wedge 0 = 0$

∴ Intermediate state can 1 (or) 2

↳ shortest path algorithm:- [Floyd's Alg]

1) Repeat for I, J = 1, 2, -- M [initialises 0]

W[I, J] = 0, then set Q[I, J] - INFINITY.

Else set Q[I, J] = W[I, J]

[End of loop]

2) Repeat step - 3 & 4 for K = 1, 2, -- M [updates Q]

3) Repeat step - 4 for I = 1, 2, -- M

4) Repeat for J = 1, 2, -- M

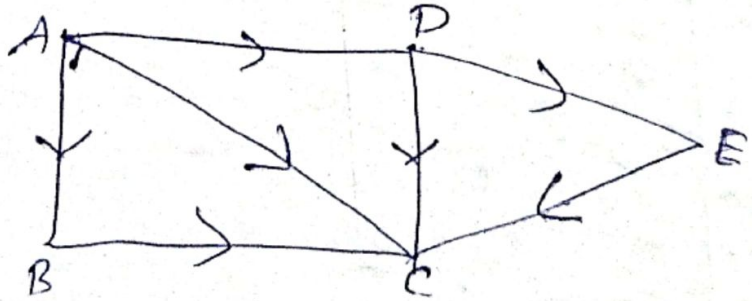
Set Q[I, J] = MIN (Q[I, J], Q[J, K] + Q[K, J])

(End of loop)

(End of step-3 loop)

(End of step-2 loop)

5) Exit



* Adjacency matrix:

$A = P =$

	A	B	C	D	E
A	0	1	1	1	0
B	0	0	1	0	0
C	0	0	0	0	0
D	0	0	1	0	1
E	0	0	1	0	0

Q[A, B] = min(Q[A, B], Q[B, K] + Q[K, B])

= min(1, Q[1,

P_A =

	A	B	C	D	E
A	0	1	1		
B		0			
C			0		
D				0	
E					0

A → B

Q[1, 1] = min

P_B =

	A	B	C	D	E
A	0				
B		0			
C			0		
D				0	
E					0

P_C =

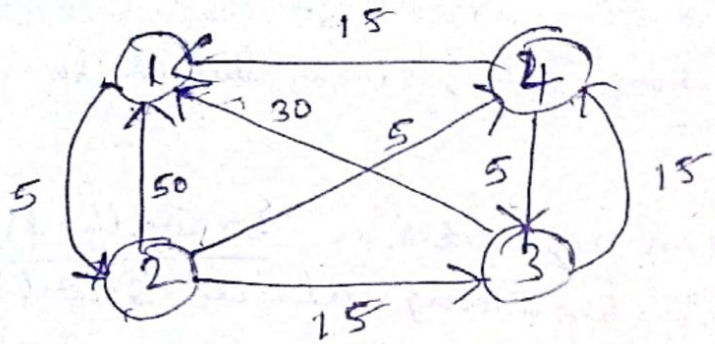
	A	B	C	D	E
A	0				
B		0			
C			0		
D				0	
E					0

P_D =

	A	B	C	D	E
A	0				
B		0			
C			0		
D				0	
E					0

P_E =

	A	B	C	D	E
A	0				
B		0			
C			0		
D				0	
E					0



$1 - 3 \Rightarrow \infty$

$W = Q =$ (Cost matrix)

	1	2	3	4
1	0	5	∞	∞
2	50	0	15	5
3	30	∞	0	15
4	15	∞	5	0

If same it is '0'.

$Q_1 =$

	1	2	3	4
1	0	0(5)	∞	∞
2	50	0	15	5
3	30	35	0	15
4	15	20	5	0

$\min(\infty, 35)$
$\min(35)$
$1 \rightarrow 2$
5

- $Q[1,1] = \min(Q[1,1], Q[1,1] + Q[1,1])$
  $= \min(0, 0+0)$
  $\min(0,0) \rightarrow \min = 0$
- $Q[1,2] = \min(Q[1,2], Q[2,2] + [2,2]$
  $= \min(5, 0) \rightarrow \min 0$
- $Q[1,3] = \min(Q[1,3], Q[3,3] + [3,3]$
  $= \min(\infty, 0+0)$
  $= \min(\infty, 0) \rightarrow \min = \infty$
- $Q[1,4] = \min(Q[1,4], Q[4,4] + [4,4])$
  $= \min(\infty, 0+0)$
  $= \min(\infty, 0) \rightarrow \min = 0$

↳ Traversing a Graph:-

1) BFS :- For this, graph should be directed.

2) DFS

1) BFS :-

This algorithm executes a breadth-first search on graph G beginning at a starting node.

1) Initialize all nodes to the ready state (STATUS = 1)

2) Put the starting nodes in QUEUE & change its status to the waiting state (STATUS = 2)

3) Repeat step-4 & 5 until QUEUE is empty

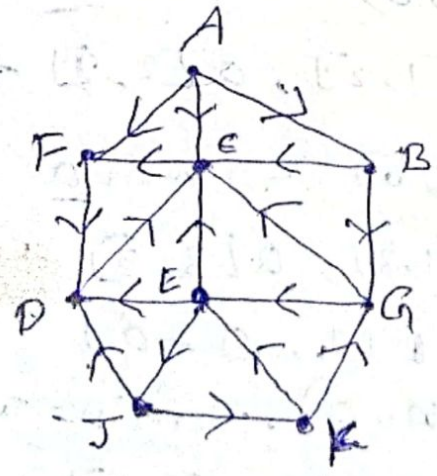
4) Remove the 1st node 'N' of QUEUE. Process N and change the status of N to the processed at state (STATUS = 3)

5) Add the rear of QUEUE all the neighbours of 'N' that are in the ready state (STATUS = 1) & change this status to the waiting state (STATUS = 2)

[End of step-3 loop]

6) Exit

Eg:-



1) BFS: Tracing [Initially add 'A' to queue & null to orig as follows

(i) Queue: A, F, C, B (TOP; neighbours)
orig: ϕ, A, A, A (Parents of neighbours)
front: 1
rear: 4 (3+1)

A

f = r = 1

A → deleted

Adjacency list

A	F, C, B
B	C, G
C	F
D	C
E	D, C, J
F	D
G	C, E
J	K, D
K	E, G

(ii) Queue: A, F, C, B, D (TOP)
orig: ϕ, A, A, A, F
Front: 2
rear: 5

A, F → deleted

(iii) Queue: A, F, C, B, D (TOP)
orig: ϕ, A, A, A, F
Front: 3
rear: 5

A, F, C → deleted.

(iv) Queue: A, F, C, B, D, G (TOP)
orig: ϕ, A, A, A, F, B
Front: 4
rear: 6

A, F, C, B, → deleted.

(v) Queue: A, F, C, B, D, G (TOP)
orig: ϕ, A, A, A, F, B
Front: 5
rear: 6

A, F, C, B, D → deleted.

(vi) Queue: A, F, C, B, D, G, E (TOP)
orig: ϕ, A, A, A, F, B, G
Front: 6
rear: 7

A, F, C, B, D, G → deleted

(VII) Queue: A, F, C, B, D, G, E, J (top) Front : 7
orig : ϕ, A, A, A, F, B, G, E rear : 8

A, F, C, B, D, G, E → deleted.

(VIII) Queue: A, F, C, B, D, G, E, J (top), K Front : 8
orig : ϕ, A, A, A, F, B, G, E, J (last node) rear : 9

A, F, C, B, D, G, E, J → deleted.

⇒ Back tracking :- We now backtracking from J using array orig to find path 'P', thus,

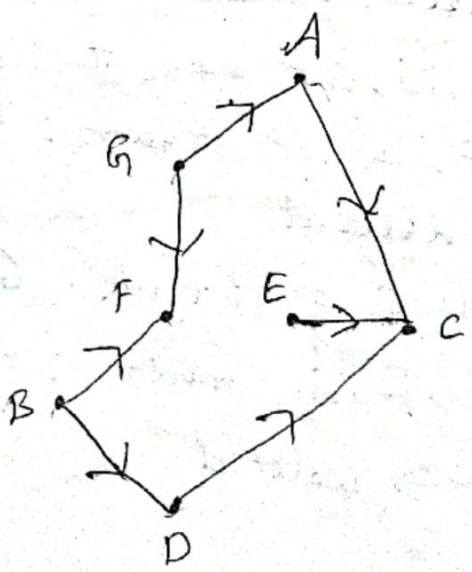
J ← E ← G ← B ← A ← ϕ

2) Remove front element 'A' from queue by setting ~~top~~ front = front + 1 & add to queue the neighbour of 'A' as follows;
(0, 1) → refer;

3) Remove front element 'F' from queue by setting front = front + 1 & add to queue the neighbour of 'F' as follows:
refer → (1, 1).

4) Same procedure for all the nodes;

Eg:-



Adjacency list:

A	C
B	D, F
C	-
D	C
E	C
F	-
G	A, F

1) Initially add 'A' to queue:

(i) queue : A, C (TOP) Front : 1
org : ϕ, A rear : 1

A → deleted.

(ii) queue : A, C. (TOP) Front : 2
org : ϕ, A, ϕ rear : 3

A, C → deleted

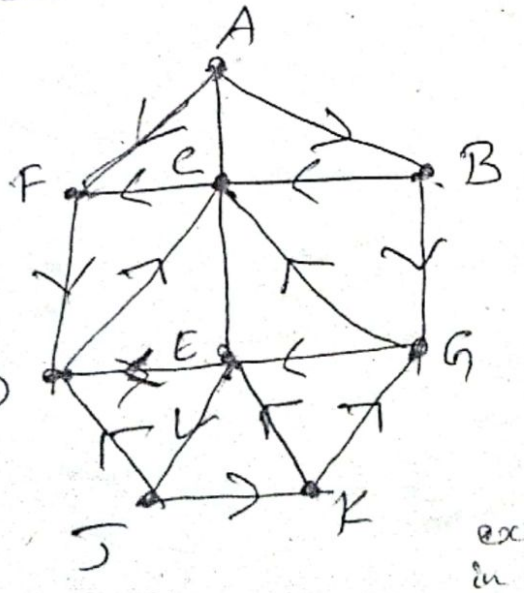
(iii) ~~queue :~~

2) DFS :-

Algorithm :- This algorithm executes a depth first search on a graph 'G' beginning at a starting nodes A.

1) Initialize all nodes to ready state [STATUS = 1]
2) Put the starting node 'A' on to STACK & change its status to waiting state [STATUS = 2]
3) Repeat step-4 & 5 until STACK is empty.
4) POP the top node 'N' of STACK, process 'N' & change its status to the processed state [STATUS = 3]
5) Push on to STACK all neighbours of 'N' that are still in ready state [STATUS=1], & change their status to the waiting states [STATUS = 2]
[End of step-3 loop]
6) Exit.

Eg:-



Adjacency List

A	F, C, B
B	G, C
C	F
D	C
E	D, C, J
F	D
G	C, E
J ✓	D, K
K ✓	E, G

exist in previous node [J]

stack : J

1) print : ϕ (last node) TOP stack : J TOP POP

POP & Push all neighbours D, J

→ stack : D, J (neighbours of TOP J)

2) print : J, → stack : D, E, C

3) print : J, K → neighbours of (K)

4) print : J, K, G, , stack : D, E, F

~~print : J, K, G, E~~

5) Print : J, K, G, C, , stack : D, E,

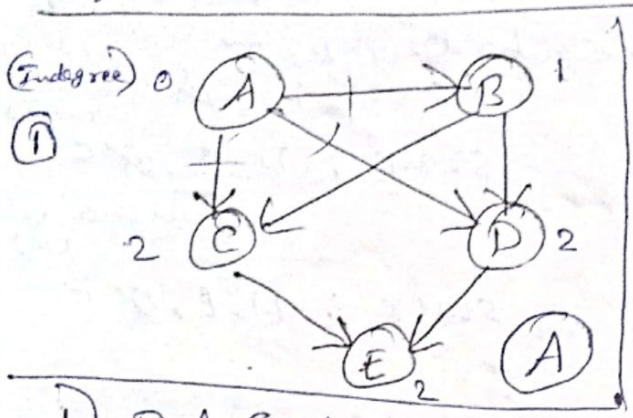
6) Print : J, K, G, C, F, stack : D,

7) print : J, K, G, C, F, E, stack : D POP

8) print : J, K, G, C, F, E, D stak : ϕ

∴ From J, we can reach possible path of all node

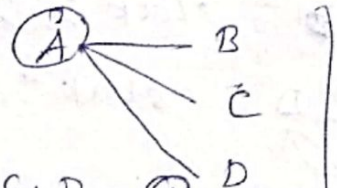
→ Topological sorting :-



The given topology should be satisfied the below condition :-

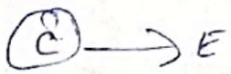
1) DAG :- Directed, Acyclic graph

2) u < v [word text 'u' should be less than 'v']

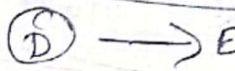


Before B, C, D, (A) should exist

Before C & D, (B) should exist.



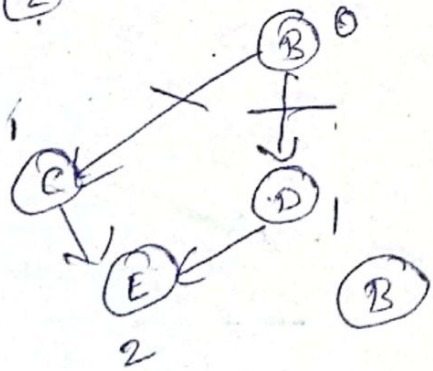
Before E, (C) should exist



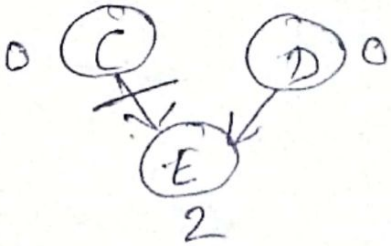
Before E, (D) should exist

3) Indegree for each node having node.

2.



3

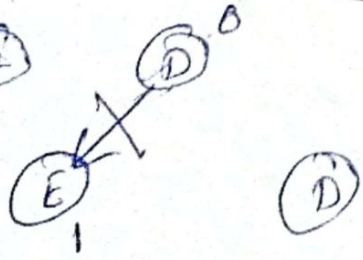


Cyclic - ✗

Directed - ✓

C

4



5

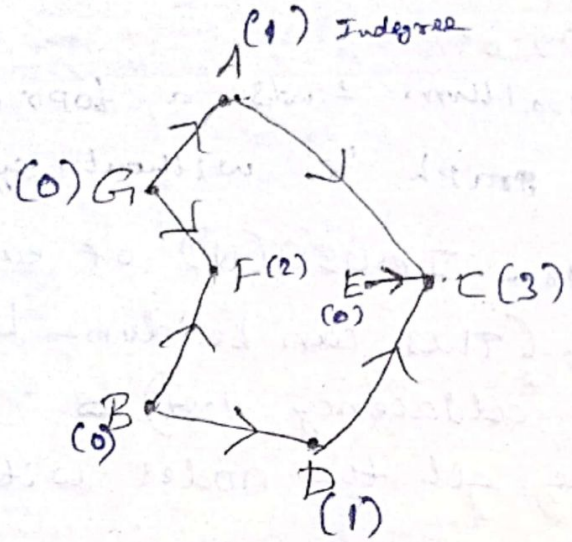
⇒ A, B, C, D, E

↳ Topological Sorting :-

Algorithm : This algorithm finds a topological sort 'T' of a graph 'G' without cycles :

1) Find the indegree INDEG(N) of each node N of G. (This can be done by traversing each adjacency list)
2) Put in a queue all the nodes with zero indegree.
3) Repeat step-4 & 5 until the queue is empty.
4) Remove the front node 'N' of the queue (by setting Front = Front + 1).
5) Repeat the following for each neighbour 'M' of the node 'N'.
   (a) Set indeg(M) = indeg(M) − 1.
   [This deletes the edge from N to M]
   (b) If indeg(M) = 0, then Add M to the rear of the queue.
   [End of LOOP]
   [End of step-3 LOOP]
6) Exit.

Eg:-



- DAG ✓
- $u < v$ ✗
- Indegree of each node ✓

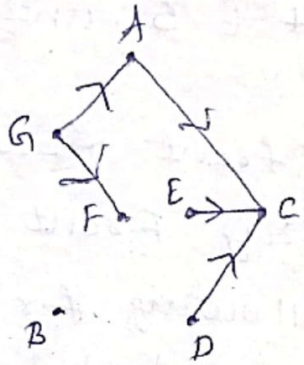
Adjacency list

A	C
B	D, F
C	–
D	C
E	C
F	–
G	A, F

1) B, E, G (0)

Front : 1
rear : 3
queue : ~~B~~, E, G

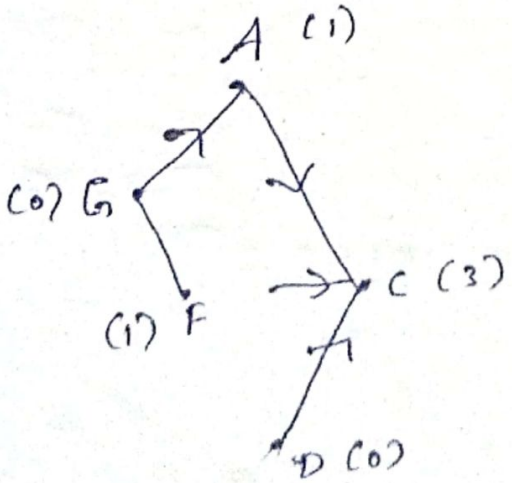
B → deleted.



2) Front : 2
rear : 3
queue : ~~B~~, E, G

(D, F) ⇒ (D) = 1 - 1 = 0 (Insert)
(F) = 2 - 1 = 1

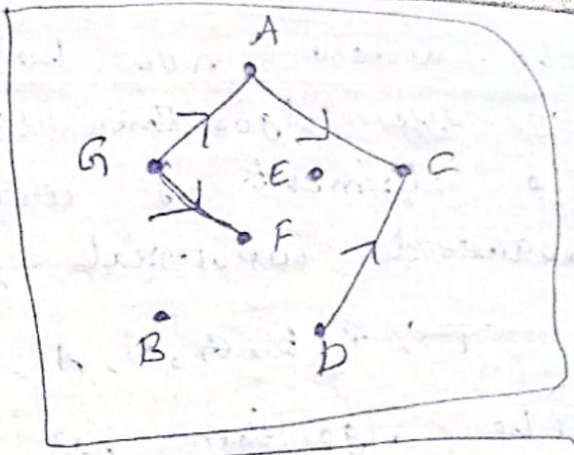
3) Front : 2
rear : 4
queue : B, E, G, D



4) Front : 2
Rear : 4
queue : ~~B~~, ~~E~~, G, D

B, E, → deleted

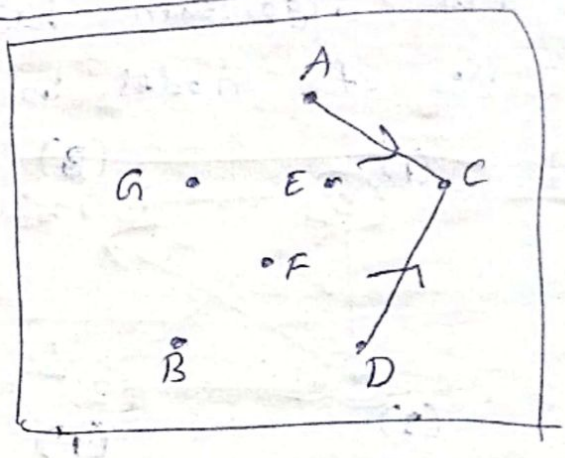
(C) = 3 − 1 = 2



5) Front : 3
Rear : 4
queue : ~~B~~, ~~E~~, ~~G~~, D

B, E, G → deleted

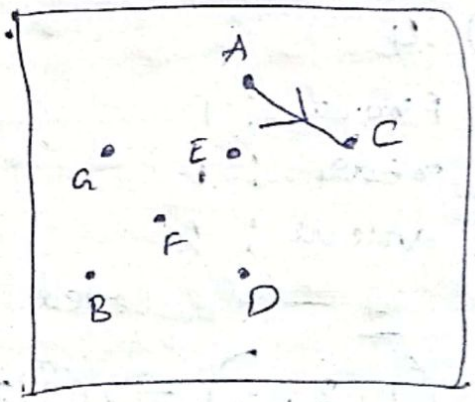
neighbours
(A) = 1 − 1 = 0
(F) = 1 − 1 = 0



6) Front : ~~3~~ 4
Rear : 6
queue : ~~B~~, ~~E~~, ~~G~~, ~~D~~, A, F

B, E, G, D → deleted.

(C) = 2 − 1 = 1



7) Front : 5
Rear : 6
queue : ~~B~~, ~~E~~, ~~G~~, ~~D~~, ~~A~~, F,

B, E, G, D, A → deleted;

(C) = 1 − 1 = 0

8) Front : 6
Rear : 7
queue : ~~B~~, ~~E~~, ~~G~~, ~~D~~, ~~A~~, ~~F~~, C.

B, E, G, D, A, F → deleted.

9) Front : 7
Rear : 7
queue : ~~B~~, ~~E~~, ~~G~~, ~~D~~, ~~A~~, ~~F~~, ~~C~~

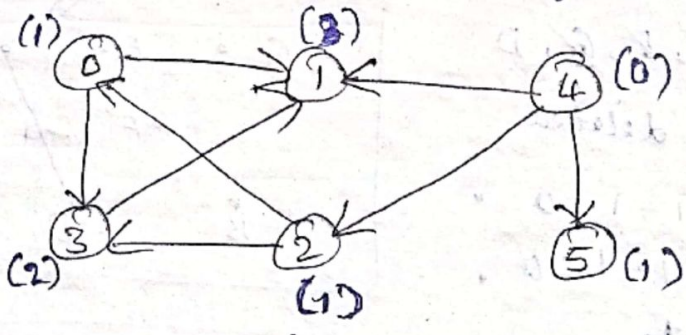
B, E, G, D, A, F, C → deleted

- The queue now has no front elements so the algorithm is completed.
- The elements of array QUEUE give the required topological sort T(S) as follows:

T: B, E, G, D, A, F, C.

- The algorithm has stopped where rear = no. of nodes in the graph 'S'.

Eg:-



1) 4

Front : 1

rear : 1

queue : 4

4 → deleted

(1, 2, 5) ⇒ (1) = 3 - 1 = 2

(2) = 1 - 1 = 0

(5) = 1 - 1 = 0

2) Front : 2

rear : 3

queue : 4, 2, 5

4, 2 → deleted

(0, 3) ⇒ (0) = 1 - 1 = 0

(3) = 2 - 1 = 1

3) Front : 3

rear : 4

queue : 4, 2, 5, 0

(4, 2, 5) → deleted

Adjacency list

0	(1, 3)
1	-
2	(0, 3)
3	(1)
4	(1, 2, 5)
5	-

4) front : 4
rear : 4
queue : 4, 2, 5, 0
4, 2, 5, 0 → deleted
(1, 3) ⇒ (1) = 2 - 1 = 1
(3) = 3 - 1 = 0

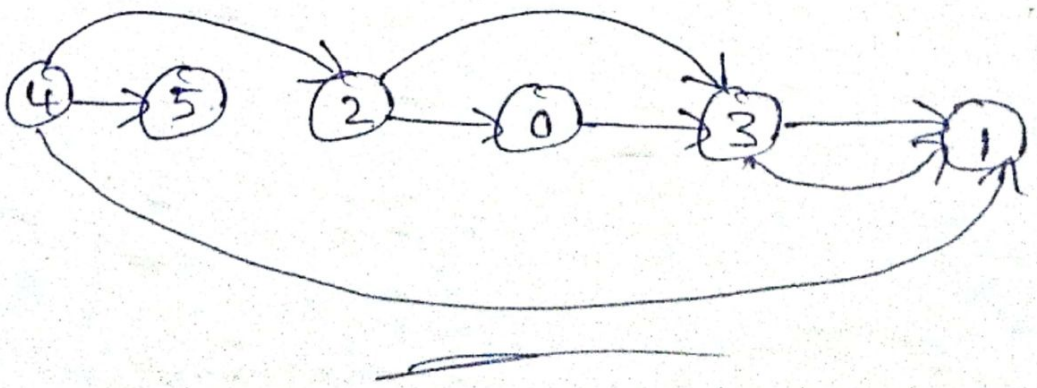
5) Front : 5
rear : 5
queue : 4, 2, 5, 0, 3, 1
4, 2, 5, 0, 3 → deleted
(1) = (1) = 1 - 1 = 0

6) Front : 6
rear : 6
queue : 4, 2, 5, 0, 3, 1
4, 2, 5, 0, 3, 1 → deleted.

---

0 1 1 3 3 0 2 2 4 5 5 4 } Finite automata method

Preorder : 0 1 3 2 4 5
Postorder : 1 3 0 2 5 4
Reverse order : 4 5 2 0 3 1
(Post order)



↳ Greedy method :-

- It is a technique of finding simplest & straight format approach.
- The decision is taken on term (basis) of current available information without verifying about the effect of current decision in future.

(any subset that satisfies given criteria)

⇒ Feasible solution :- That may / may not to optimal (best / most favorable)

* characteristic & features :-

1) To construct the solution in an optimal way.
   - Algorithm maintains 2 sets:
     - one contains closed item,
     - other contains rejected items.

2) Greedy algorithm makes good local choice:
   - an optimal solution,
   - Feasible solution.

* components of Greedy algorithm :-

1) candidate set : A solution is created from the set.
2) selection function :- used to check the best candidate to added to solution.
3) Feasibility function :- used to determine whether candidate can be used to contribute to the solution.
4) objective function :- used to assign value to solution / partial solution.
5) solution function :- used to indicate whether a complete solution has been reached.

* Areas of application :-

1) Finding shortest path.
2) Finding minimum spanning tree [MST]
3) Job sequencing with deadlines.
4) Huffman coding;
5) Fractional knapsack Problem.

* Pseudocode for greedy algorithm :-

```
Algorithm Greedy (a, x)
{
  solution = 0;
  for i = 1 to n do
  {
    x := select(a);
    if feasible (solution, x) then
      solution := union (solution, x);
  }
  return solution;
}
```

(adding)

(1) [Container loading Problem]

Fractional Knapsack problem :- In this

(1) we can divide the weights.

object (o) :	1	2	3	4	5	6	7
Profit (P) :	10	5	15	7	6	18	3
weight (w) :	2	3	5	7	(1)	4	1

n = 7
m = 15 (bag)

$\frac{\text{profit}}{\text{weight}} = \frac{10}{2} = 5$ ✓ ②

⑥ $= \frac{5}{3} = 1.6 \left(\frac{2}{3}\right)$

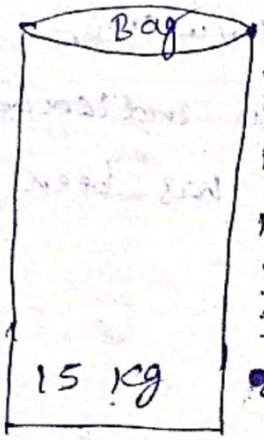
④ $= \frac{15}{5} = 3$

$= \frac{7}{7} = 1$ ⑦

① $= \frac{6}{1} = 6$ ✓ (Highest Profit)

③ $= \frac{18}{4} = 4.5$

⑤ $= \frac{3}{1} = 3$



15 - 1 = 14
14 - 2 = 12
12 - 4 = 8
8 - 5 = 3
3 - 1 = 2
2 - 2 = 0

kg kg
1, $\frac{2}{3}$, 1, 0, 1, 1, 1

$x(x_1, x_2, x_3, x_4, x_5, x_6, x_7)$

- Determining weight :-

$\sum x_i w_i = \frac{1}{\text{kg}} \times \frac{2}{\text{weights}} + \frac{2}{3} \times 3 + 1 \times 5 + 0 \times 7 + 1 \times 1 + 1 \times 4 + 1 \times 1$

$= 2 + 2 + 5 + 1 + 4 + 1 = 15$

$\sum x_i w_i \leq m$ ✓

- Determining Profit :-

$\sum x_i P_i = \underset{\text{kg}}{1} \times \underset{\text{Profit}}{10} + \frac{2}{3} \times 5 + 1 \times 15 + 0 \times 2 + 6 \times 1 + 1 \times 18 + 3 \times 1$

$= 10 + 3 + 15 + 6 + 18 + 3$

$= 55$

$\max \sum x_i P_i$ } objective function

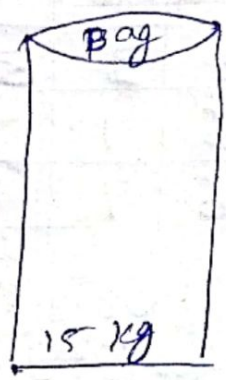
(2) object (o) : 1 2 3 4 5 6 7

Profit (P) : 5 10 15 7 8 9 4 (51)

weight (W) : 1 3 5 4 1 3 2 (15)

n = 7
m = 15

$\frac{\text{profit}}{\text{weight}}$ =	$\frac{5}{1}$	$\frac{10}{3}$	$\frac{15}{5}$	$\frac{7}{4}$	$\frac{8}{1}$	$\frac{9}{3}$	$\frac{4}{2}$
=	5 (2)	3.3 (3)	3 (4)	1.7 (X)	8 (1)	3 (5)	2 (6)



15 - 1 = 14
14 - 1 = 13
13 - 3 = 10
10 - 5 = 5
5 - 3 = 2
2 - 2 = 0

(o, i) $x(x_1, x_2, x_3, x_4, x_5, x_6, x_7)$

1 1 1 0 1 1 1

1 → completely inserted

0 → not completely inserted.

$\sum x_i w_i = 1 \times 1 + 1 \times 3 + 1 \times 5 + 0 \times 4 + 1 \times 1 + 1 \times 3 + 1 \times 2$

$= 1 + 3 + 5 + 1 + 3 + 2$

$= 15$

$\sum x_i w_i \leq m$ ✓

$\sum x_i p_i = 1 \times 5 + 1 \times 10 + 1 \times 15 + 0 \times 7 + 1 \times 8 + 1 \times 9 + 1 \times 4$

$= 5 + 10 + 15 + 8 + 9 + 4$

$= 51$

$\max \sum x_i p_i = 51$ ✓