PIE Tech

POLLACHI INSTITUTE OF ENGINEERING AND TECHNOLOGY

(Approved by **AICTE** and Affiliated to **Anna University**) *sky is the limit*

Department of Electrical and Electronics Engineering

Regulation 2021

II Year /III Semester

CS3353-C Programming and Data Structures

CS3353 C Programming & Data Structures.

UNIT-I C PROGRAMMING FUNDAMENTALS

Data types - Variables - Operations - Expressions and Statements - Conditional Statements - Functions - Recursive functions - Arrays - Single and Multi Dimensional Arrays.

Introduction:

→ C is a structured Programming Language developed by a programmer Dennis Ritche.

→ It is also called as a high Level Language.

→ It has small Instruction set.

→ It is Case Sensitive Language.

Keywords, Variables and Constants

→ Keywords are the standard words in C.

→ All keywords are written in lower case. (small letter).

→ Each words which have special meaning. The meaning of keywords cannot be changed.

→ Some of keywords used in C are,

int do while for break continue
double goto float char enum
switch void long short case

Identifiers:

→ Identifier is a Collection of alphanumeri Characters in which the first Character mu not be numeric.

→ Name of the Identifier must not be the keyword.

Validity of Variable names:

→ The first letter of the variable name must not be digit or any special character.

→ Special Characters such as $, #, % are not allowed. except underscore (_).

→ Variable name is case Sensitive.

→ Blank space, special characters, commas, use of quotes are not allowed.

→ Arithmetic operators should not be used.

→ Variable name should be Informative. It should describe the purpose of it.

Ex:

Count, tax_id, INDEX, xyz

#id, 1A — Not valid.

Constants:

The specific alphabetical (or) numerical value that never gets change'd during the processing of the Instruction is called Constant.

EX: PI.

Comparision b/w Constant and Variables.

Constants	Variables
1. Constant can not be changed.	Values of the variables may be changed during processing.
2. Storage location is given a name.	storage location is given names.
3. EX PI = 3.142	EX: Name = "AAA"

Header Files:

It Contain standard library functions. For using these library functions directly in the program; it is necessary to include the header file at the beginning of the program.

Commonly used header files are,

1. stdio.h - It is standard input output header file. In this file the functions for printf, scanf, fprintf, fscanf are defined.

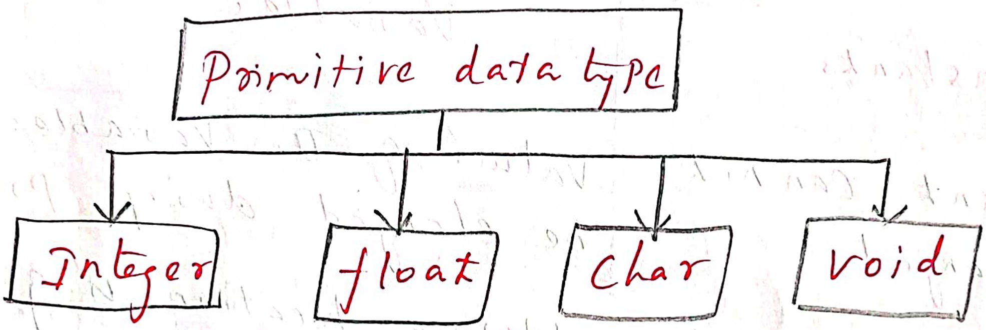
flo

2. Conio.h - It is Console Input Output Header file. By using clrscr(), the Console Screen gets cleared.

3. math.h - All the functionalities related to mathematical operations are defined.

4. alloc.h - Allocating the memory dynamically.

Datatypes:



Integer Datatype:

→ These datatypes are used to store whole number.

Type	Size	Range
int (or) Signed int	2	-32,768 to 32767.
unsigned int	2	0 to 65535.
Short int (or) Signed short int	1	-128 to 127.
unsigned short int	1	0 to 255
long int (or) signed long int	4	-2,147,483,648 to 2,147,483,647.
Unsigned long int	4	0 to 4,294,967,295.

float Datatype:

These are the data types used to store the real numbers. (ie number with fractional part).

Type	Size	Range
float	4	3.4E -38 to 3.4E +38
double	8	1.7E -308 to 1.7E +308.
long double	10	3.4E -4932 to 3.4E +4932.

Char Datatype

It is used to store the character value.

Type	Size	Range
char or signed char	1	-128 to 127
unsigned char	1	0 to 255.

Void data type:

Void data types means no value. This datatype normally associated with a function that return no value.

Expressions using Operators:

1. Arithmetic operator	+	Addition
	—	subtraction
	*	Multiplication.
	/	Division
	%	Mod

2. Relational operator	<	- Loss than
	>	- greater than
	<=	less than equal to.
	>=	greater than equal to.
	==	Equal to
	!=	NOT equal to.
3. Logical operator	&&	And.
	||	Or
4. Assignment	=	Is assigned to
5. Increment	++	Increment by one.
6. Decrement	--	Decrement by one.

Conditional Operator

$$\boxed{\text{Condition ? } \underbrace{\text{expression1.}}_{\text{True Condition}} \text{ : } \underbrace{\text{expression2}}_{\text{False condition.}}}$$

EX:

a>b ? true : false.

If a is greater than b, If yes then the value will be true otherwise false.

Precedence of operator

1. Functions
2. Power.
3. MOD.
4. *, /
5. +, -
6. =, <, >; <=, >=, <>
7. NOT
8. AND
9. OR.

EX:

a = 3/3 * 4 + 3 / 8 + 3

a = 3/3 * 4 + 3 / 8 + 3   operation /

= 1 * 4 + 3 / 8 + 3.   Operation *.

= 4 + 3 / 8 + 3   operation /.

= 4 + 0 + 3   Operation +.

= 7.

Input and Output Operations

In C inputting some data is done by Scanf and outputting or printing the data on console is done by Printf statement.

Syntax:

```
Scanf (format specifier, Variables);
```

EX:

```
scanf ("%d", &val);
```

```
printf (format specifier, variables);
```

EX:

```
printf ("%d", val);
```

Decision Making and Conditional Statement

1. If Statement
2. While Statement
3. Do-While statement
4. Switch Case "
5. For Loop

1. IF Statement:

Two types of If Statement. They are,

Simple IF - Single statement
Compount IF - group of statement.

	Types	Syntax	Example.
1.	Simple if	if (condition) statement	if (a<b) printf("a is smaller than b")

2.	Compound if	if (Condition)
		{ statement1;
		...
		....
		}
			if (a<b)
			printf ("a is smaller than b")
			printf("b is larger than a").

3.	if..else	if (condition)
		statement;
		else
		statement;
			if (a<b)
			printf ("a is smaller than b");
			else
			printf ("a is larger than b")

4.	Compound if...else	if (condition)
		{ statement1
		....
		}
		else
		{ statement1
		- - -
		- - -
		}
			if (a<b)
			{
			printf ("a is smaller than b")
			printf (" b is larger than a")
			}
			else
			{ printf ("a is larger than b")
			} printf ("b is smaller than a")

if.. else if

```
if (condition)
{ statement 1;
  ---
}
else if (condition)
{ statement 1;
  ---
}
else
{
 statement 1;
  ---
  --
}
```

```
if (a<b)
printf ("a is smaller than b");
else if (a<c)
printf ("a is smaller than c");
else
printf ("a is larger than b & c");
```

2. While Statement

While statement is executing repeatedly until the condition is false.

Simple While

```
While (condition)
   Statement
```

```
While (a<10)
printf ("a is smaller than 10");
```

Compound While

```
While (condition)
{ Statement 1;
  ---
}
```

```
While (a<10)
{
 printf ("a is less than b");
 a++;
}.
```

3. Do. While

It is used for repeated exeution. Difference between while and do. while is that, while statement the condition is checked before executing any statement whereas do while statements, the statement is executed first and then condition is tested.

fu

do. While

```
do
{
 statement;
 - -
 - -
} While (condition);
```

Ex: (10)

```
do int a=0
{
printf ("~~a is less than~~ Hi ECE")
 a++;
} while (a<10);
```

4. Switch case Statement

from multiple cases if only one case is to be executed at a time then switch case statement is executed.

Syntax

```
Switch (condition)
{
Case Caseno: statements
 break;
default: statement
```

Ex:

```
printf (" Enter your choice");
switch (choice)
{
Case 1:
 printf ("you Hi selected 1");
 break;
Case 2:
 printf ("you Hello selected 2");
 break;
default:
 printf ("good bye");
}
```

5. For Loop

It is not a statement it is a loop, using which the repeated executions of statements occurs.

syntax

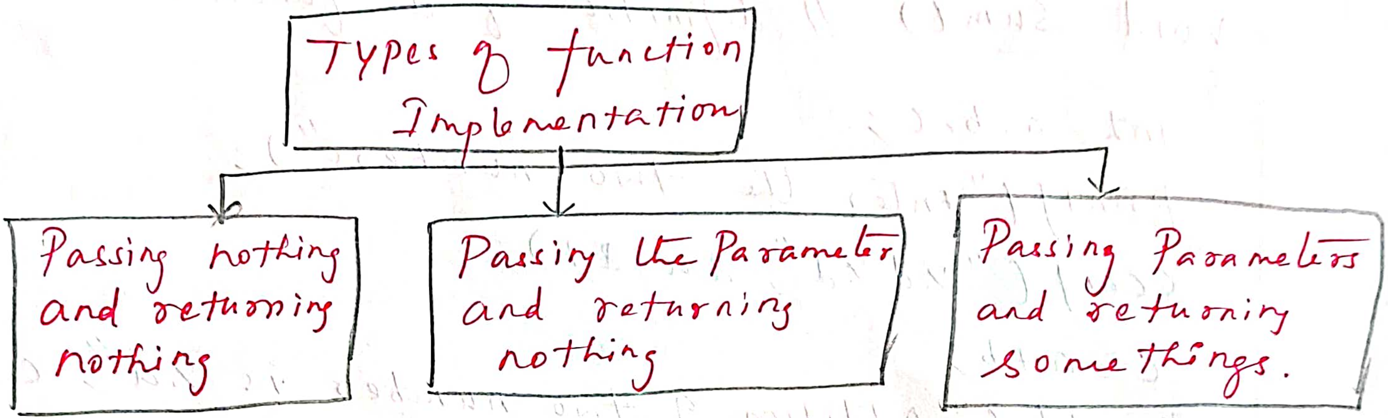
```
for (initialization; termination; step Count)
 statement
```

Ex:

```
for (i=0, i<10, i++)
 c[i] = a[i]+b[i
```

functions

If Certain set of instrustions is required frequently then those instructions are wrapped within a function.


Types of function Implementation
Passing nothing and returning nothing
Passing the Parameter and returning nothing
Passing Parameters and returning somethings.

Declaration of function:

Syntax:

```
Return-type Function_name( Data-type Parameter)
```

Definition of function:

Syntax:

```
Return-type Function-name( Data-type Parameter)
{
    // body of function.
}
```

Call the function:

Syntax:

```
Function-name (Parameters);
```

EX:

```
void main()
{
   void sum(); // declaration of function.
   Sum(); // call the functions.
}
void sum() // definition of the function.
{
  int a,b,c;
  printf("Enter the two numbers");
  scanf("%d%d", &a,&b);
   c=a+b;
  printf("Addition of two number is %d", c);
}
```

Parameter Passing Method:

Type1 : Passing nothing and Returning nothing.

EX:

```
 void main()
 {
    void sum();
    sum();
 }
void Sum()
{
  int a,b,c;
  printf("Enter the two numbers");
  scanf("%d%d", &a,&b);
   C= a+b;
  printf("Addition of two numbers is %d", c);
}
```

→ In above example no parameter is passed and there is no return value from the function.

→ Void data indicates that the function is return nothing.

TYPE 2: Passing the Parameter and returning Nothing

EX:

```
  main()
  {
    int a,b;
    void sum ( int a, int b );
    printf ("Enter two numbers");
    scanf ("%d%d", &a ,&b);
    Sum (a, b);
  }
 void sum ( int x, int y)
  {   int c;
      c = x+y;
      printf(" Addition is %d", c);
  }
```

Here Parameter a, b are passed.

Two methods of parameter Passing.

1. Call by Value
2. Call by Reference.

Difference between Call by Value + Call by Ref.

prin
sc

	Call By Value	Call By Reference
1.	When a function is called, then in that function actual values of the parameters are passed.	When a function is called, then in that function. addresses of the parameters are passed.
2.	The Parameter are simple variables.	The parameter are pointer variables.
3.	Compiler executes this type of function slowly as values get copied to formal parameters.	Compiler executes this type of function faster as addresses get copied to the formal parameters.

4.	main()
	{
	x=10;
	fun(x);
	printf("%d", x);
	}
	void fun(x)
	{ x=15;
	}
	O/p is 10
		main()
		{ *x=10;
		fun(x);
		printf("%d", *x);
		}
		void fun(x)
		{ x=15;
		}
		O/p is 15.

TYPE 3: Passing the parameters and returning

EX:

```
void main()
{ int a,b,c;
  int sum(int, int)
printf (" Enter two numbers");
scanf ("%d %d", &a, &b);
 C= sum (a,b);
printf (" The Addition is =%d", c);
}
 int sum ()
 {
   C= a+b;
   rethorn c;
 }
```

1) Write a C program to interchange two variables without using third variables.

```
#include<stdio.h>
void main()
{
  int a=11;
  int b=20;
  a =a+b;  31
  b = a-b;  11
  a = a-b;  20
  printf ("\n a=%d b=%d", a, b)
}
```

2) Write a C program to get two numbers and exchange these numbers using pass by value and pass by reference.

```
#include <stdio.h>
void swap_by_value (int a, int b)
{ int temp;
   temp = a;
   a = b;
}  b = temp;
```

```
void swap_by_ref (int *a, int *b)
{  int temp;
   temp = *a;
   *a = *b;
   *b = temp;
}
int main()
{  int x, y;
printf (" Enter first number:");
scanf ("%d", &x);
printf (" Enter second number:");
scanf ("%d", &y);
printf ("Before swapping x = %d and y = %d", x, y);
Swap_by_value (x, y) // call by value.
printf (" After swapping x = %d and y = %d", x, y);
printf (" Before swapping x = %d and y = %d", x, y);
Swap_by_ref (&x, &y); // call by reference
printf (" After swapping x = %d and y = %d", x, y);
}
```

Recursive Functions

Recursion is a programming technique in which the function calls itself repeatedly for some input.

Properties of Recursion:

1. There must be at least one Condition in the recursive function which do not involve the call to recursive routine. This is Called base Case property.

2. Each recursive call must reduce to some manipulation and must go closer to base case Condition.

EX:

```
if (n==0)
   return 1;
else
   return n*fact(n-1);
```

Write a C program to find factorial of a number using recursive function.

```
#include <stdio.h>
void main (void)
{ int n,f;
  int fact(int n);
  printf(" Enter the number");
  scanf("%d", &n);
  f = fact(n);
  printf(" The factorial of %d is %d ,n,f);
}
```

```
int fact (int n)
{  int x, y;
   ~~if (n<0)~~
   if (n==0)
     return 1;
   x = n-1;
   y = fact (x);
   return (n*y);
}
```

GCD (Greatest Common Divisor)

GCD is calculated using following function

$$\boxed{gcd(a,b) = gcd(b, a \bmod b)} \text{ if } a > b$$

$$gcd(a,0) = a$$

Ex:

Consider two number 30 and 18.

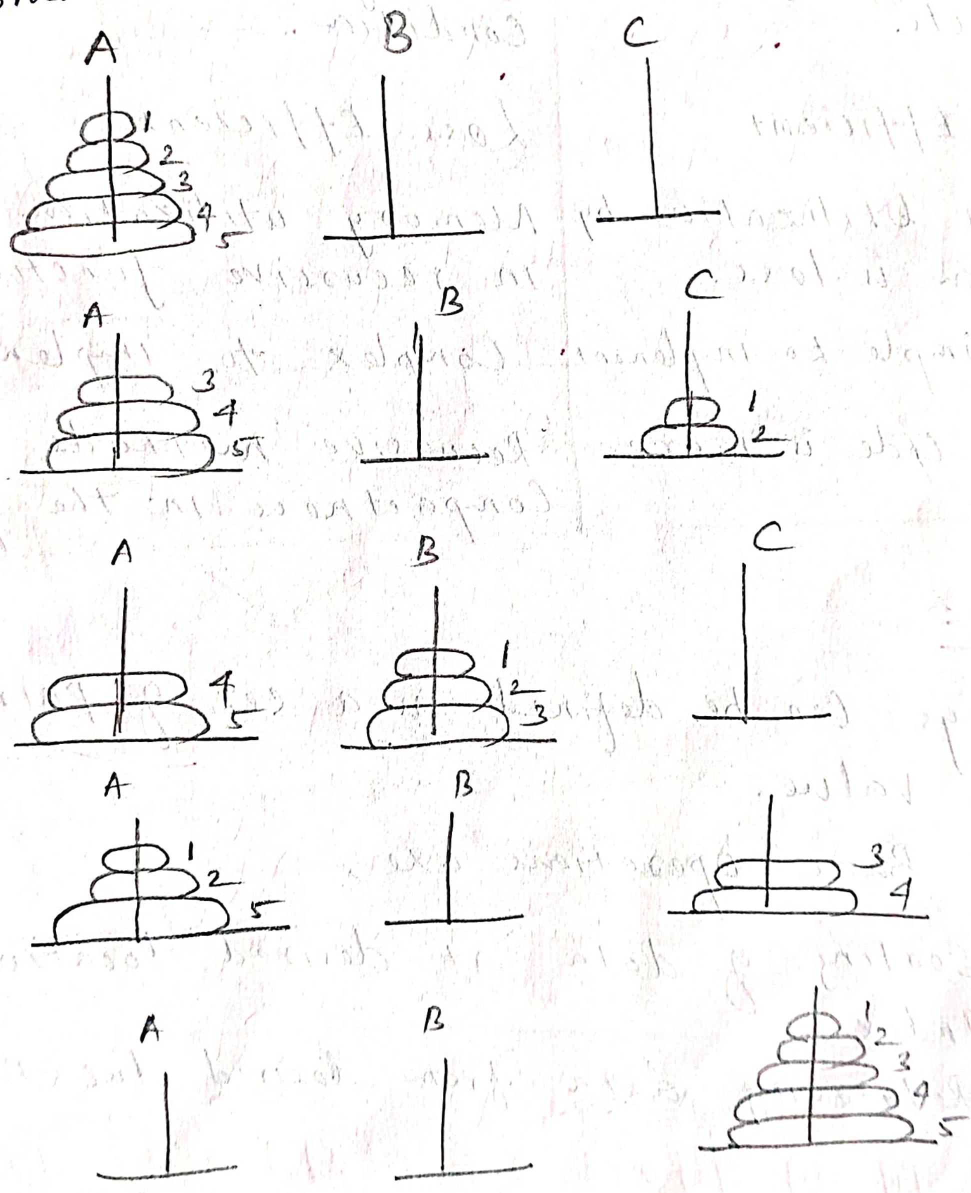
a	b	Remainder a/b.
30	18	12
18	12	6
12	6	0
6	0	GCD = 6

Tower of Hanoi

"Tower of Hanoi" states that the move the five disks from peg A to peg C using peg B as a Auxillary.

Conditions are,

1. Only the top disk on any peg may be moved to any other peg.
2. A larger disk should never rest on the smaller one.



Comparison b/w Iteration and Recursion:

	Iteration	Recursion
1.	Iteration is a process of executing Certain set of instructions repeatedly, without calling the self function.	Recursion is a process of executing Certain set of instruction repeatedly by calling the self function repeatdely.
2.	It is implemented with the help of for, while, do-while programming Constructs.	It is obtained by calling the same function again and again over some Condition.
3.	More Efficient	Less Efficient.
4.	Memory utilization by iteration is less.	Memory utilization is more in recursive functions.
5.	It is simple to implement	Complex to implement.
6.	Line of code is more	Recursive methods bring Compactness in the program.

Arrays:

→ Arrays can be defined as a set of pair-index and the value.

→ Two Basic operations are,

(i) Storing of data at desired location or index.

(ii) Retrieving data from desired location.

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
10	20	30	40	50	60	70	80	90	100

Advantages of Array.

1. Elements Can be retrieved or stored very efficiently in array with the help of index or memory location.
2. All the elements are stored at Continuous memory location. Hence searching of elements from sequential organization is easy.

Disadvantages of Array.

1. Insertion and deletion of elements become Complicated due to sequential nature.
2. Memory fragmentation occurs if we remove the elements randomly.

Syntax:

```
data_type name_of_array [size];
```

Ex:

```
int a[10];
```

Initialization:

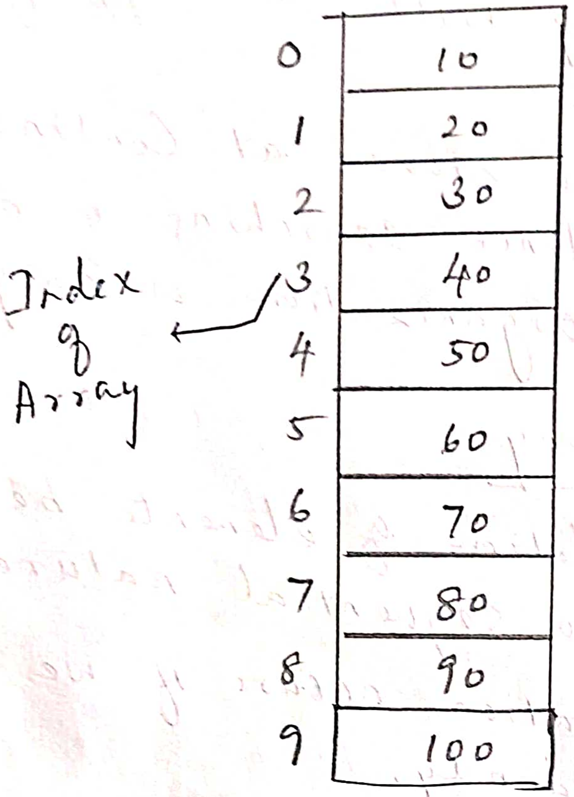
It is possible to initialize an array during declaration only. for example.

```
int a[10] = {0,1,2,3,4,5,6,7,8,9}
```

Types:

1. Single Dimensional Array.
2. Two Dimensional Array.

Single Dimensional Array.



Here a[2] = 30.

a[8] = 90.

Program.

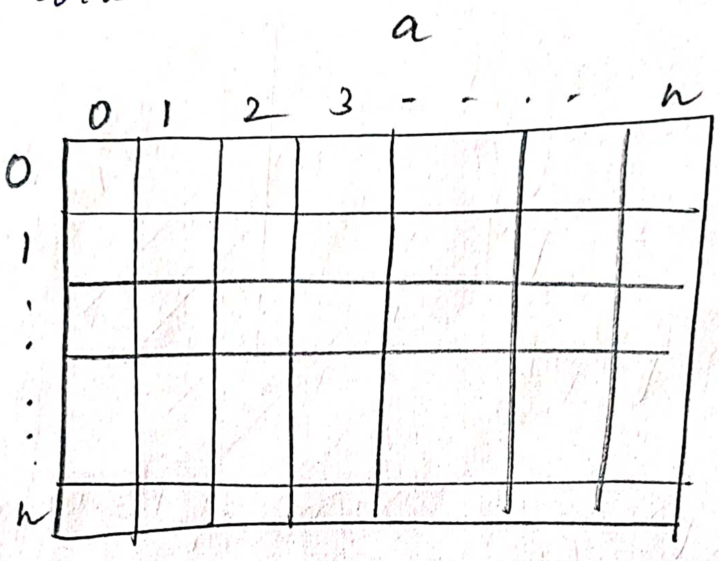
```
#include <stdio.h>
main()
{
  int a[10], index;
  printf("Elements in an Array a");
  for(index=0; index<=9; index++)
  scanf("%d", &a[index]);
  Printf(" You have stored there numbers
                              in the array");
  for(index =0; index <=9; index ++)
   printf ("%d", a[index]);

}
```

Two Dimensional Arrays:

→ It is something which you can Compare with the two storied building!

→ Extra space which is arranged in rows and columns.



Syntax:

Data-type Name-of-Array [row_size] [column-size];

Ex:

```
 int  a[10][10];

for (i=0; i<=2; i++)
{ for (j=0; j<=2; j++)
  {
     scanf ("%d", &a[i][j]);
  }
}
```

Unit-II C programming - Advanced Features.

Structures - Union - Enumerated Data types - Pointers - Pointers to variables, Arrays and functions - File Handling - preprocessor Directives.

Structures:

→ A Structure is a group of items in which each item is defined by some identifiers. These item are called members of the structure.

→ Structure is a Collection of various data items which can be ~~defines~~ of different data types.

Syntax:

```
Struct name
{
    member1;
    member2;
      :
    membern;
};
```

Example:

```
struct stud
{ int roll_no;
  char name [10];
  float marks;
};
struct stud stud1, stud2;
```

Intead of Struct we using typedef.

Comparison between Arrays and Structures.

	Array	Structure
1.	Array is a Collection of similar type of elements.	Structure is a Collection of Variety of elements which can be of different datatypes.
2.	Array elements Can be accessed by the index placed within [ ].	Structure elements can be accessed with the help of .(dot) operator.
3.	To represent an array, array name is followed by [ ].	To represent structure a keyword struct has to be used.

4.	EX:
	int a[20];
		EX:
		struct student {
		int roll_no;
		} char name [20];

Initializing Structure

There are 3 ways by which structure can be defined.

1.
```
struct student {
 int roll_no;
 char name[10];
 float marks;
}S1;
```

2.
```
struct student {
  int roll_no;
  char name [10];
  float marks;
 };
struct student S1;
```

3.
```
typedef struct student
{
   int roll_no;
   char name[10];
   float marks;
} S ;

S, S1;
```

Array of structures.

Structure is used to store information of one particular object and if one has to store large number of such objects then array of structure is used.

Union

Union is a user defined data structure which is just similar to structure. It is used to store members of different data types.

Ex:

```
union Employee
{ int empid;
  char empName[30];
};
```

Difference between structure and Union.

	Structure	Union
1.	The keyword struct is used to define the structure.	The keyword union is used to define Union.
2.	All the members of structure can be accessed at a time.	Only one member of union is accessible at a time
3.	The memory is allocated for each member of the structure	One block of memory is used by all the members of union.
4.	One or more members of structure can be initialized at once	A union may only be initialized with a value of the type of it first member.
5.	It takes more memory.	Less memory.

Enumerated Data types

The enum is an abbreviation used for enumerated data type.

```
#
enum week {mon, Tue..
               sun};
int main()
{ enum week day;
day=
printf("%d",
return 0;
}
```

Syntax:

enum identifier_name { sequence of items };

Ex:

states 0 1 2

enum fruit { Mango, Orange, Banana, Grapes }.

Here mango = 0, Orange = 1, Banana = 2 Grapes = 3.

Pointers

→ Pointers is a variable that represents the memory location of some other variable.

→ purpose of pointer is to hold the memory location and not the actual value.

Advantage of Pointers

Data type * ptr name;

→ It allow the dynamic memory Management.

→ It helps to return more than one values from the function.

→ It allows to Pass arrays, strings and structures efficiently.

→ It provides direct access to memory.

→ It improve the execution speed of Program.

→ It help to build the Complex data structures.

Initialization.

→ Once the pointer variable is declared then it can be used further.

→ pointer variable is basically used to store some address of the variable which is holding some value.

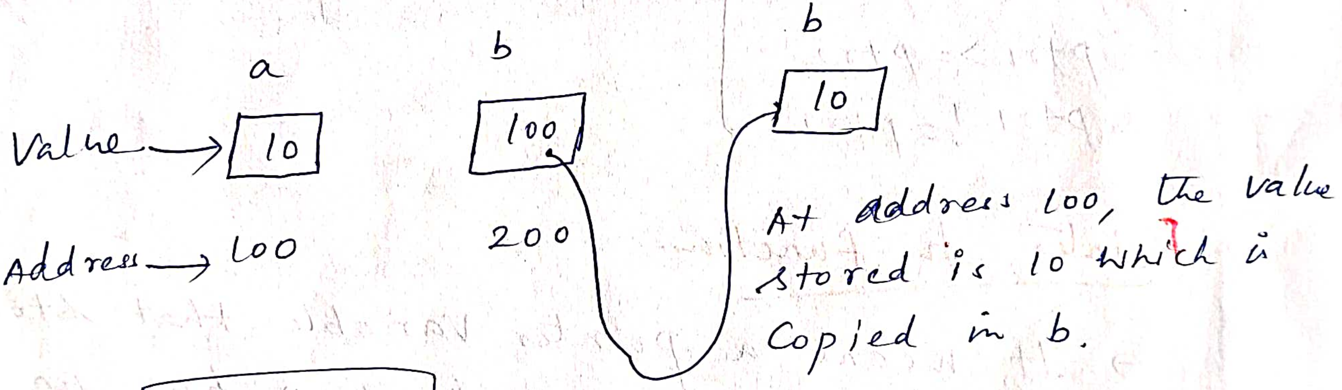
Consider,

```
int *ptr;
int a,b;
a=10;
ptr =&a;
b = *ptr
```

Here,

* means Content at the specified address.

& means address at.



At address 100, the value stored is 10 which is copied in b.

```
a =10
ptr= &a.
 b = *ptr
```

Pointer Arithmetic

Pointer Arithmetic is a method of Calcula: the address of an object with the help of arithmetic operations on pointers.

Operation	Meaning
x = *ptr1 * *ptr2	Multiplication.
x = ptr1 - ptr2	subtraction.
ptr1 -- or ptr1 ++	Incremented or decremented
x = ptr1 + 10	we can add some constant to pointer variable.
x = ptr1-20	subtract
ptr1 < ptr2 ptr1 = ptr2 ptr1 <= ptr2 ptr1 >= ptr2 ptr1 != ptr2	Relational operations are possible on pointer variable

Pointer to function.

→ It means a pointer variable that stores the address of function. The function has an address in the memory same like variable.

Syntax:

```
Return-Type * Pointer-variable (data-type);
```

EX:

```
float(*fptr) (float);
```

File Handling:

→ file is a Collection of records.

→ File of size n has n records in sequence where each record is a collection of one or more fields.

Ex:

Rollno	Name	Marks
1	John	66
2	Mathew	70
3	Steve	60.

Classification of File.

There are two types of files.

1. Text File.
2. Binary File.

→ Text files are files in which textual informatia may be stored with essentially no formatting.

→ Binary files in which any type of data encoded in binary form. The Binary files Contain the binary digits. That means it Contains the sequence of 1's and 0's.

Comparison between Text File and Binary File.

	Text File	Binary file
1.	Text files Contains plain text data.	Binary File contain the data in binary form.
2.	The text file does not contain the graphical data.	Binary file can store the data such as text, graphics, images and sound.
3.	Text files Can be directly read and interpreted.	Binary files cannot be read directly.
4.	Text files are not executable files.	Binary files can be executable files.

File operations

1. Creating a File / opening a File

Syntax:

```
file pointer = fopen ("filename", "mode");
```

Where,

→ filename is the name of file you want to create or open.

→ mode can be,

read - "r"
write - "w"
append - "a".

Ex:

```
fp = fopen("student.dat", "w");
```

2. Closing the File

Syntax:

```
fclose(filepointer);
```

Ex:

```
fclose(fp);
```

3. Setting the pointer to start of file

Syntax:

```
rewind(filepointer);
```

Ex:

rewind(~~filepointer~~ fp);

4. Writing a character to file

Syntax:

```
fputc(character, filepointer);
```

Ex:

```
fput ( ch, fp);
```

5. Reading a character from file

Syntax:

```
fgetc(filepointer);
```

EX:

```
char ch;
ch = fgetc(fp);
```

6. Writing string to file.

Syntax:

```
fputs (string, filepointer);
```

EX:

```
fputs (s, fp);
```

7. Reading string from file

Syntax:

```
fgets (stringaddress, length, filepointer);
```

Ex:

```
char s[80];
fgets (s, 79, fa);
```

8. Writing of characters, strings; integers, flo

to file

Syntax:

```
fprintf (file pointer, "format string", list of
                                        variable
```

9. Reading of variables from file:

syntax:

```
fscanf (filepointer, "format string", list of address
                                      of variables);
```

Ex:

```
fscanf ( fp, "%d %s", &rno, &name );
printf (" Read data is %d %s", rno, name );
```

10. Writing Contents to file through structure:

Syntax:

```
fwrite (pointer to struct, size of struct,
                    no of data items, file pointer);
```

11. Reading Content of file through structure:

Syntax:

```
fread (pointer to struct, size of struct, no of
                    data items, file pointer );
```

Ex:

```
fread ( &s, sizeof (s), 1, fp);
```

Detecting End of file

The end of file is detected using a macro EOF ie (End of file) When this Condition occurs then we should not read the Contents further.

EX:

```
#include <stdio.h>
void main()
{
  FILE *fP;
  char c;
  fP = fopen("toy.c", "r");
  c = getc(fP);
  While (c != EOF)
   {
     putchar(c);
     c = getc(fP);
   }
 fclose(fP);
}.
```

Sequential File Organization

We want to store the marks of 10 students in a file. We will organize the file in following manner.

Record Number	University seat no	Name of Student	Marks
1			
2			
3			
⋮			
10	--	--	--

Preprocessor Directives

→ It is a separate step applied on the sour file before presenting it to compilation process.

→ C preprocessor is a text substitution toll. It instructs the compiler to do all the necessary preprocessing before and after the compilation.

→ preprocessor directive is written at the beginning of the program with the # proceed to i

	Preprocessor	Syntax and Description.
1.	Macro	#define. This macro defines constant value and can be any of basic data type.
2	Header file inclusion	#include <file-name>.
3.	Conditional Compilation.	#ifdef, #endif, #if, #else, #ifndef.
4.	Other directives	#undef, #prgma.

Ex:

1. # define SIZE 10.
   This directive tells the compiler to replace the instance of SIZE with 10.

2. #include <stdio.>
   This directive tells ~~the~~ to get stdio.h from system libraries and add the text to the current source file.

3. #include "myfile.h".

This line tells the Compiler to get myfile.h from local director and add the contents with the current source file.

4. #undef F_SIZE.

#define F_SIZE 80.

It tells to undefine existing F_SIZE and define it as 80.

5. #ifndef MSG.

#define MSG "Welcome".

#endif.

It tells to define MSG only if MSG isn't already defined.

Linear Data Structures - LIST

1) Data structure:

⇒ The way of organizing data structure is called data structures.

Types of data structures are;

⇒ Array, stack, queue, Linked List, Tree, ~~Queue~~ Graph

2) Abstract Data types (ADTs):

(*) An ADT is a mathematical model of a data structure.

<u>Eg:</u> Lists, sets, and Graphs.

1) List ADT: A list is a dynamic Collection of items. (or) elements E1, E2, E3.... En of size 'n' arranged in Linear Sequence.

A list is a size zero or with no elements.

List operations:

1. Insert (X, L)
2. Delete (X, L)
3. Find (X, L)
4. Make empty (L)
5. previous (X/L)
6. Next (X, L)

Implementation of List ADT

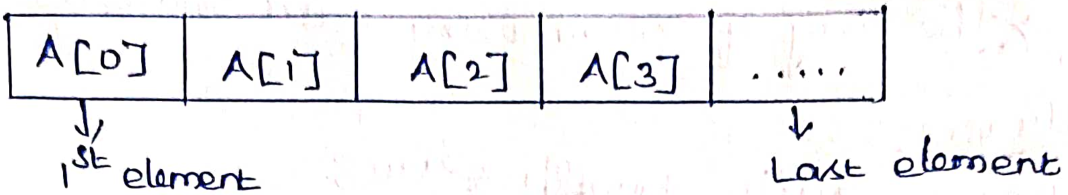
↙ ↘

Array based Implementation

Linked List based Impleme

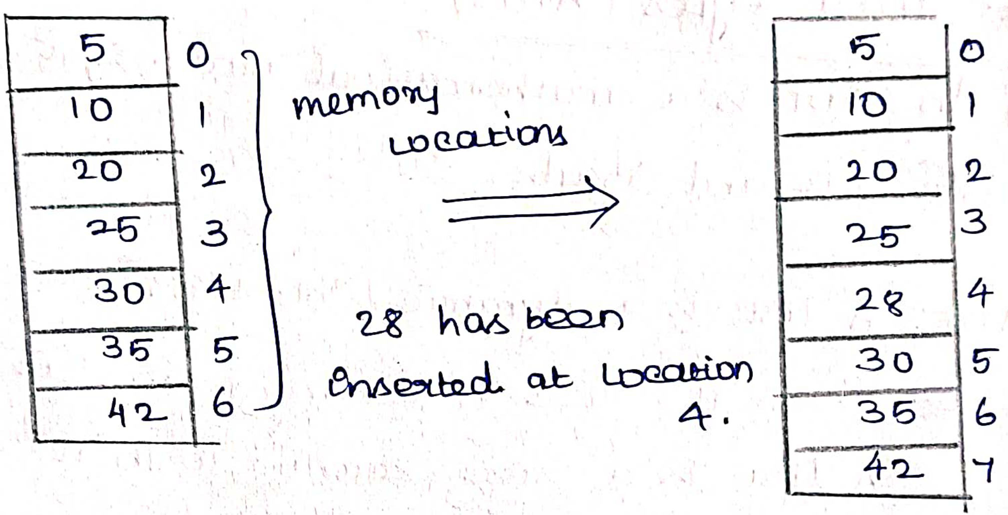
3) Array based Implementation (N/D - 2018)

An array is a DS which can store a fixed sequential collection of elements of the same type.



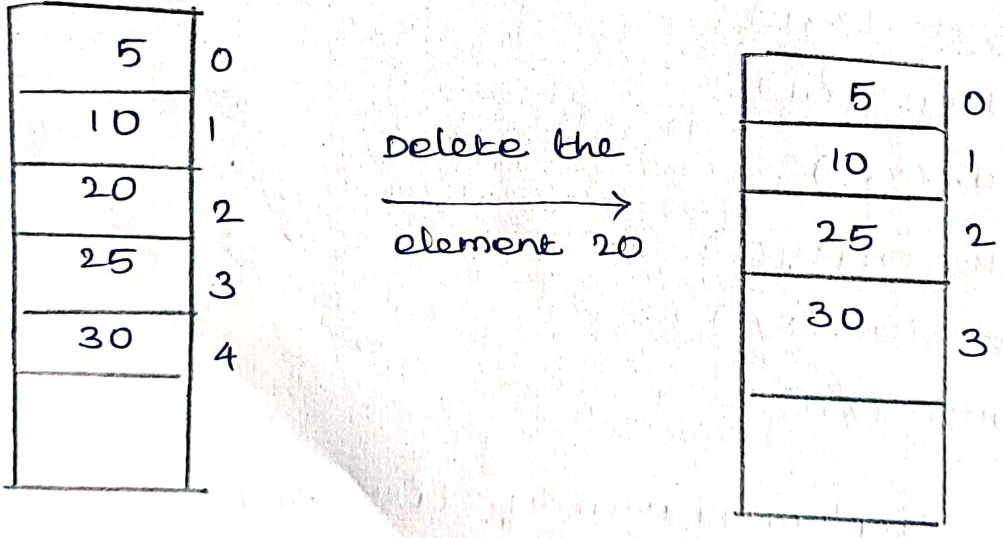
Operations:

① Insertion: List of size 'n'.



28 has been inserted at location 4.

② Deletion: Element from the list can removed.



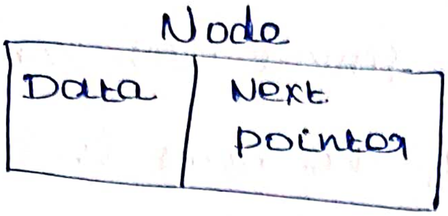
3) Print: Print operation is used to display all elements in list

4) Find: Find operation searches for a particular elements in the List.

+) Linked list Implementation: [N/D - 2019]

It is a linear data structure which is a collection of nodes.

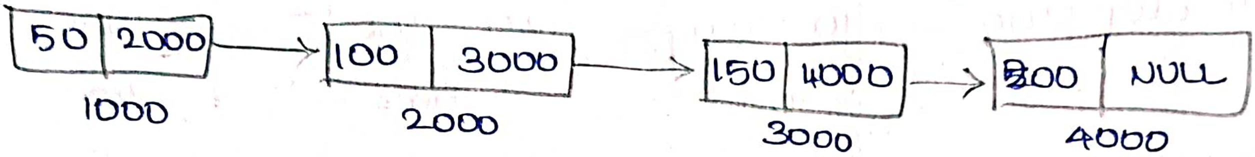
Two parts



Data field - element

Next pointer field - address of next node.

Eg:



Advantages: [A/M - 2019]

(*) Dynamic DS

(*) Size is not fixed.

(*) Easy operations.

Types of Linked List:

(i) Singly linked list

2) Doubly linked list

3) Circularly linked list

Representation:

```
struct node
{
  int data;
   struct node  *next;
};
```

5) Singly linked list: [N|D-2019]

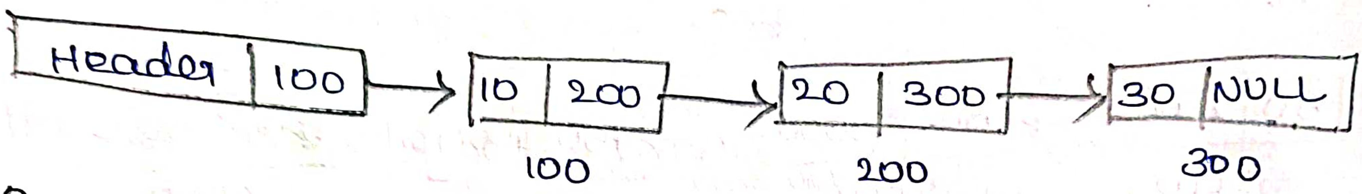
→ Linear Data structure.

→ Collection of nodes.

→ Every node linked with some sequential ma..

→ Header of the next is linked with next node

Eg:



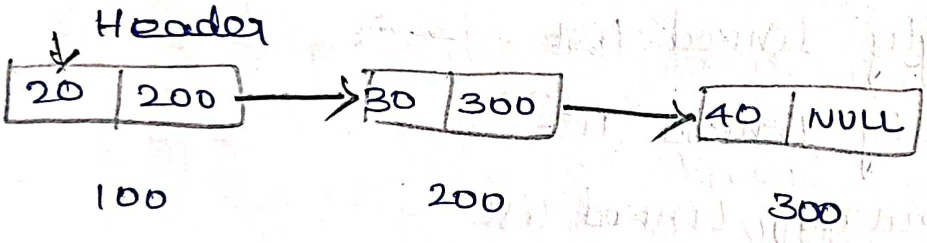
Operations:

① Insertion: [N|D - 2021]

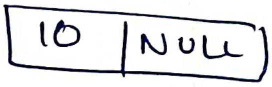
Three ways of Insertion.

① Insertion a node at first place:

Inserting a new node at first poisition of existing singly linked list.



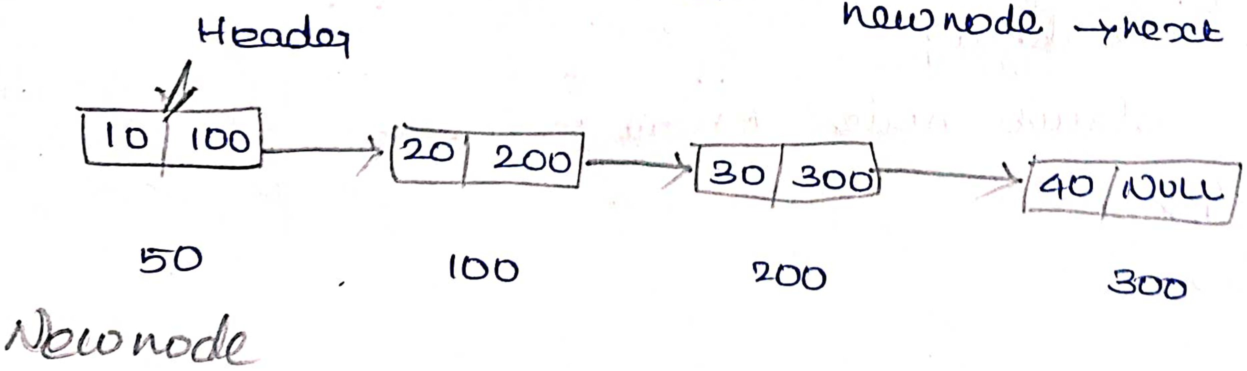
newnode = malloc (size of (struct node));



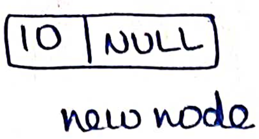
New node.

newnode → data = 10

newnode → next = NULL

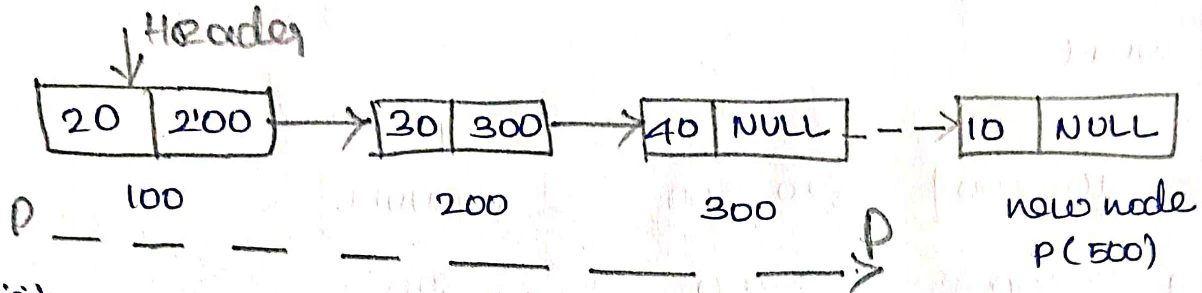


2) Insert a node at Last place:



newnode → data = x

newnode → next = NULL

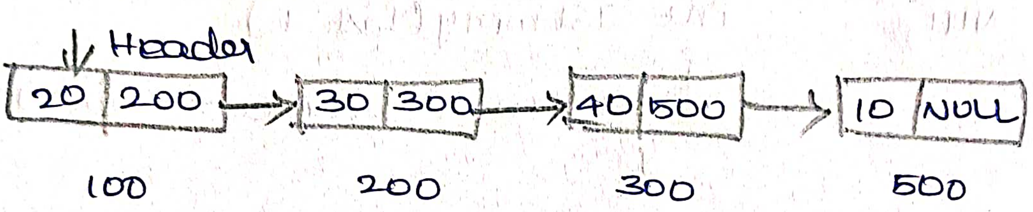


ie)

p = header

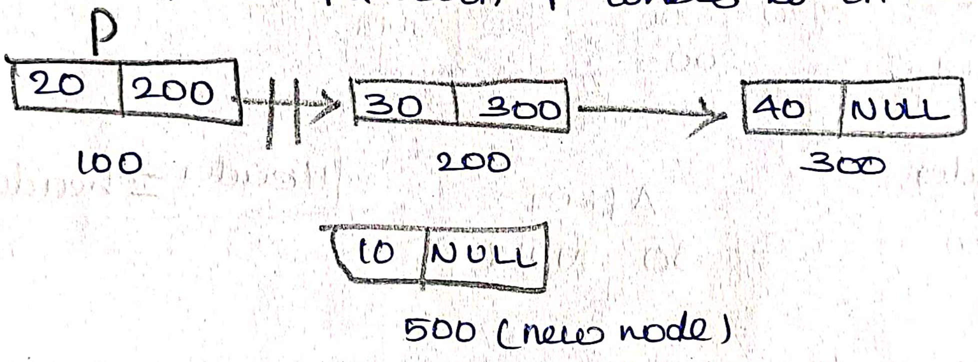
find last position

p → next = newnode.



3) Insert a node in an Intermediate place:

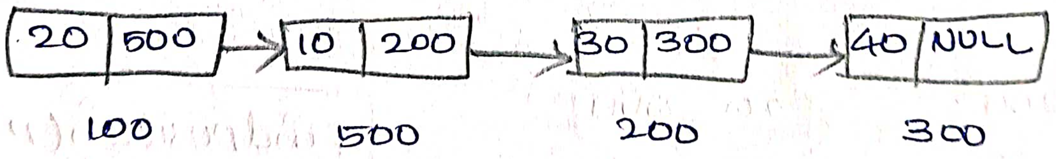
find position p where to insert.



Now,

new node → next = p → next

p → next = newnode.



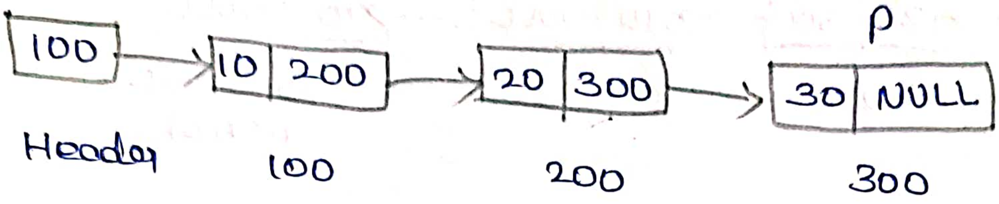
iii) De

② Find operation: find the position of search elem

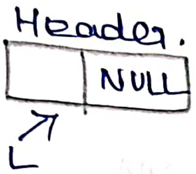
(i) Find :

(ii) Find previous

(iii) Find Next



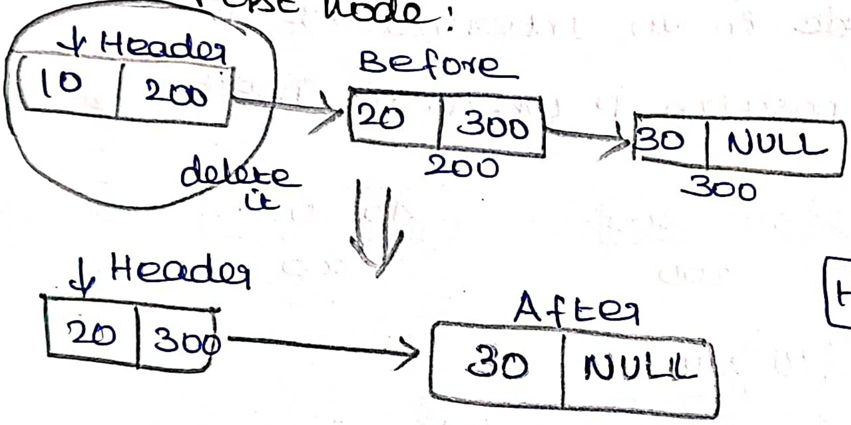
3) check whether the list is Empty:



```
int IsEmpty(List L)
{
 return (L->next==NULL)
}
```

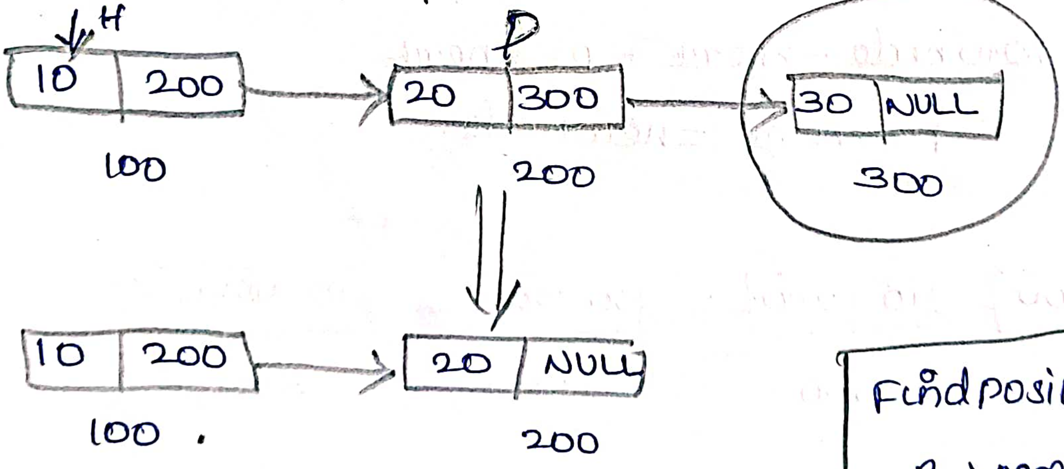
4) Delete operation: Remove the node from List.

(i) Delete first node:



Header = header->next

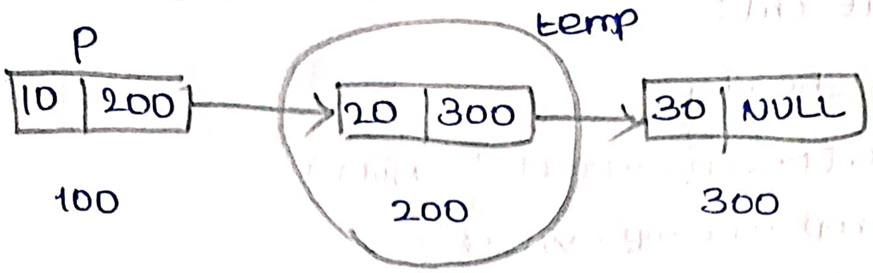
ii) Delete last node:



Findposition(P)

P->next = NULL

iii) Delete Intermediate node:



Find previous(X, L)

P→next = temp→next

free temp.

6) Doubly Linked List:

(*) Linear Linked List - Linear DS.

Three fields are:

- ⇒ Previous Address field
- ⇒ Data field
- ⇒ Next Address field

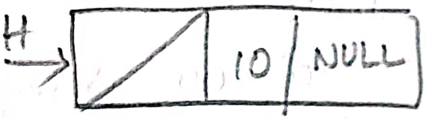
Prev Add	Data field	Next Add

Node

Operations:

(i) Creation: New node creation.

```
struct dnode
{ int data;
  struct dnode *prev;
  struct dnode *next;
};
```



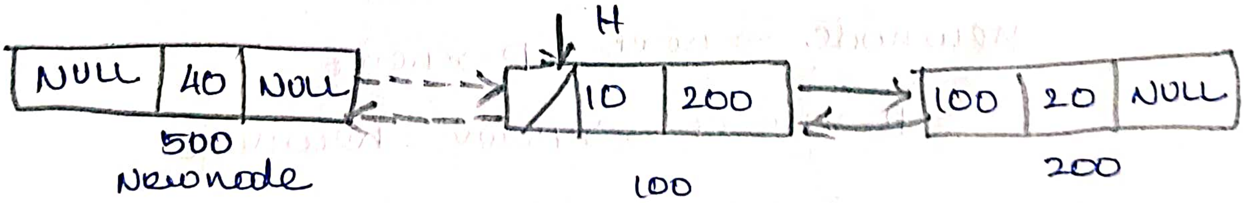
newnode

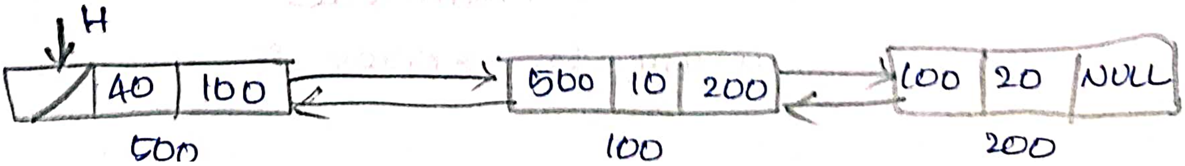
(ii) Insertion:

1) Insert a node at beginning:

```
newnode →next = header;
    header →prev = newnode;
        header = new node.
```

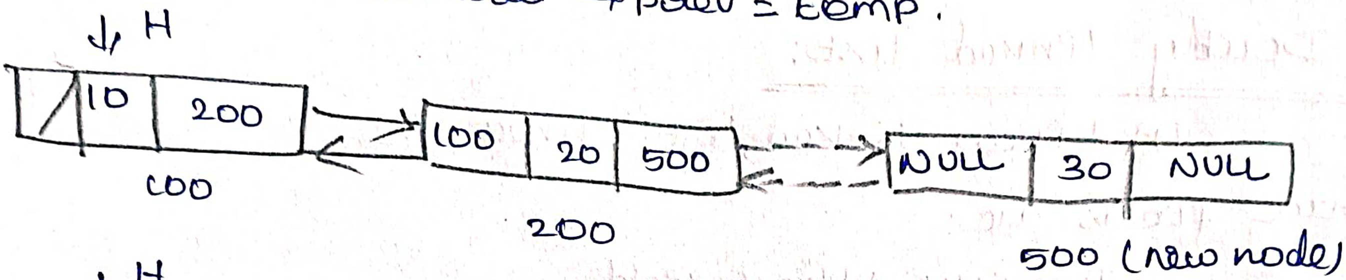
eg:

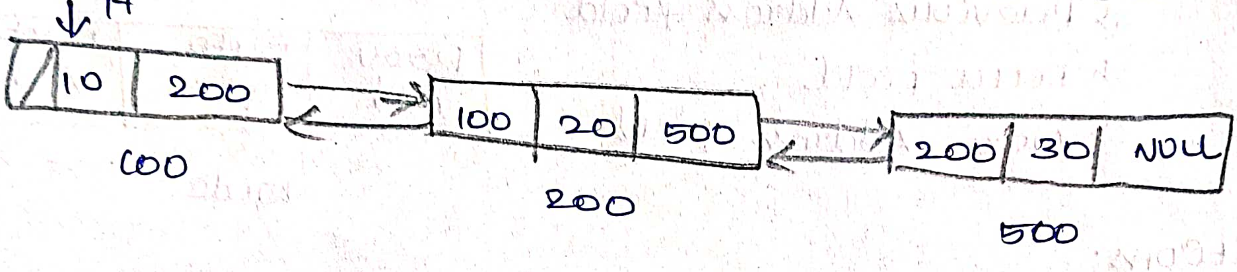




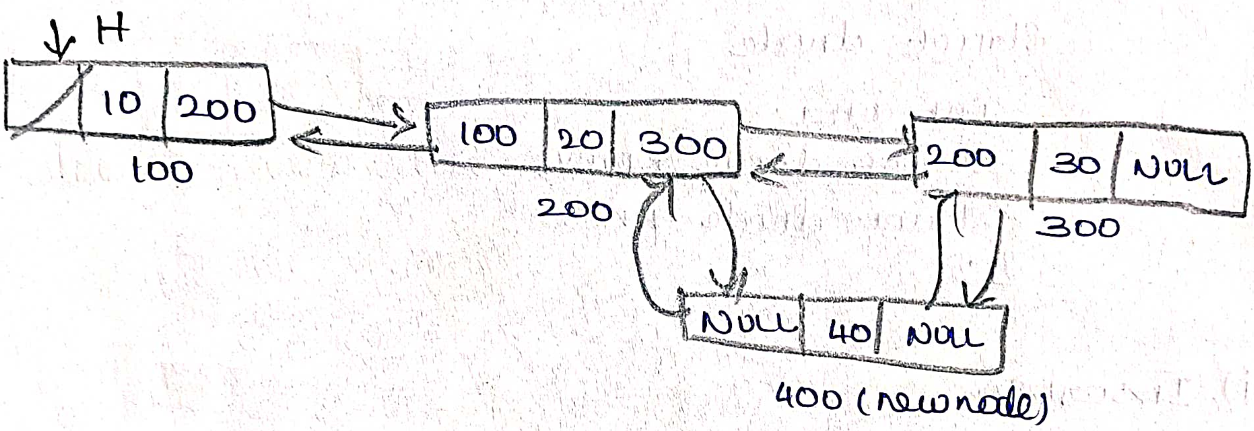
2)Insert a node at end:

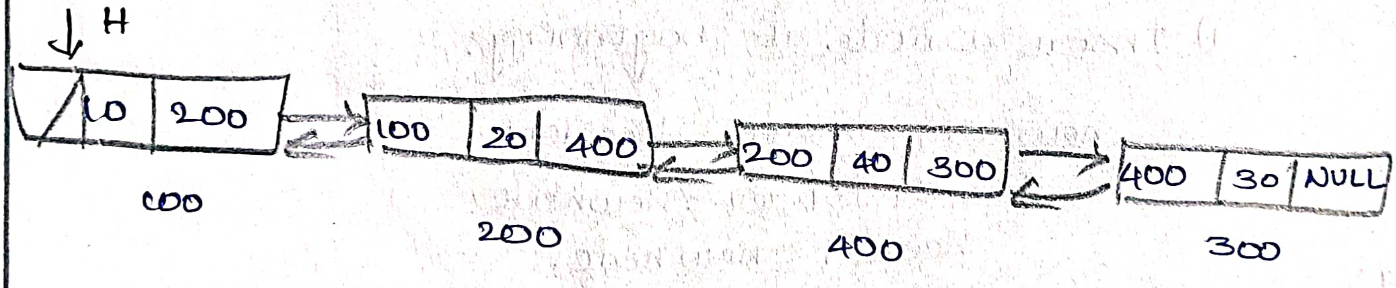
```
temp =heada
while (temp → next != NULL)
    temp = temp → next;
    temp → next = new node
    newnode → prev = temp.
```





3) Insert a node at middle:



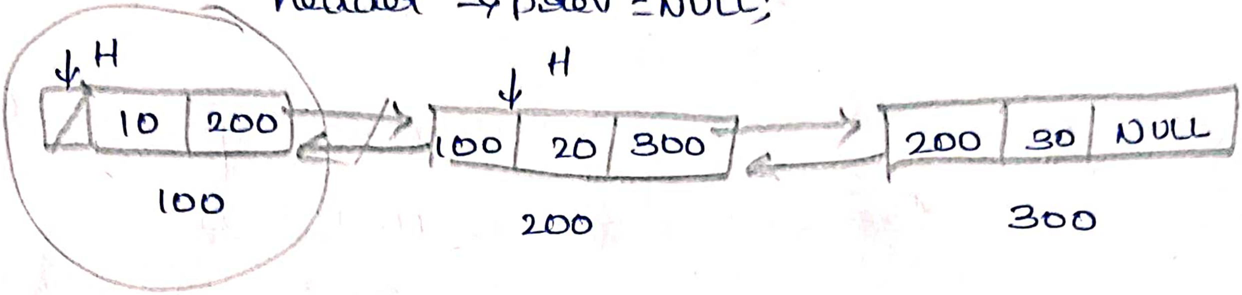


```
new node → next = p → next
    p → next → prev = new node
        p → next = new node
        newnode → prev = p
```

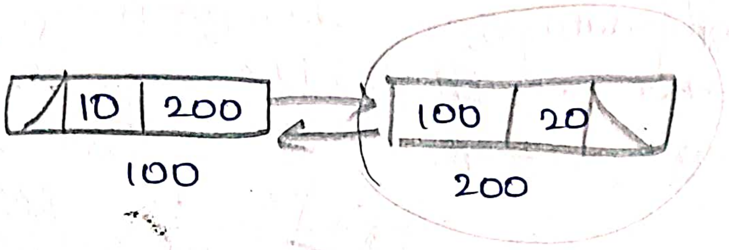
4) Deletion:

i) Delete the first node:

```
temp = header
header = header → next
header → prev = NULL;
```

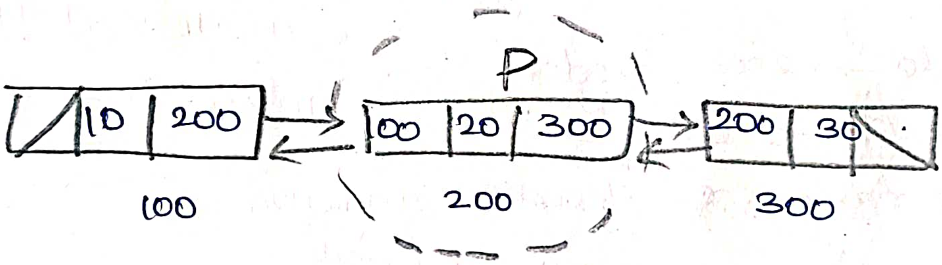


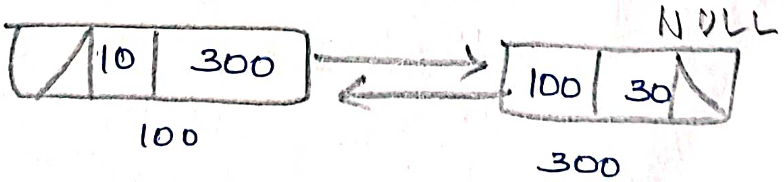
2) Delete a node at end:



3) Delete Intermediate node:

```
Find position p
p → prev → next = p → next
p → next → prev = p → prev
free (P);
```



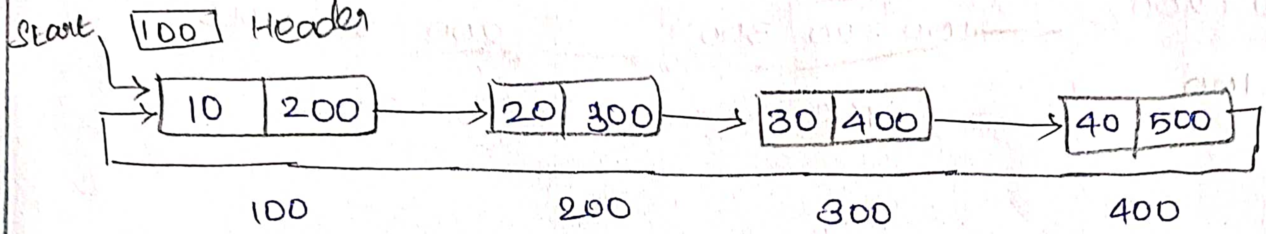


6) Circularly linked list [A/M - 2020]

Ins

The link field of the last node is made to Poin the start/first node of the list. A circular linked list has no beginning and no end.

Eg:

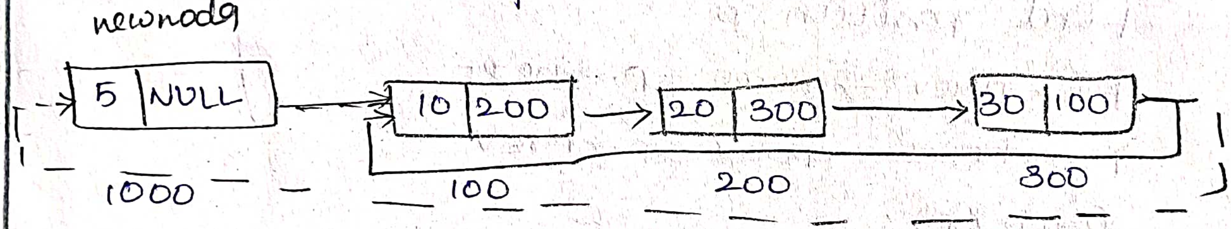


```
P → data=200
temp → data=10;
P → next = temp;
temp → next =P;
```

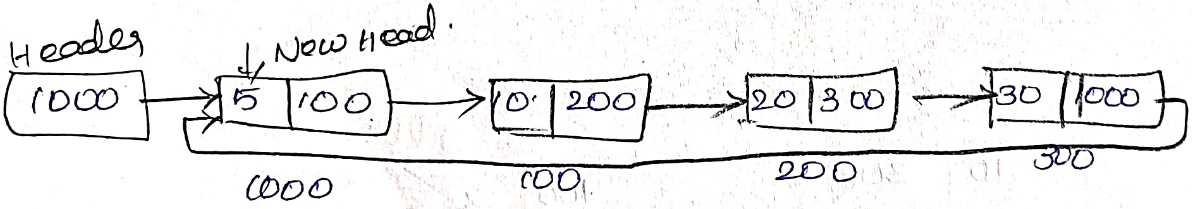
Operations:

i) First Insert:

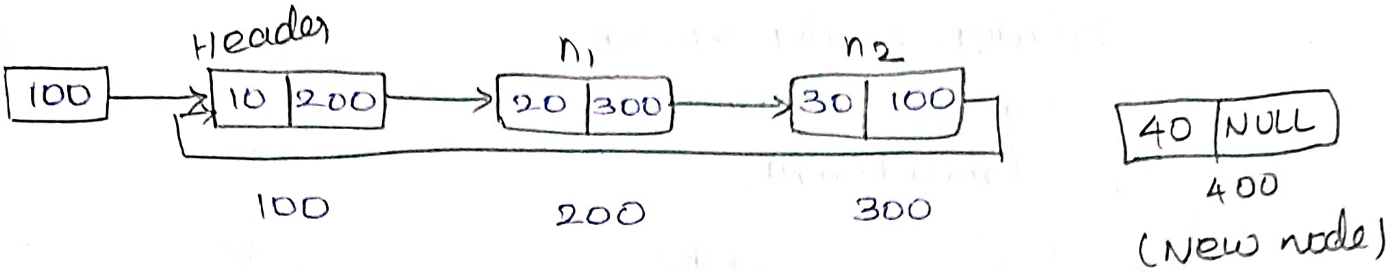
using search() function we can insert at first



```
newnode → next = header
header = newnode
temp → next = header.
```

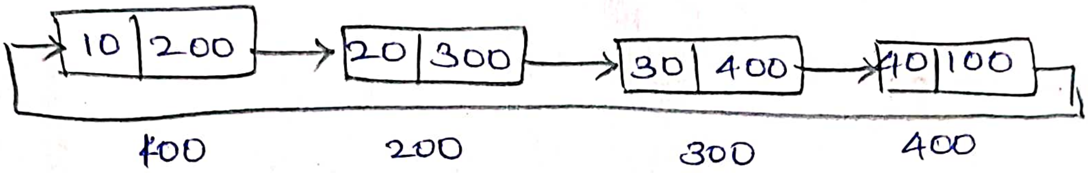


Insert at Last position:

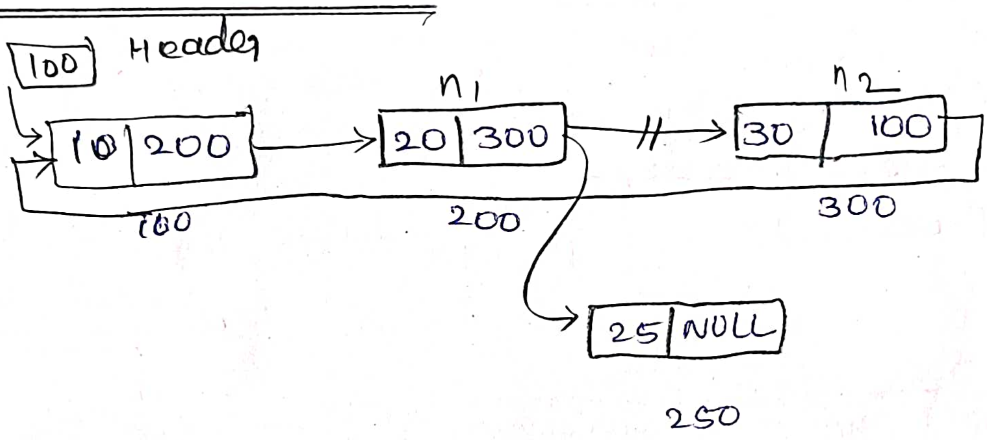


n2 → next=newnode

newnode → next = header.

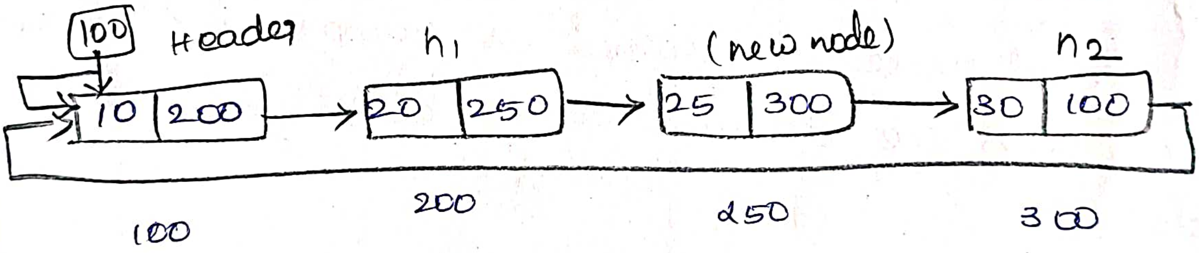


Insert at middle:



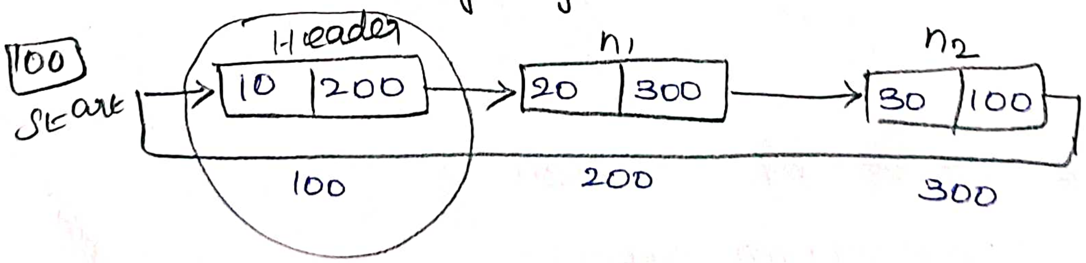
n → next = newnode

newnode → next = n2

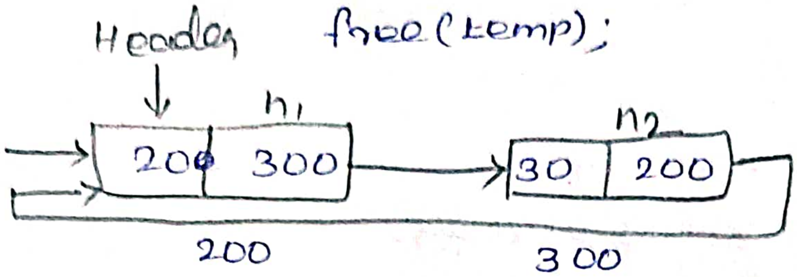


Deletion:

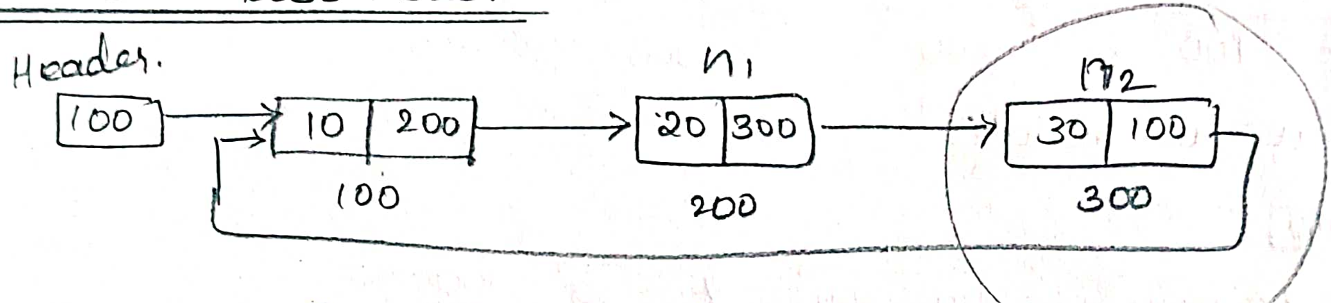
Deletion at Begining:



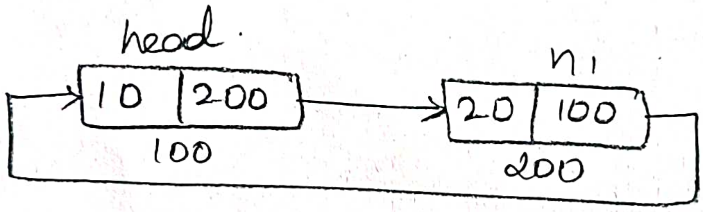
```
temp = header;
header = header -> next
n2 -> next = header
free(temp);
```



Delete at last node:

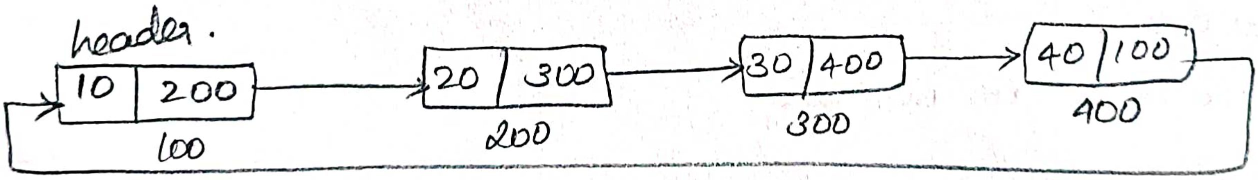


```
n1 -> next = header
free (n2);
```



Display of circular list:

Head node is assigned as temp node. If we have a CLL then the data will be displayed as temp -> data



```
temp -> next ! = header
temp = temp -> next.
```

8) Applications of list [N/D-2021. A/M-2022]

Polynomial Manipulation:

Polynomial expressions contains terms with non-zero, co-effients and exponents:

$$p(x) = a_0 x^n + a_1 x^{n-1} + \dots + a_n.$$

Each node contains three fields namely.

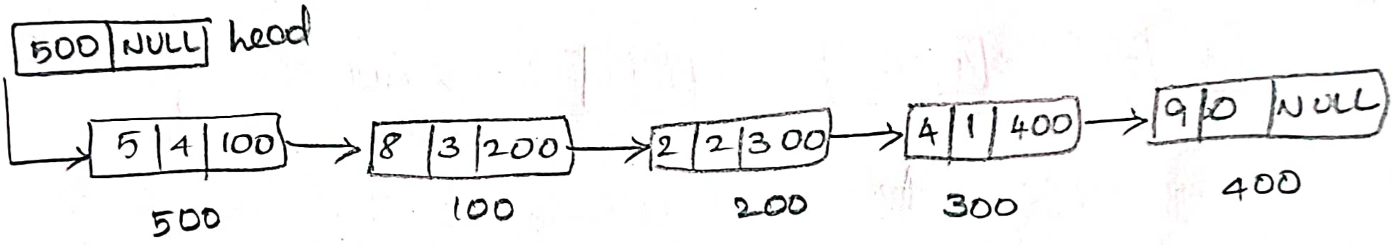
(*) Co-efficeint field

(*) Exponent field

(*) Link-field.

Co-efficient	Exponent	Link

$5x^4 - 8x^3 + 2x^2 + 4x + 9$



Polynomial ADT:

```
Struct poly
{
  int coeff;
   int exp;
  struct poly *next;
 };
```

Creation of polynomial:

```
poly create (poly *head1, poly *~~next~~node1)
{
  poly *ptr;
  if (head1 = =NULL)
```

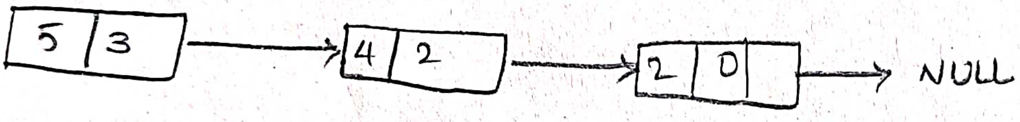
```
    { head1 = newnode1;
      return (head1);
    }
else
    { ptr = head1;
      while (ptr → next != NULL)
        ptr = ptr → next;
      ptr → next = newnode1;
    }
   return (head1);
  }
```

Eg: $5x^3 + 4x^2 + 2x^0$

$5x^1 + 5x^0$

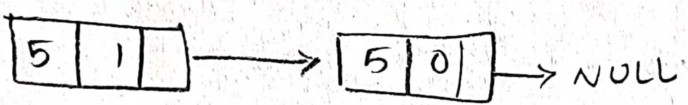
O/p − $5x^3 + 4x^2 + 5x^1 + 7x^0$

List 1:

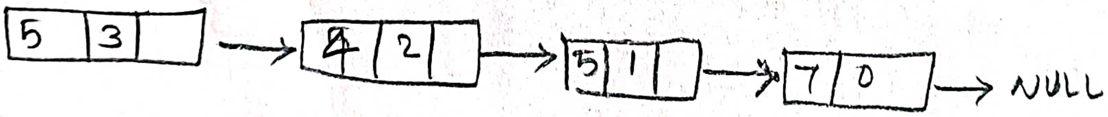


+

List2:



Resultant-list:

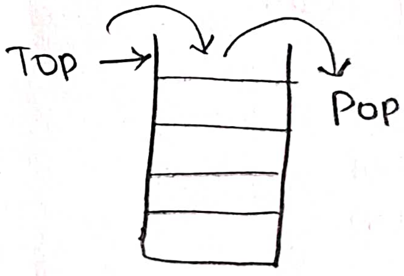


Linear Data Structures, Stack, Queue

Stack : [N/D - 2021]

A stack is a non-primitive linear DS and is an ordered collection of homogeneous data elements.

The last element inserted will be on top of the stack, since deletion is done from the same end. LIFO.



Stack:

Operations:

① PUSH,
② POP

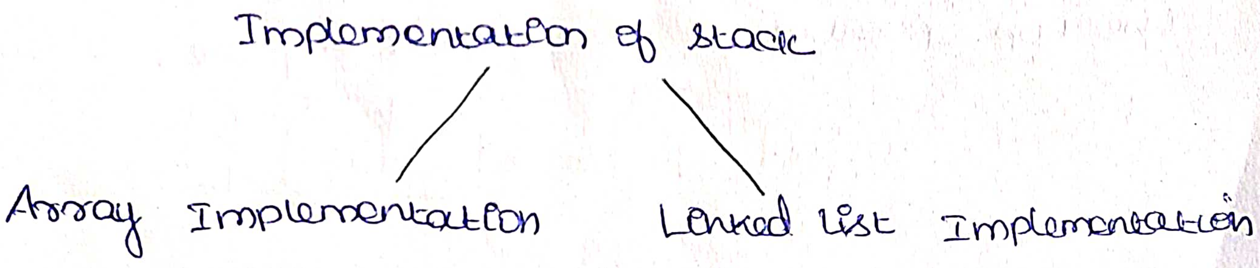
Push / Insertion operation:

This operations insert an element always on top of the stack. TOP = top + 1

Delete / Pop operation:

Deleting an element from the top of the stack is called pop operation. The pop operation can be decremented by 1.

top = top - 1



2) Operations :

1) Push()
2) Pop()
3) Isfull()
4) Is Empty()

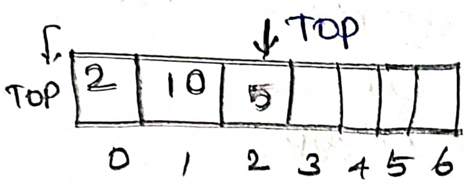
Push operation :

It adds a new element to the stack.

Each time a new element is inserted on the sta

the top pointer is incremented by one.

Eg:



push(2)
Push(10)
Push(5)

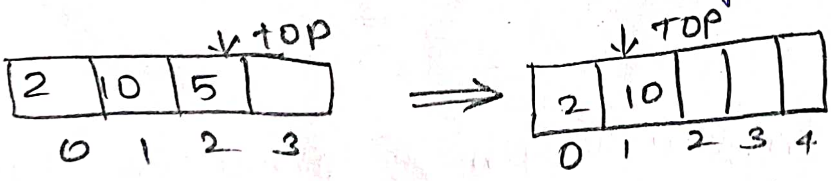
top = top +1

```
int isfull (stack s)
{ if (TOP = = Arraysize)
  return (1);
}
```

Pop operation :

A pop operation deletes the top most element from the stack.

Each time an element is removed from the stack, the top pointer is decremented by one.



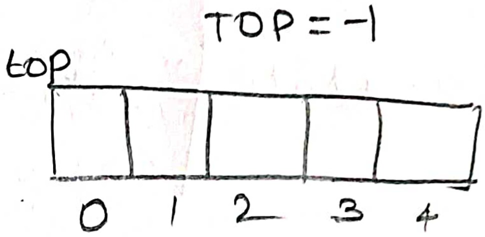
POP(5)

TOP = top -1

```
void pop (stack S)
{ if (IsEmpty(S))
    Error ("empty stack");
else
  { x = S[TOP];
    TOP = TOP -1;
  } }
```

Is Empty:

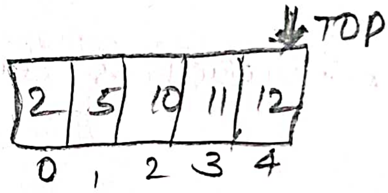
```
int IsEmpty (stack S)
  { if (TOP == -1)
      return (1);
  }
```



Empty stack

IsFULL:

```
int Isfull (stack S)
  { if (TOP == Arraysize)
      return (1);
  }
```



Full stack.

3) Applications of stack -(N/D-2018)

(or)

Evaluating Arithmetic Expression.

[Conversions of Infix to Postfix Expressions]

Applications:

(*) Expression Evaluation

(*) Backtracking (Game playing, Finding path)

(*) Memory Management.

Evaluating arithmetic expressions: (N/D-2018)

Infix	Prefix	Postfix.
a+b	+ab	ab+
a+b*c	+a *bc	abc*+
(a+b)*(c-d)	*+ab -cd	ab+cd-*
b*b-4*a*c	-*bb **4ac	bb*4a*c*-

Infix: operators are written b/w the operands they on, Eg: 3+4

Prefix: operators are written before the operands, Eg: +34

Postfix: operators are written after the operands. Eg: 34+

Conversions of Infix to postfix Expressions:

Steps:

1) first, we have to take infix expression.

2) we read the expression from left to right.

3) we read this expression one by one and check whether it is operand or operator.

4) If it is operand, then we print it and if operator we store it onto the stack.

5) In the end, we retrieve operator from the stack and print it.

f:

Infix	postfix
1) 2+3*4	234*+
a*b+5	ab*5+

5+6/3*(5+6)-7.

Eg:

Stack	O/P	
+	2	
+	2	Read '+' and placed on stack.
* +	23	Now read 3 and placed on o/p
* +	234*+	Now 4 is read & placed on o/p

1) (a+b) * c/d + e/f

Step 1:

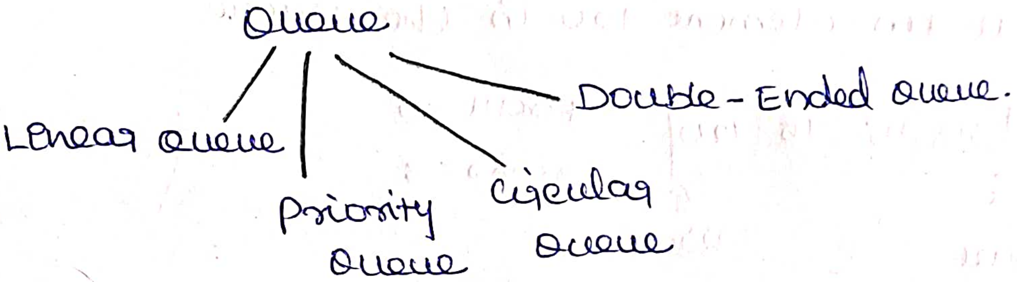
I/P stack	process	O/P stack
(	The left paranthesis is encountered, then it is pushed on to the stack	
(	'a' is read, so it is placed on to the o/p	a
+ (	'+' is read, then push it on to the stack	a

+ c	'b' is read so it is placed on to the o/p	ab
) + (	Symbol ')' is read, now we pop the operators	ab+
*	The operator '*' is read & place them is field	ab+
*	the operand 'c' is read & placed on the o/p	ab+c
/ *	'/' is read & placed on to the stack	ab+c
/ *	'd' is read & placed on the o/p	ab+cd
+ / *	'+' is scanned & its an operator, so it check. precedence inside stack.	ab+cd/*
/ +	'/' is read & placed on to the o/p	ab+cd/*e
/ +	'f' is read & placed on to the o/p	ab+cd/*ef
	The operator is poped & placed on to the stack	ab+cd/*ef/+

Queue ADT

A Queue is an ordered collection of elements in which insertion are made at one end is referred to as the "REAR end", and end from which deletions are made is referred to as the front end.

Queue is a FIFO - First in First out lists.



Implementation of queues:

① Array based Implementation.

② Linked list Implementation.

Operations of queue:

① Enqueue

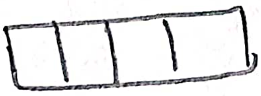
② Dequeue

Array Implementation of Queue:

Enqueue operation:

It is used to add new element into a queue, at the rear end.

Assign the new element in the array by incrementing the rear.

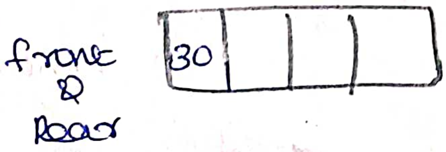


Empty queue array size 4

rear = -1

front = 0

Insert an element 30 in the queue

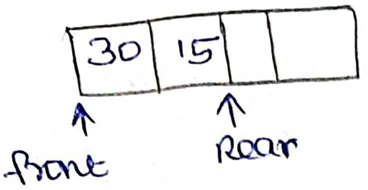


front = 0

rear = 0

rear = rear + 1

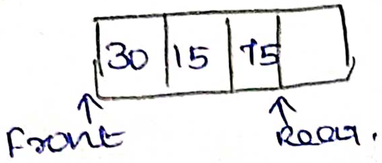
Insert an element 15 in the queue.



front =0

rear =1

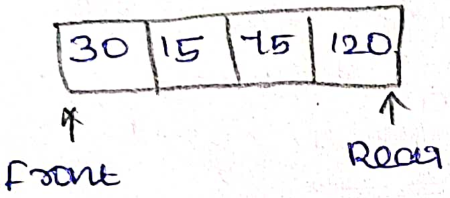
Insert an element 75 in the queue



front =0

rear =2

Insert an element 120 in the queue



front =0

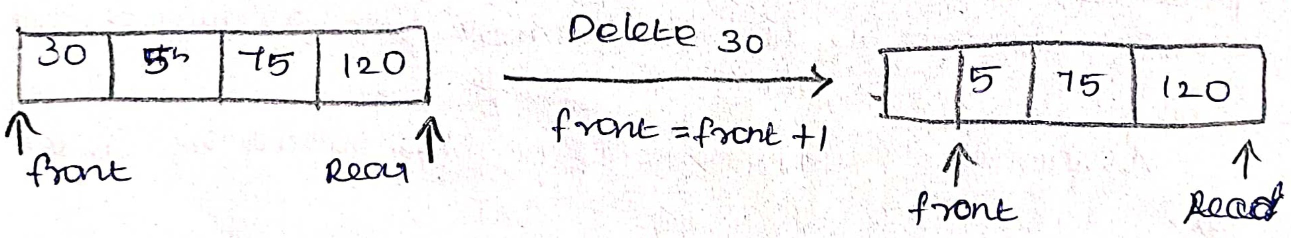
rear =3

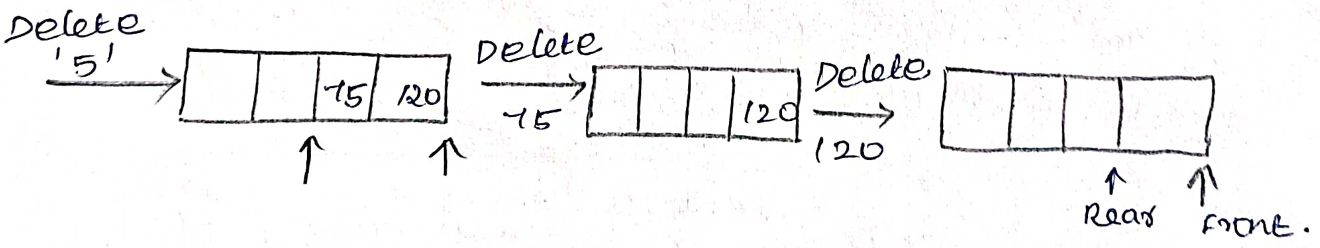
Routine

Dequeue operation:

It is used to delete an element from the front end of the queue. When Implementing the dequeue operation underflow condition of a queue, is to be checked.

Increment the variable by 1.





Linked List Implementation of queue.-[A/M-2022]

A queue can be implements by singly linked list.

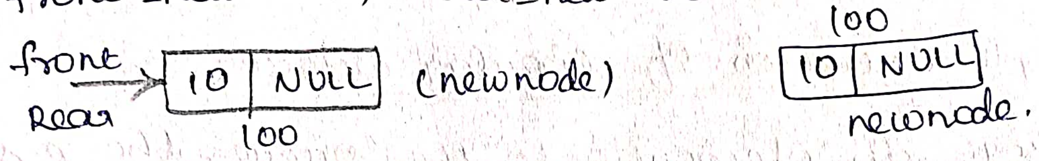
Enqueue operation:

It is used to add a new element onto a queue.

1) Allocate the memory for the newnode.
2) Assign the value for datapart of the newnode.
3) Assign the link of a rear to the newnode.
4) Assign the rear to the newnode.

rear = 0 }
front = 0 } Queue is Empty.

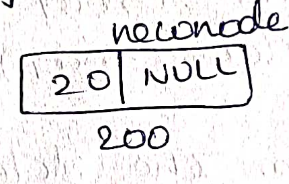
front = newnode ; rear = newnode.

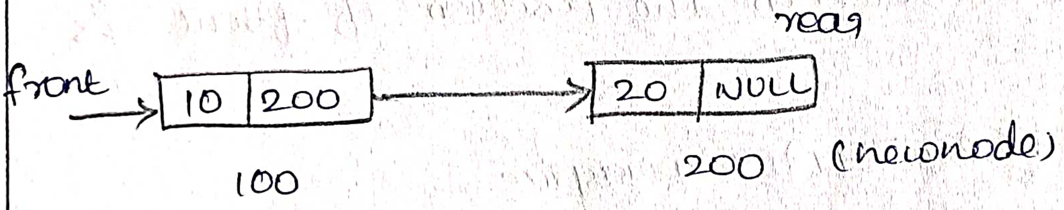


It represents queue having a single node.

Insert a newnode to the existing node.

rear → next = newnode.
rear = newnode.





Dequeue operation:

It is used to remove an element from the front of the queue.

1) Assign the front pointer to the temp pointer.
2) The front pointer is made to point after the first node, & the other nodes remains unchanged.
3) Free the allocated mry of the temp pointer.

Applications of queue : [N/D - 2021]

⇒ When data is transferred asynchronously b/w to proc

eg: IO Buffers.

⇒ When resource is shared among multiple consumers.

Eg: Include CPU and Disk scheduling.

(*) In recognizing palindrome.

(*) keyboard buffer.

(*) Job scheduling

(*) Round Robin scheduling

(*) simulation.

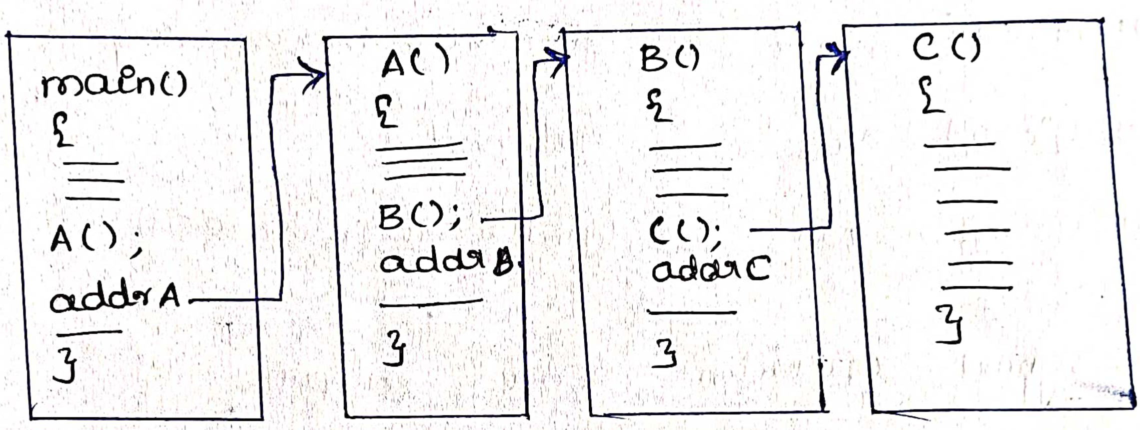
Applications of stack:-

Function Calls:

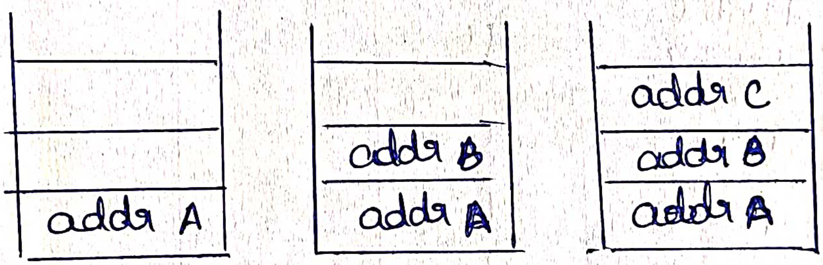
Function call is a dynamic data structure where elements are stored at contiguous memory locations.

Function call stack is maintained for every function call where it contains its own local variables and parameters of the callee function.

Eg: Three program with A, B, C functions.



Function calls.



when fun A is called

when fun B is called

When fun C is called

1) Balanceing Symbols:

Each time parser reads one character at a time. If the character is opening delimiter like '(', '{', '[' then pushed on to the stack

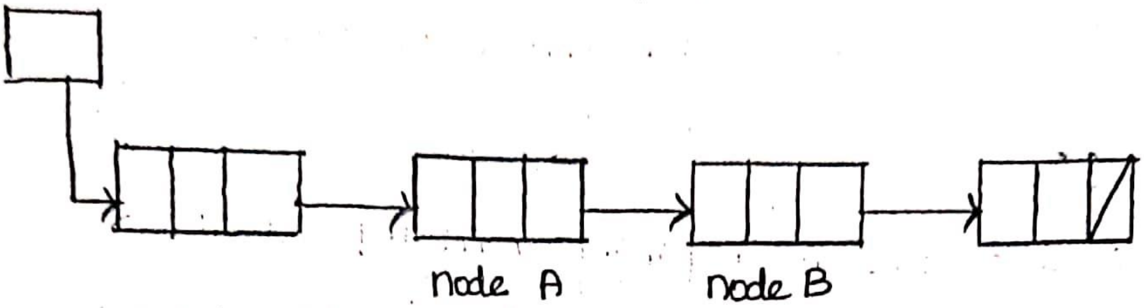
If the character is closing delimiter like ')', '}', ']' then POPED on to the stack.

Example	Vaid?	Description
(A+B)+(C+D)	Yes	Having balanced symbol
((A+B)+(C-D)	NO	one closing brace missing
((A+B)+[C-D])	Yes	Balanced
((A+B)+[C-D]]	NO	Unbalanced (mis match)

Input symbol	Operation	Stack	output
(	Push (	(	
)	POP (		
(	Push (	(	
(	push (	((	
)	POP (	(	
[	push [	([	
(	Push (	([(	
)	POP)	([	
]	POP [	(	
)	POP (		
	Stack is Empty		True

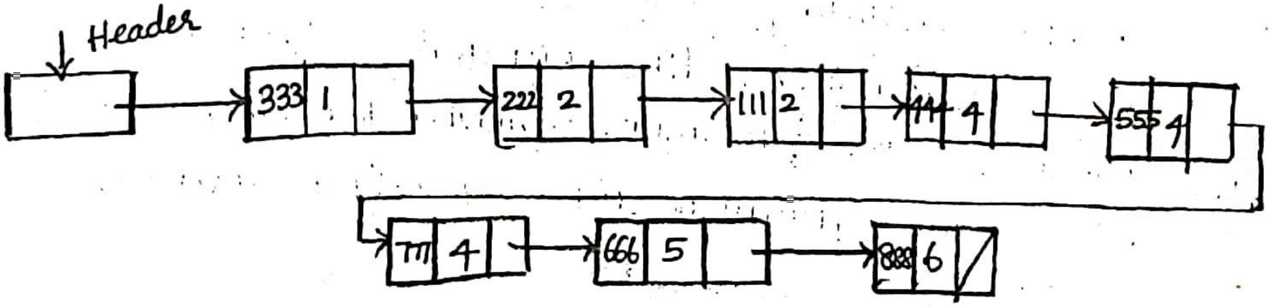
Linked List representation of Priority Queue.

Information	Priority number	Address of next node



Priority of node A is higher than priority of node B. (or) Both have same Priority → node A was added before node B.

Location	Info	Priority	Link
1	222	2	
2			
3	444	4	
4	555	4	
5	333	1	
6	111	2	
7			
8	666	5	
9	777	4	
10	888	6	
11			



UNIT-III

NON LINEAR DATA STRUCTURES - TREES.

1) Tree ADT
2) Tree Traversals
3) Binary Tree ADT
4) Expression Trees
5) Applications of Trees
6) Binary Search Tree ADT
7) Threaded Binary Trees
8) AVL Trees
9) B-Tree
10) B^+ Tree -
11) Heap.
12) Applications of Heap.

A tree is a data structure made up of nodes (or) Vertices and edges without having any cycle. The tree with no nodes is called the null (or) empty tree. A tree that is not empty consists of a root node and potentially many levels of additional nodes that form a hierarchy.

Terminologies Used in Tree :

Root :

The top node in a tree. (A)

Child :

A node directly connected to another node when moving away from the root. Except Root node all are child node.

Parent :

The converse notation of a child. ie., parent node that has an edge to a child node. (A, B, C)

Siblings :

A group of nodes with the same parent. (D and E are siblings for B)

Descendant : (children, grandchildren, greatgrand children)

A node reachable by repeated proceeding from parent to child. Also known as subchild. (lower hierarchy)

Ancestor : [Parent, Grandparent, Great-GP and so on]

A node reachable by repeated proceeding from child to parent. (higher hierarchy)

Leaf : (Terminal node) Eg; D, E, F, G

External node (not common). A node with no children.

Branch node (Internal node) : A node with atleast one child. Eg., Root node. (Non-Terminal) nodes. Eg; A, B, C

Degree: for a given node, its number of children. A leaf is necessarily degree zero. Eg; Degree (A) = 2.

Edge : The connection between one node and another.

Path : A sequence of nodes and edges connecting a node with a descendant.

Level : The level of a node is defined as : 1 + the number of edges between the node and the root. Level = Depth + 1.

Level of Root = 0.

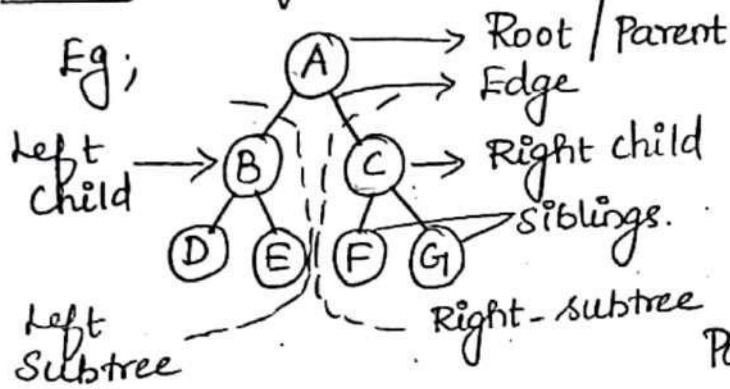
__Height of tree__: The height of a tree is the (N/D-2018) height of its root node. [Starts from 0]. Leaf don't have height.

__Height of node__: The height of a node is the number of edges on the longest path between that node and a leaf.

__Depth__: The depth of a node is the number of edges from the tree's root node to the node. [Starts from 0]

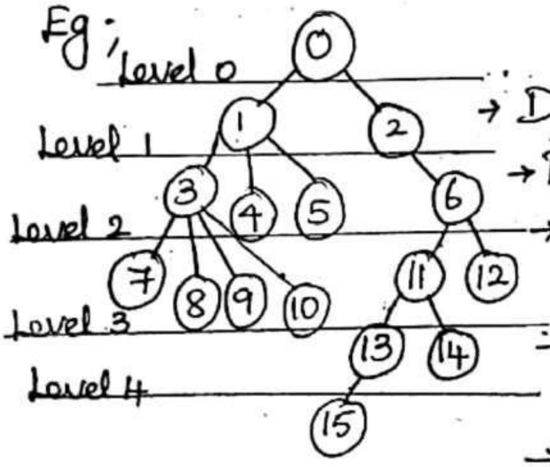
__Forest__ : A forest is a set of n ≥ 0 __disjoint trees__.

Eg;



Depth of node C : 01
Height of tree : 2.
Height of Root : ~~3~~ 2
Height of node C : 01
Depth of tree : 2.
Depth of Root : 0.
Level : 3 levels. (1&2)
Path from A to F : A–C–F

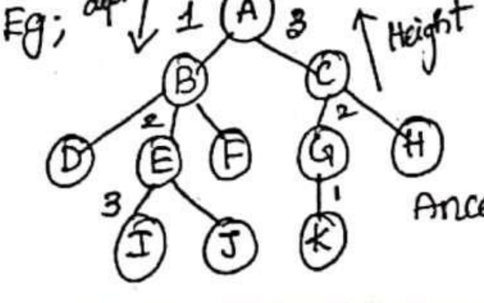
Eg;



→ It has 16 nodes.
→ Depth : 5
→ Degree of Tree :- 5
→ Degree of node 0 :- 2
→ Root : Node 0
→ Node 4 is a leaf and Node 4 is the child of 1.
→ Root node 0 is the Grandparent of node 4.
→ Nodes 3, 4 and 5 are siblings. since it has same parent 1.
→ Height of tree : 6 5

In a tree with 'N' nodes, there will be maximum 'N–1' edges.

Eg;



Depth of tree : 3
Depth of A : 0
Depth of B : 1
Depth of K : 3

Height of tree :- 3.
Height of A : 3
Height of B : 2
Height of K : 0

Ancestor :- A is ancestor for all nodes.

(2) Tree Traversals.

Tree traversal (also known as tree search) is a form of Graph traversal and refers to the process of visiting (checking and/or updating) each node in a tree data structure, exactly once. Such traversals are classified by the order in which the nodes are visited.

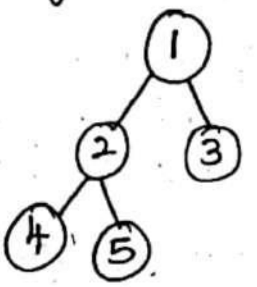
- Inorder
- Preorder &
- Postorder.

(N/D-2018)

Unlike linear data structures (Array, Linked List, Stacks, Queues, etc) which have only one logical way to traverse them, trees can be traversed in different ways.

DFS: Depth First Search.

Eg.,



Inorder (Left, Root, Right):
42513

Preorder (Root, Left, Right):
12453

Postorder (Left, Right, Root):
45231

BFS: (Breadth first Search)

1 2 3 4 5.

Inorder:

1) Traverse the left subtree.
2) Visit the root.
3) Traverse the right subtree.

```
Void inorder (Tree T)
{
 if (T!=NULL)
 {
 inorder (T->left);
 PrintElement (T->Element);
 inorder (T->right);
 }
}
```

Preorder :

1) visit the root
2) Traverse the left subtree
3) Traverse the right subtree.

```
Void Preorder (Tree T)
{
   if (T! = NULL)
   {
    PrintElement (T->Element);
    Preorder (T->left);
    Preorder (T->right);
} }
```

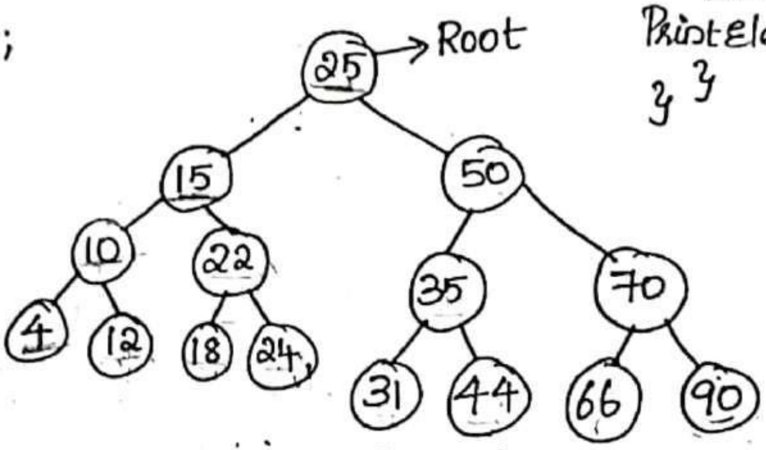
Postorder :

1) Traverse the left subtree
2) Traverse the right subtree
3) Visit the root.

```
   Void Postorder (Tree T)
{
    if (T! = NULL)
   {
    Postorder (T->left);
    Postorder (T->right);
   PrintElement (T-> element);
} }
```

L-R-Ri
R-L-Ri
L-Ri-R

Eg;



Inorder : 4-10-12-15-18-22-24-25-31-35-44-50-66-70-90

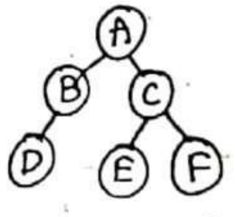
Preorder: 25-15-10-4-12-22-18-24-50-35-31-44-70-66-90

Postorder: 4-12-10-18-24-22-15-31-44-35-66-90-70-50-25

(N/D-2018)

BFS - Breadth first Search.

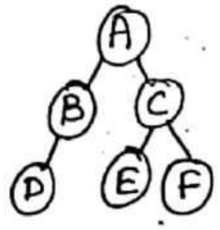
* It uses the Queue for storing the nodes.
* Constructs wide and Short tree.
* Vertex-Based algorithm.



BFS : A-B-C-D-E-F

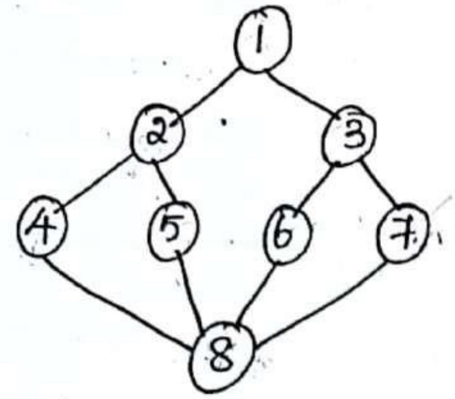
DFS - Depth first Search.

* It uses the stack for traversal of the nodes.
* constructs narrow and long trees.
* Edge-based algorithm.



DFS : A-B-D-C-E-F.

Eg;



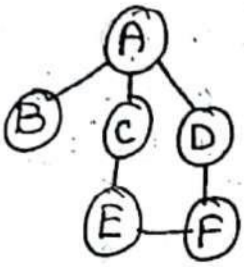
(we all connections at one time)

BFS: 1-2-3-4-5-6-7-8

(we use only one connection at a time)

DFS: 1-2-4-8-5-6-7-3

Eg;



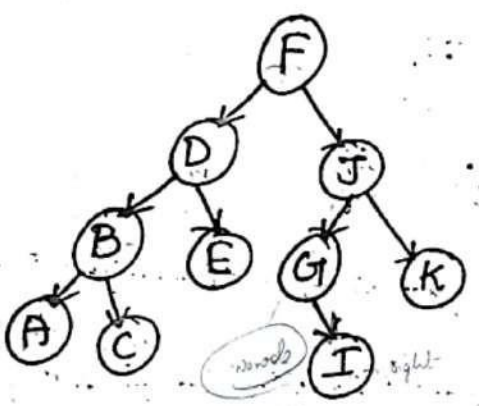
BFS: A B C D E F

DFS: A B C E F D (or)

A D F E C B (or)

A C E F D B

Eg;



L R Ro

Inorder: ABCDEFGIJK

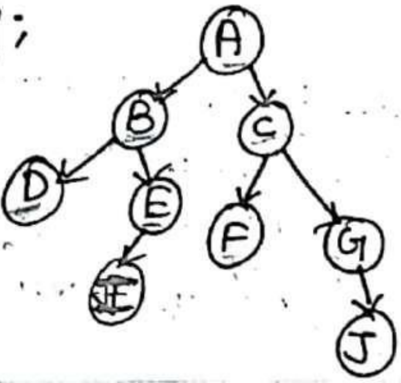
R L Ri

Preorder: FDBACEJGIK

L Ri R

Postorder: ACBEDIGKJF

Eg;

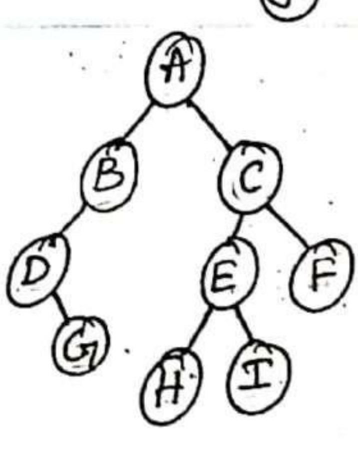


Inorder: DBIEAFCGJ

Preorder: ABDEICFGJ

Postorder: DIEBFJGCA

Eg;



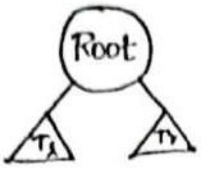
Inorder: DGBAHEICF

Preorder: ABDGCEHIF

Postorder: GDBHIEFCA

(3) BINARY TREE ADT

A Binary tree is a tree in which no node can have more than two children. The maximum degree of any node is two. This means the degree of a binary tree is either zero (or) one (or) two.



T_l — Left Tree
T_r - Right Tree.

In the above fig., the binary tree consists of a root and two sub trees T_l and T_r. All nodes to the left of the binary tree are referred as left subtrees and all nodes to the right of a binary tree are referred to as right subtree.

Implementation:

A Binary tree has atmost two children, we can keep direct pointers to them. The declaration of tree nodes is similar in structure to that for doubly linked lists, in that a node is a structure consisting of the key information plus two pointers (left and right) to other nodes.

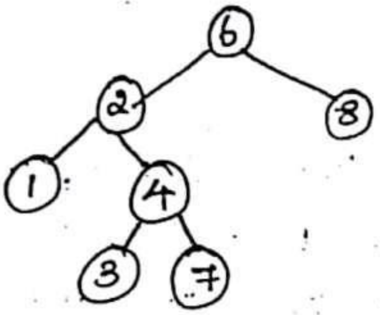
Binary tree node declaration:

```
typedef struct tree_node *tree_ptr;
struct tree_node                              Struct node
{                                             {
  element_type element;          (or)           int data;
  tree_ptr left;                              Struct node *left;
  tree_ptr right;                             Struct node *right;
};                                            };
typedef tree_ptr Tree;
```

Types of Binary Tree :

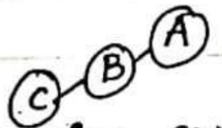
* Strictly binary tree.
* Skew tree
* Left Skewed binary tree.
* Right Skewed binary tree.
* Fully Binary tree (or) Proper binary tree
* Complete Binary tree.
* Almost Complete Binary tree.

*** Strictly Binary tree :** It is a Binary tree where all the nodes will have either zero (or) two children. It does not have one child in any node.

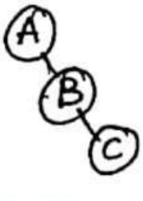


*** Skew tree :** It is a binary tree in which every node except the leaf has only one child node. There are two types of skew tree, they are left skewed binary tree and right skewed binary tree.

*** Left Skewed Binary tree :** A left skew tree has node with only the left child. It is a binary tree with only left subtrees.



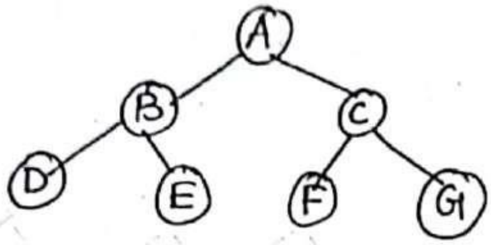
*** Right Skewed Binary tree :** A right skew tree has node with only the right child. It is a Binary tree with only right subtrees.



*** Full Binary tree (or) Proper Binary tree :**

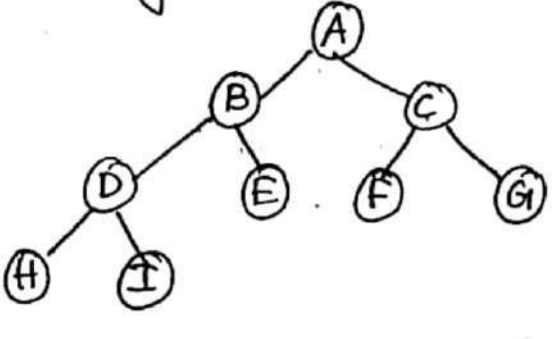
A Binary tree is a full binary tree if all leaves are at the same level and every non leaf node has exactly two children and it should contain maximum possible number of nodes in all levels. *A full Binary tree of height h has $2h+1-1$ nodes.

$\boxed{2^{h+1}-1}$

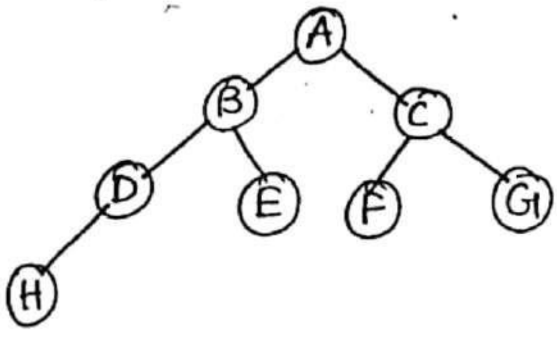


$2_{h+1}-1$ nodes.

*** Complete Binary tree:** Every non-leaf node has exactly two children but all leaves are not necessary at the samelevel. A complete Binary tree is one where all levels have the maximum number of nodes except the last level. The last level elements should be filled from left to right.



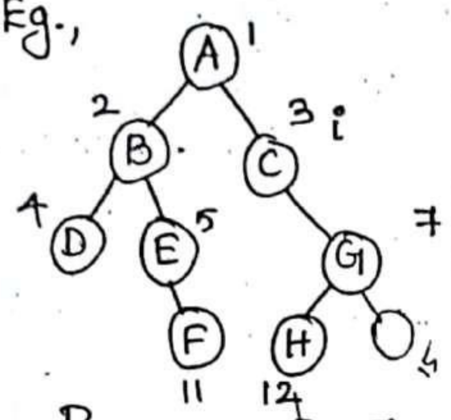
*** Almost Complete Binary tree :** An almost complete Binary tree is a tree in which each node that has a right child also has a left child. Having a left child does not require a node to have a right child.



Implementation of Binary tree : (In memory)

- Array Implementation
- Linked list Implementation.

Eg.,



Using Array.

7

0	A	B	C	D	E		G				F	H
0	1	2	3	4	5	6	7	8	9	10	11	12

Parent :- $\left[i/2 \right] \Rightarrow \lfloor i/2 \rfloor \Rightarrow \lfloor 3/2 \rfloor \Rightarrow \lfloor 1.5 \rfloor \Rightarrow 1$

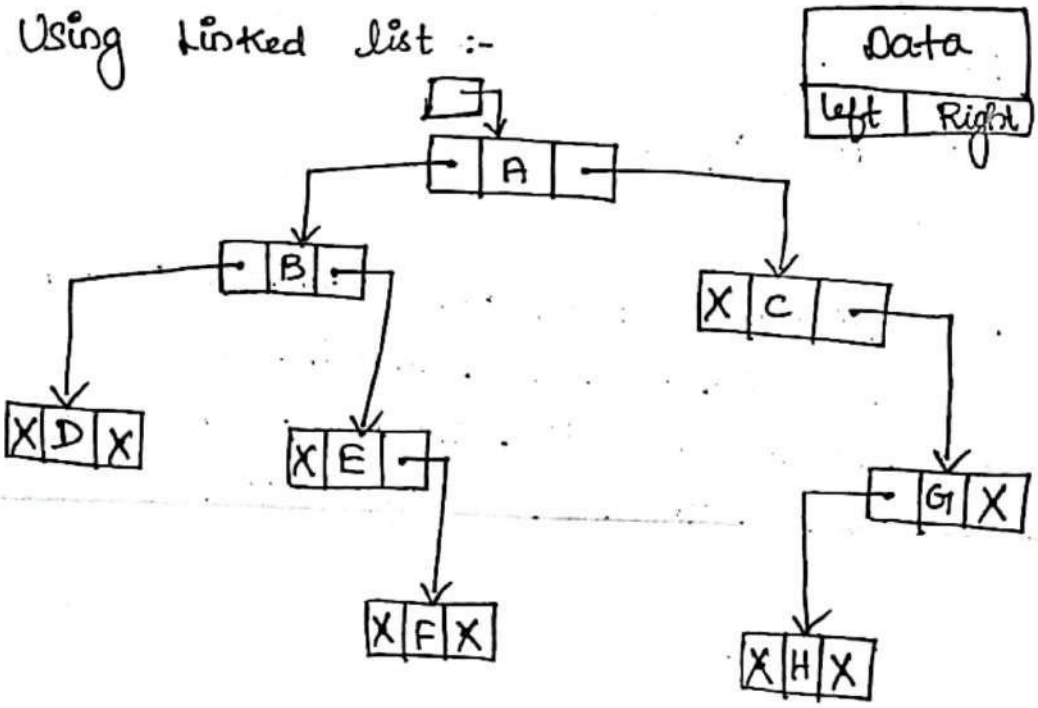
If c is parent for eg., then,

leftchild = $2 * i \Rightarrow 2 * 3 \Rightarrow 6$

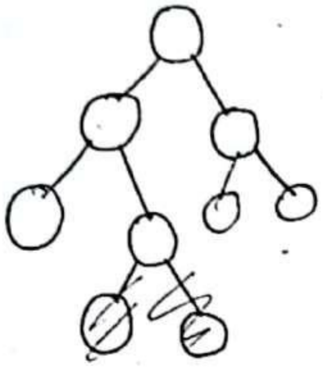
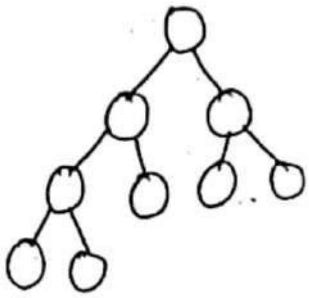
Right child = $2 * i+1 \Rightarrow 2 * 3+1 \Rightarrow 6+1 \Rightarrow 7$

So if parent is c, then leftchild is at 6th Position and right child is at 7th position.

Using Linked list :-



Left child Address	Data	Right Child Address

Full Binary tree	Complete Binary tree
* A full binary tree (Sometimes proper binary tree (or) 2-tree) is a tree in which every node other than the leaves has two children.	* A Complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.
	
* Each node has zero (or) two children.	* All levels of the tree is full, except possibly the last level, where nodes are filled from left to right.

(4) Expression Trees. N/D-2018

An expression tree is a representation of expressions arranged in a tree-like data structure. In other words, it is a tree with leaves as operands of the expressions and node contain the operators.

Similar to other data structures, data interaction is also possible in an expression tree.

Expression trees are mainly used for analyzing, evaluating and modifying expressions, especially complex expressions.

Types:

- * Algebraic expressions
- * Boolean expressions.

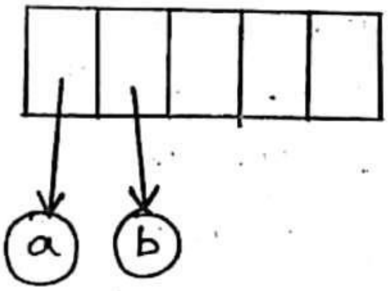
Construction of an expression tree.

The evaluation of the tree takes place by reading the postfix expression one symbol at a time.

* If the symbol is an operand, a one-node tree is created and its pointer is pushed onto a stack.

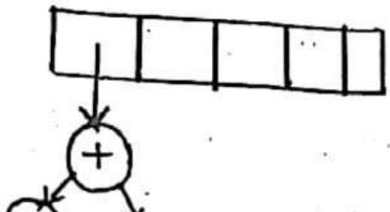
* If the symbol is an operator, the pointers to two trees T_1 and T_2 are popped from the stack and a new tree whose root is the operator and whose left and right children point to T_2 and T_1 respectively is formed. A pointer to this new tree is then pushed to the stack.

Example : The input is: ab+cde+**.
Since the first two symbols are operands, one-node trees are created and pointers are pushed to them onto a stack. for convenience the stack will grow from left to right.



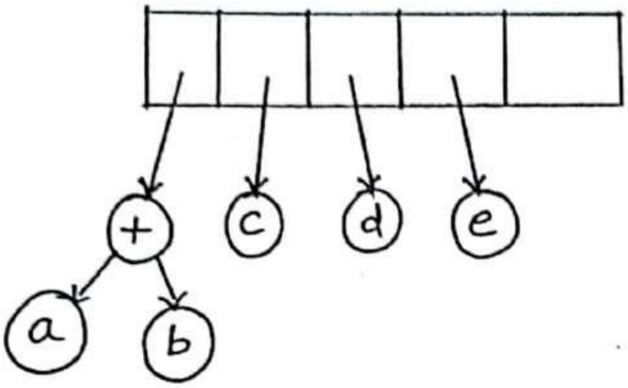
Stack Growing from Left to Right.

⇒ The next symbol is '+'. It pops the two pointers to the trees, a new tree is formed, and a Pointer to it is pushed onto the stack.



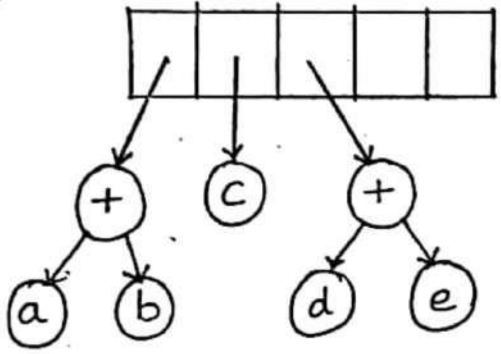
Formation of a new tree.

⇒ Next c, d and e are read. A one-node tree is created for each and a pointer to the Corresponding tree is pushed onto the stack.



Creating a one-node tree.

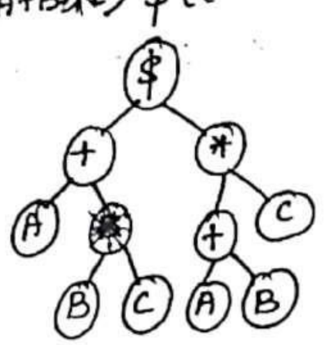
⇒ Continuing, a '+' is read, and it merges the last two trees.

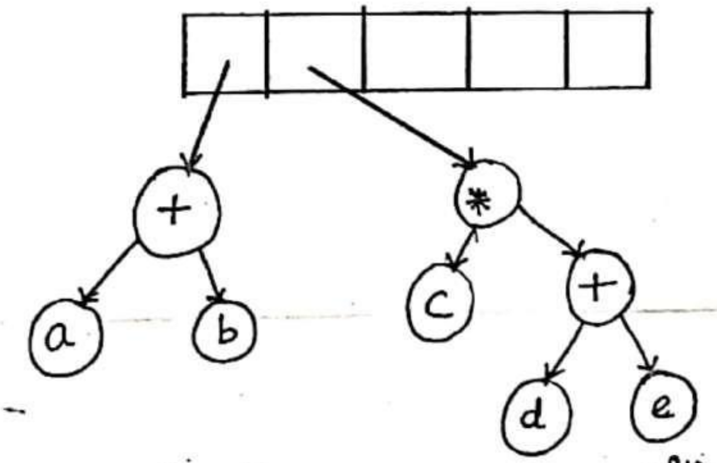


Merging two trees.

⇒ Now, a '*' is read, the last two tree pointers are popped and a new tree is formed with a '*' as the root.

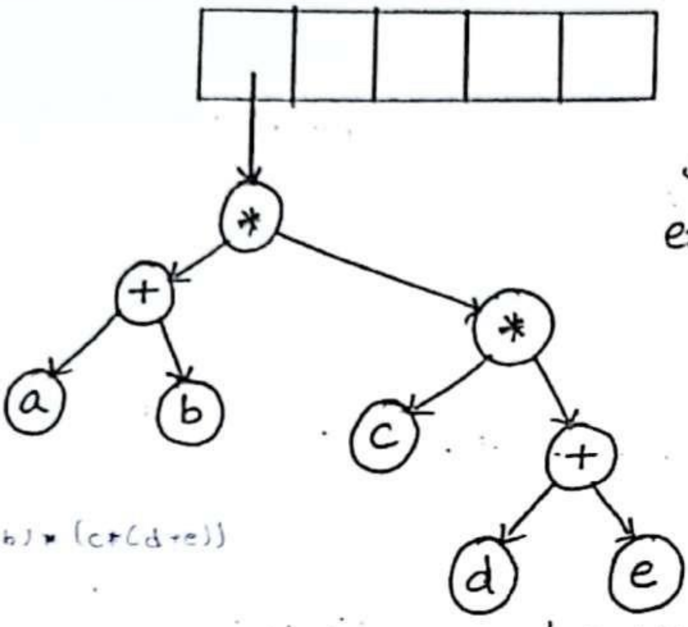
Eg (A+B*C) $ ((A+B)*C)





forming a new tree with a root.

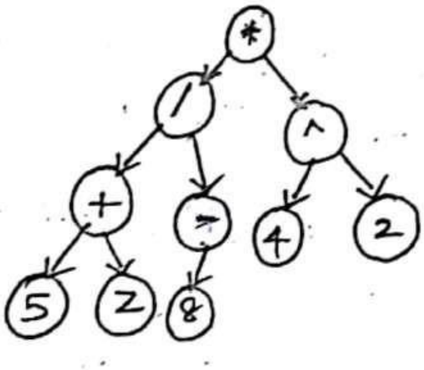
⇒ finally, the last symbol is read. The two trees are merged and a pointer to the final tree remains on the stack.



Steps to construct an expression tree ·
ab+cde+**.

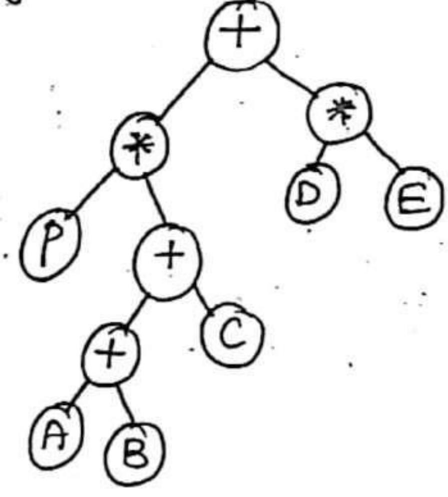
(a+b)*(c+(d+e))

Algebraic expressions :-
((5+z)/-8) * (4^2)

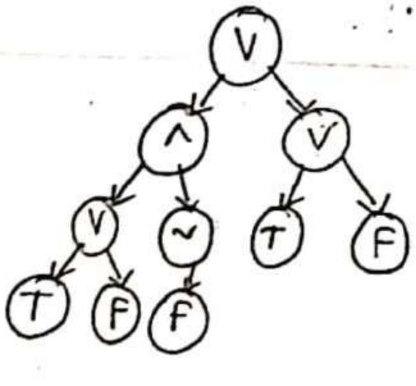


-8

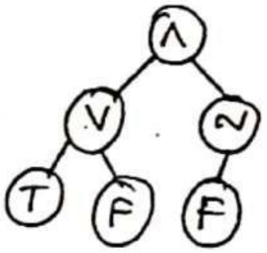
Eg(2)~~P*~~((A+B))+c) + (D*E)



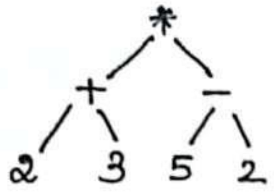
Boolean Expressions :-
((true V false) ∧ ~false) V (true v false))



Eg(2)
(true V false) ∧ ~ false

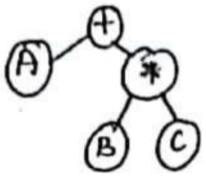


Eg (2) (2+3) * (5-2)

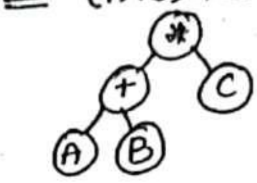


Note :- Internal node – operator
External node – operand

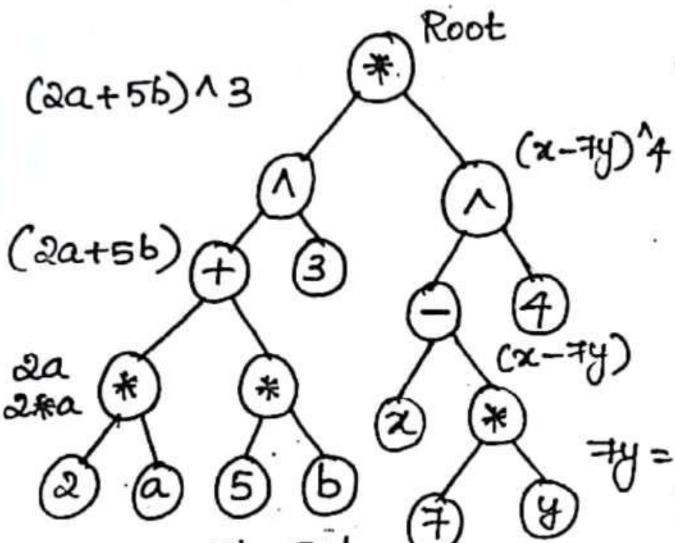
Eg A+B*C



Eg (A+B)*C



Eg(3) (2a + 5b)^3 * (x-7y)^4



7y = 7*y

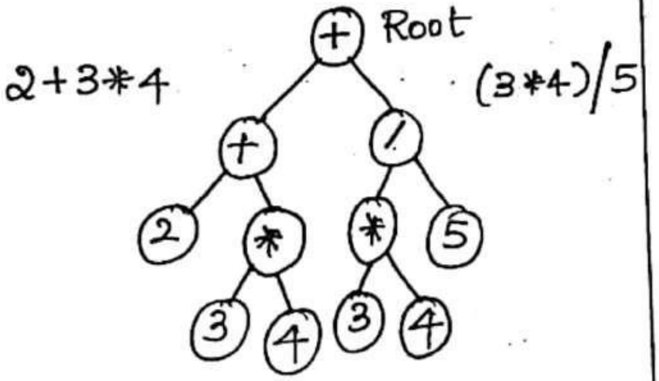
5b = 5*b

1) Divide expressions into two small mathematical term to get Root operator.

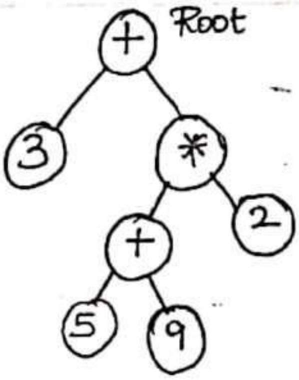
2) Scan symbol from right to left.

3) Place operator then expression on to the node.

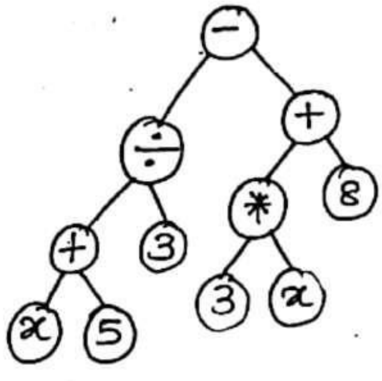
Eg (4) 2+3*4+(3*4)/5



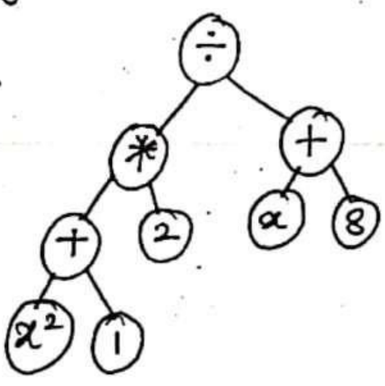
Eg(5) 3+ ((5+9) * 2)



Eg(6) ((x+5) ÷ 3) - (3x+8)



Eg(7) $((x^2+1)*2) \div (x+8)$

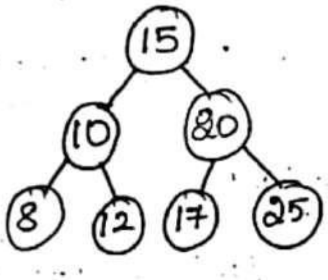


(5) Applications of trees.

- * Binary Search Trees / Binary Sorted trees.
- * Decision Trees.

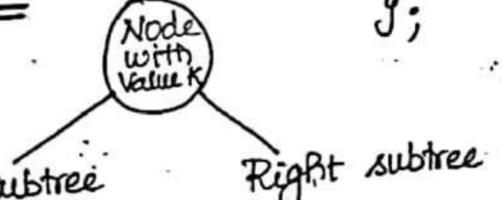
Binary Search Tree (BST) :- It is a binary tree where each node has a comparable key (and an associated value) and satisfies the restriction that the key in any node is larger than the keys in all nodes in that node's left subtree and smaller than the keys in all nodes in that node's right subtree.

Left subtree ≤ Right subtree.



```
Struct node
{
  int data;
Struct node *leftchild;
Struct node *rightchild;
};
```

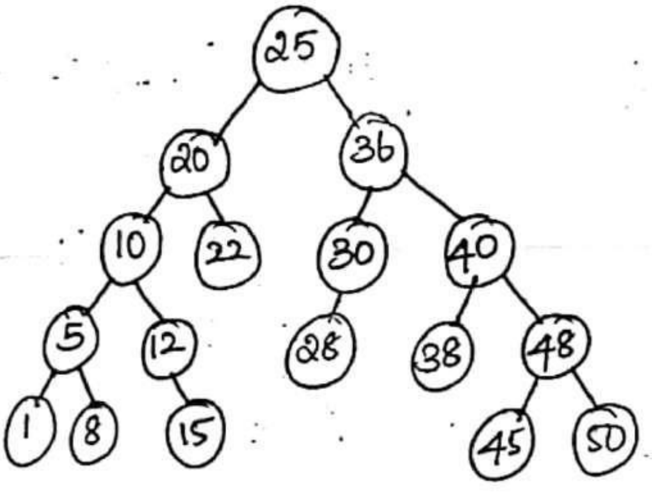
Representation:-



[Contains only smaller values] [Contains only larger values]

All Values <=K All Values > K.

Eg.,



Left Subtree (Keys) ≤ node (Key) ≤ right-subtree (Keys)

Operations on BST.

*Search *Insertion *Deletion.

SEARCH in BST: (N|D-2018)

Step 1 : Read the search element from the user.

Step 2 : Compare, the Search element with the value of Root node.

Step 3: If both matches, then display "Given node found".

Step 4 : If both not matches, Check whether search element is smaller (or) larger than that node value.

Step 5 : If larger, then continue Search process at right subtree

Step 6 : If smaller, then search at left subtree.

Step 7 : Repeat the same until we found exact element.

Step 8 : If we reach the node, then display "Element found", if not then display "Element not found".

INSERTION in BST:.

Step 1 : Create a Newnode with given value and set its left and right to NULL.

Step 2 : Check whether tree is empty.

Step 3 : If Empty, then set root to newnode.

Step 4 : If the tree is not empty, then check whether value of newnode is smaller (or) larger than the node. (here it is root node)

Step 5 : If Newnode is smaller (or) equal to root node then move to left else move to right child.

Step 6 : Repeat the above step until we reach to a leaf node.

Step 7: After reaching a leaf node, then insert the newnode as left child. if newnode is smaller (or) equal to that leaf, else insert it as right child.

DELETION IN BST :

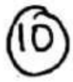
Step 1 : Find the node to be deleted using Search operation.

Step 2 : Delete the node using free function (if it is a leaf) and terminate the function.

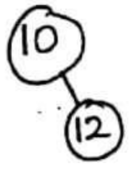
Insert the following into a Binary Search Trees.

10, 12,. 5, 4, 20, 8, 7, 15 and 13.

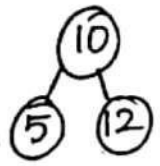
Insert 10.



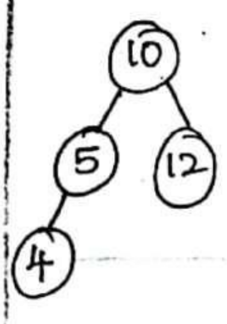
Insert 12



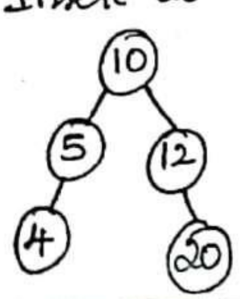
Insert 5



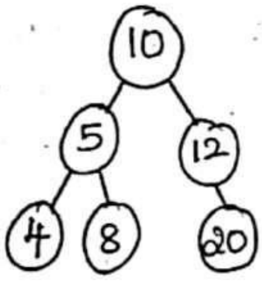
Insert 4



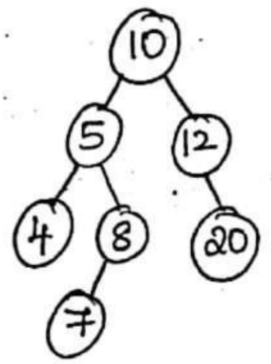
Insert 20



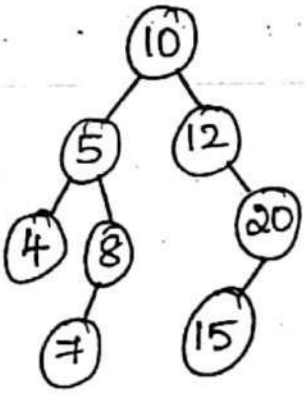
Insert 8.



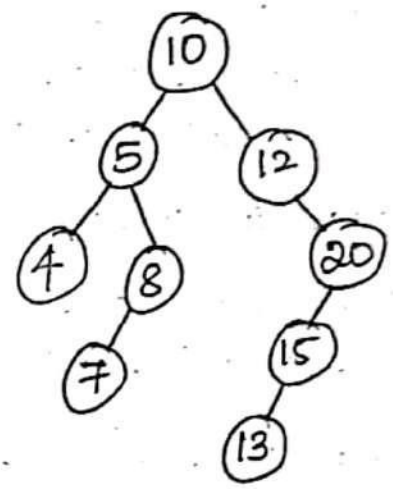
Insert 7.



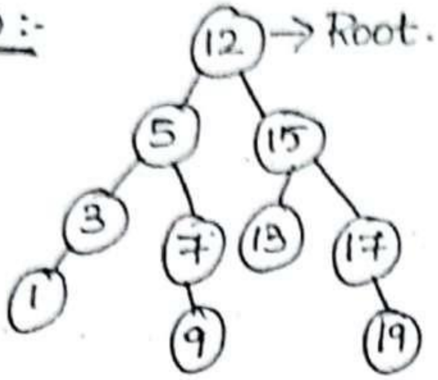
Insert 15.



Insert 13.



Deletion :-

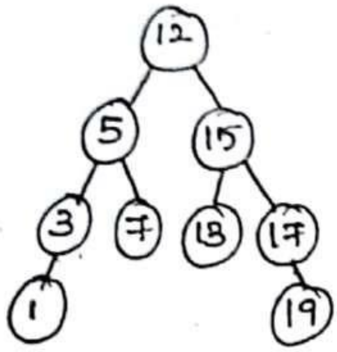


Case 1 : No child

Case 2 : one child

Case 3 : Two children.

Delete Node 9 :-



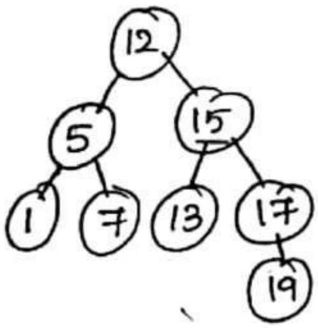
Case 1 : No child.

9 is a leaf node, so we just cut the link and wipe off the node that is clear it from memory.

Delete Node 3 :-

Case 2 : One child.

Wipe it off from memory.



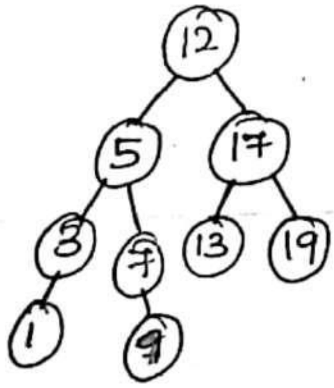
Delete Node 15 :

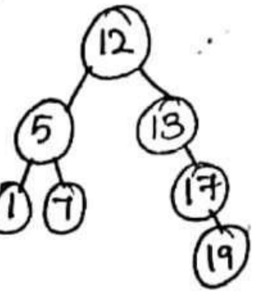
Case 3 : Two child.

Wipe it off from memory.

(minimum element from right side deleted)

(or) Largest element from left side.





Algorithm: INSERTION.

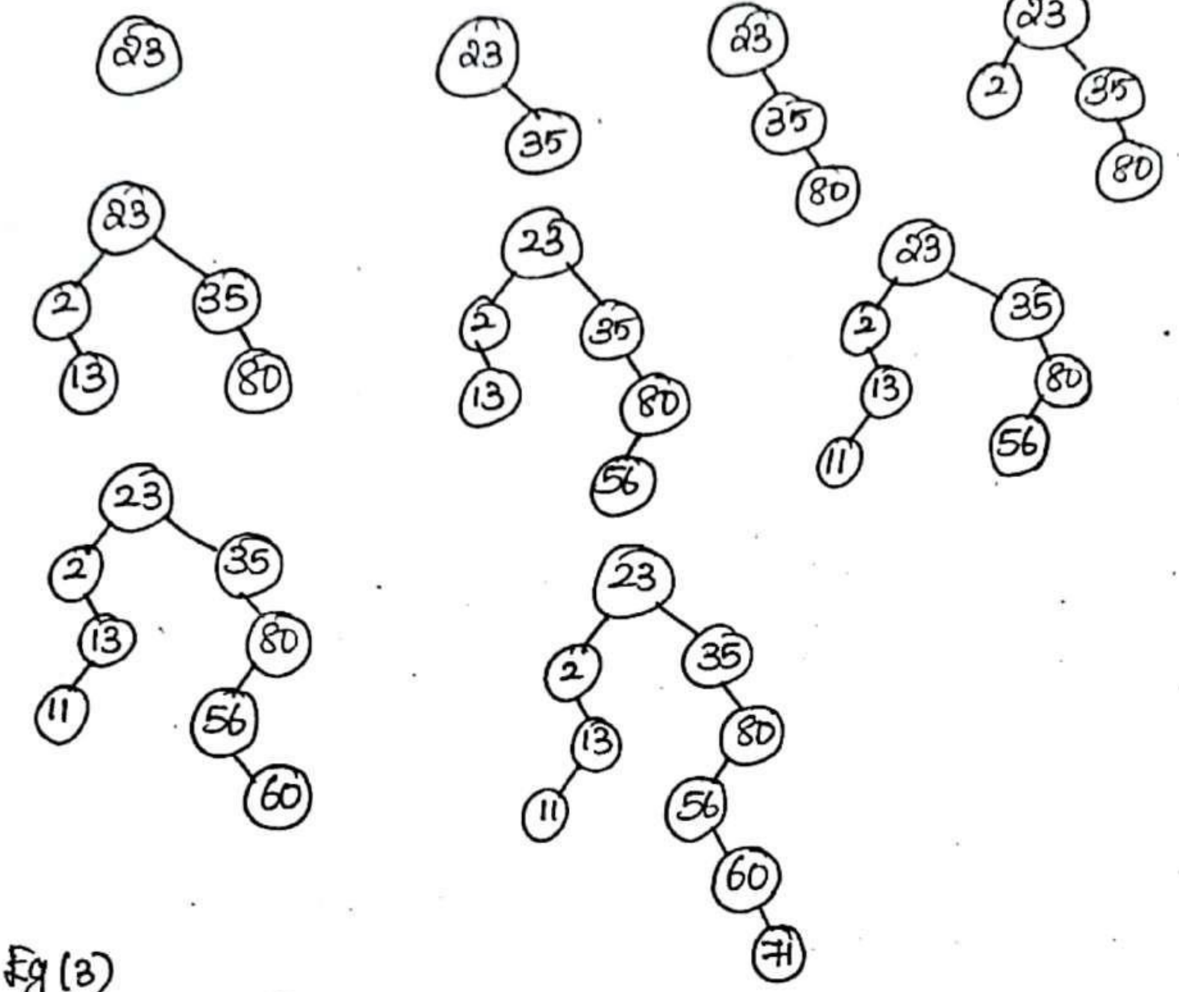
```
Void insert (int data)
{
 Struct node *tempnode = (struct node *) malloc
                         (sizeof (struct node));
 Struct node * current;
 Struct node * Parent;
 tempNode -> data = data;
 tempNode -> leftchild = NULL;
 tempNode -> rightchild = NULL;
 if (root == NULL)
 {
    root = tempNode;
 } else {
    current = root;
    Parent = NULL;
 While (1) {
  Parent = current;
 if (data < parent -> data)
 { current = current -> leftchild;
    if (current == NULL)
    {
    Parent -> leftchild = tempNode;
    return;
    }
   }
   else {
    current = current -> rightchild;
    if (current == NULL)
   {
   Parent -> rightchild = tempNode;
   return;
  }}}
 }
}
```

Algorithm for Search:-

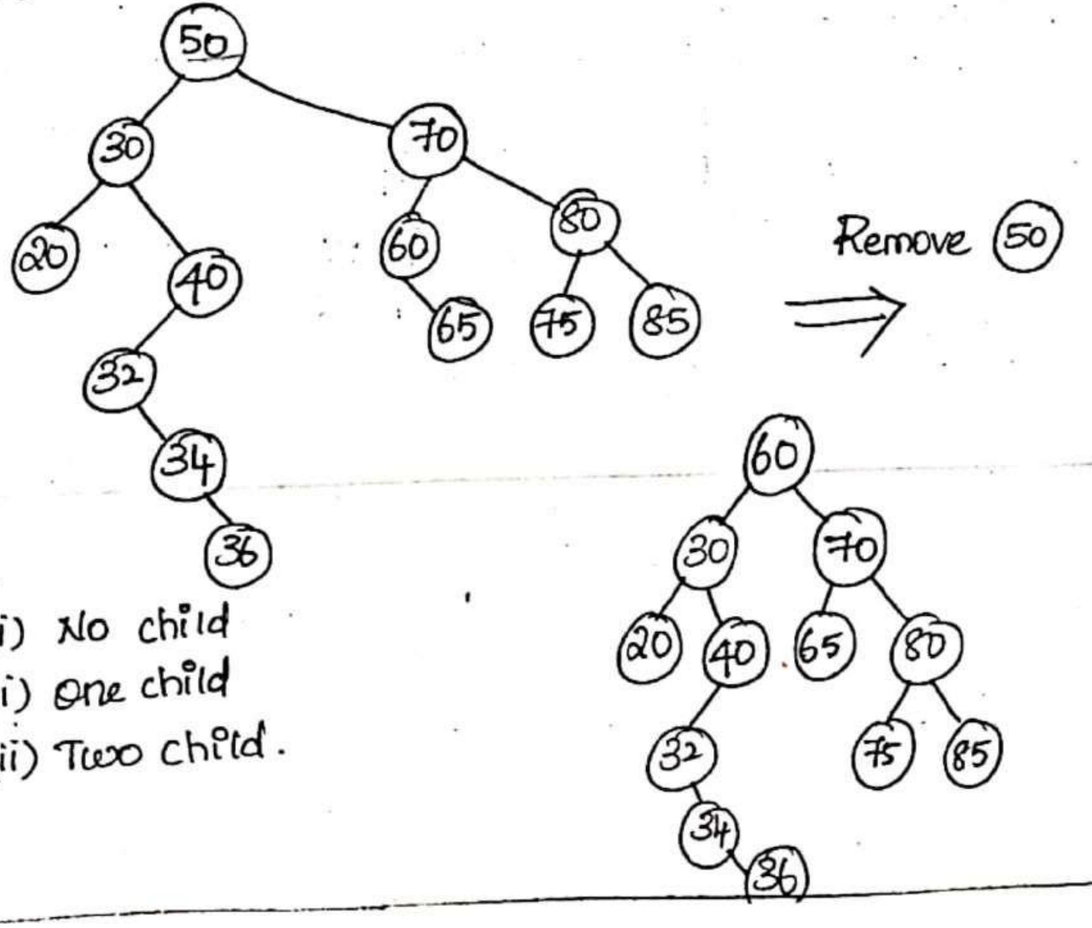
```
struct node* Search (int data)
{
 Struct node* current = root;
Printf ("visiting elements:");
while (current -> data != data)
{  if (current != NULL)
{  Printf ("%d", current -> data);
//goto left tree.
if (current -> data > data)
{ current = current -> leftchild;
 }
else //goto right tree.
 {
 current = current -> rightchild;
 }
 if (current == NULL)
 {
  return NULL;
  }
 }
}
```

Eg (2): Binary Search tree.

23, 35, 80, 2, 13, 56, 11, 60, 71.



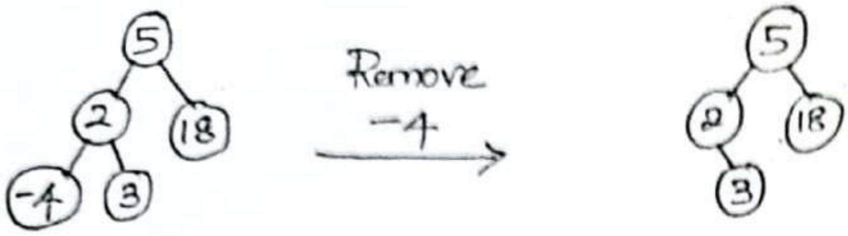
Eg (3)



(i) No child
(ii) One child
(iii) Two child.

(i) No child :-

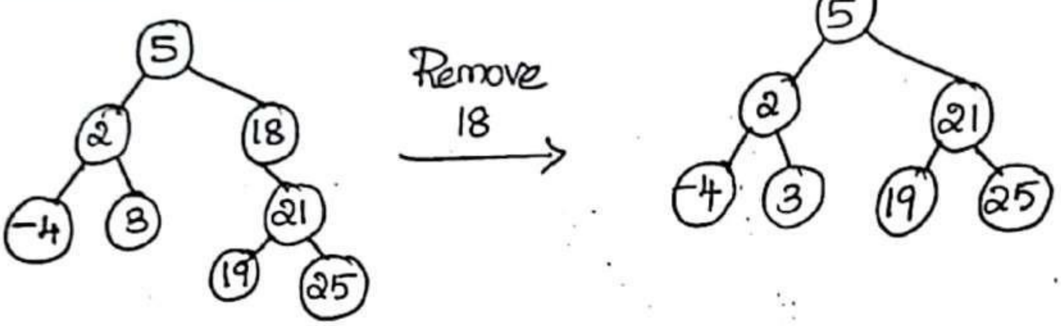
Node to be removed has no children.



(ii) One child :-

"Node to be removed has one child."

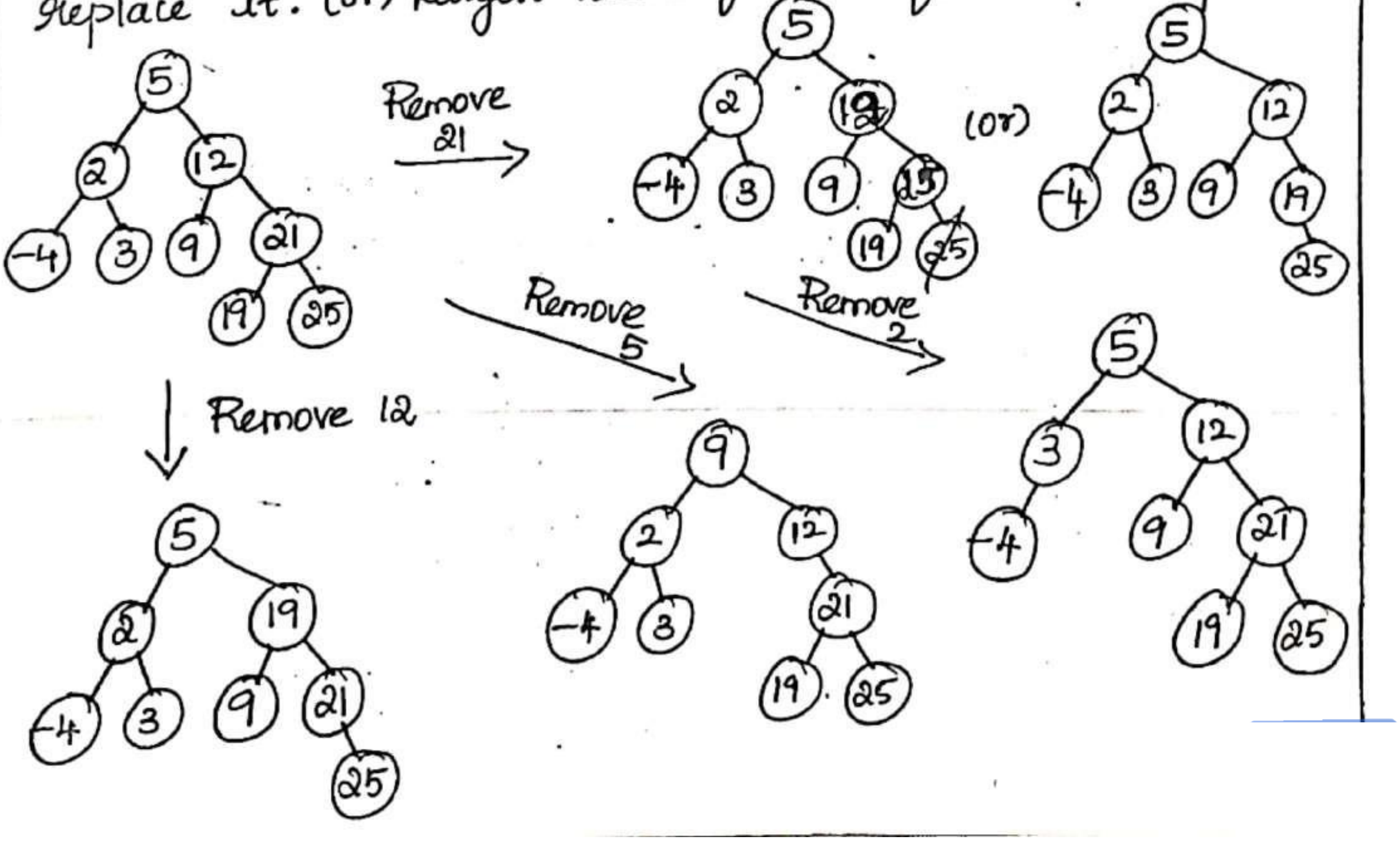
→ Links single child directly to the parent of the removed node.



(iii) Two child :-

Node to be removed has two children.

→ find minimum values in the right subtree & replace it. (or) Largest value from left subtree & replace it.

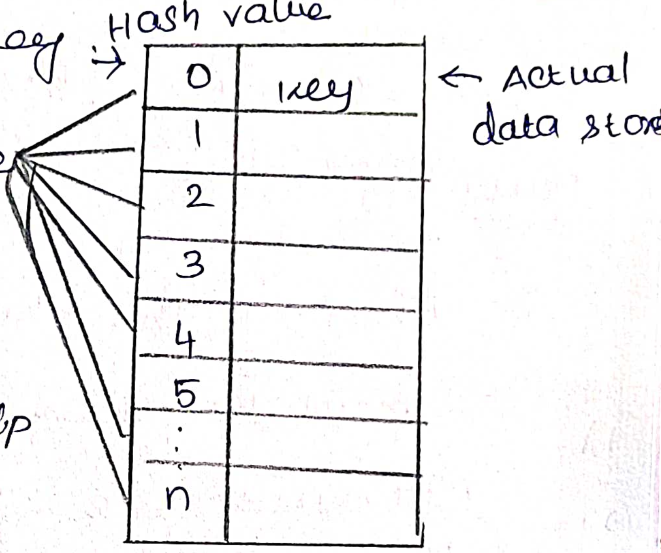


Hashing:

Hashing is the process of mapping large amount data item to a smaller table with the help of a hashing function.

It is the process of indexing and retrieving element in a datastructure to provide faster way of finding the element using the hash key.

Key → Hash function → Hash value



"Hash Table" is just an array which maps a key(data) onto the datastructure with the help of hash function.

"Hash function" is a function which takes a piece of data (key) as I/p and O/p an integer (hash value) which maps the data to a particular index in the hash table.

Hash function:

The mapping b/w an item and the slot where that item belongs in the hash table is called the hash function. The hash function will take any item in the collection and return an integer in the range of slot names, b/w 0 and m-1.

0	1	2	3	4	5	6
None	None	None	None	None	None	None

. 54, 26, 93, 17, 77, 31

h(Item) = Item % m

Item	Hash value	
54	10	(54 mod 11)
26	4	(26 mod 11)
93	5	(93 mod 11)
17	6	(17 mod 11)
77	0	(77 mod 11)
31	9	(31 mod 11)

11) 54 (4
44
10

designated position.

0	1	2	3	4	5	6	7	8	9	10
77	None	None	None	26	93	17	None	None	31	54

Note that 6 out of 11 slots are occupied. This is referred to as "Load factor" and commonly denoted by λ

$$\lambda = \frac{\text{no. of items}}{\text{Table size.}}$$

$$\boxed{\lambda = 6/11}$$

According to the hash function, two (or) more items whould need to be in the same slot. This is referred to as "Collision" also called as "clash".

Two methods of hash functions:

(*) Folding method

(*) mid - square method.

Folding Method :

If our item was the phone number

436 - 555 - 4601

(43, 65, 55, 46, 01) groups of 2.

43+65+55+46+01 ⇒ 210

If hash table has 11 slots, then (210 mod 11) = 1

So, the phone no. 436-555-4601 hashes the slot 1,

Group of 2 (43, 65, 55, 46, 01)

Reverse ⇒ 43, $\overrightarrow{56, 55, 64}$, 01 → Reverse.

43 + 56 + 55 + 64 + 01 = 219

(219 mod 11) = 10

Mid-Square Method :

First we square the item, and then extract som

portion of the resulting digits.

If the item was 44, then $44^2 = 1936$.

By extracting the middle two digits, 93 and performing the remainder step we get (93 mod 11) = 5

Item	Remainder	Mid-square
54	10	3
26	4	7
93	5	9
17	6	8
77	0	4
31	9	6

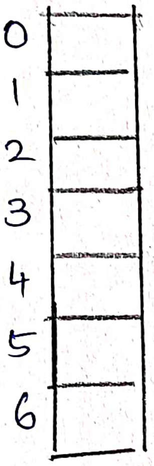
Separate chaining: (N/D - 2019)

The idea is to make each cell of hash table point to a linked list records that have same hash function value.

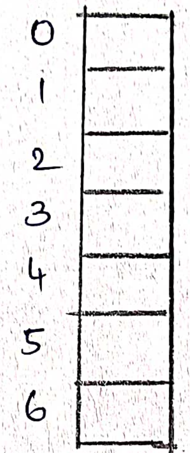
Simple hash function as "key mod 7" and sequence of key as,

50, 700, 76, 85, 92, 73, 101

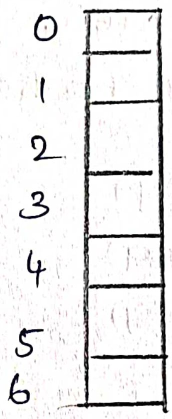
Eg 1:



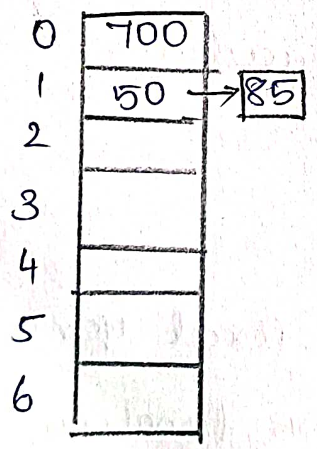
Empty Table



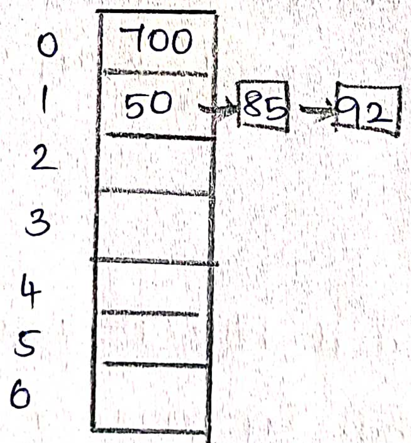
Insert 50
(50 mod 7)
=1



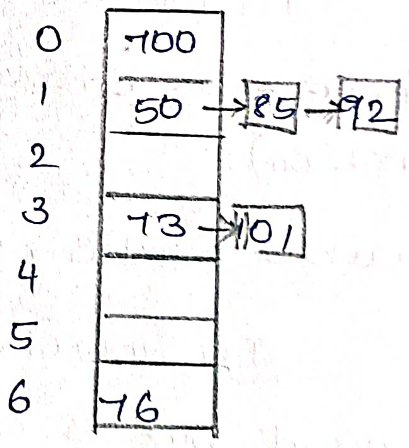
Insert 700 & 76
(700 mod 7)
=0
76 mod 7
=6



Insert 85
(85 mod 7)
=1
Collision occurs
add to chain



Insert 92
(92 mod 7)
=1
Collision occurs
add to chain



Insert 73 and 101
(73 mod 7)
=3
(101 mod 7)
=3

Open Addressing:

Open addressing is a method of collision resolu

in hash tables.

$$\boxed{h_0(key) = (key \bmod arraysize)}$$

with this method a hash collision is resolved by prob

(or) searching through alternate locations in the array

until either the target record is found.

(*) Linear probing

(*) Quadratic probing

(*) Double hashing.

Operations:

⇒ Insert (key)

⇒ Search (key)

⇒ Delete (key)

Linear probing:

In which the interval between probes is fixed - often set to 1.

Quadratic probing:

In which the interval b/w probes increases linearly (hence, the indices are described by a quadratic function).

Double hashing:

In which interval b/w probes is fixed for each record but is computed by another hash function.

Linear Probing

Let us consider a simple hash function as "key mod 7" and sequence of key as,

50, 700, 76, 85, 92, 73, 101.

0	
1	
2	
3	
4	
5	
6	

Empty Table

0	
1	50
2	
3	
4	
5	
6	

(50 mod 7) =1

Insert 50

0	700
1	50
2	
3	
4	
5	
6	76

(700 mod 7) =0

(76 mod 7) =6

Insert 700 and 76

0	700
1	50
2	85
3	
4	
5	
6	76

(85 mod 7) =1 Collision occurs

Insert 85 Collision occurs insert 85 at next field.

0	700
1	50
2	85
3	92
4	
5	
6	76

Insert 92

0	700
1	50
2	85
3	92
4	73
5	101
6	76

Insert 73 and 101

If collision occurs on the particular index field insert the element on next of the index field.

Quadratic Probing

ho(x) = (Hash(x) + i^2) % Hash table size.

Key : 7, 36, 18, 62 and use Hash table size as 11.

0		0		0		0	62
1		1		1		1	
2		2		2		2	
3		3	36	3	36	3	36
4		4		4		4	
5		5		5		5	
6		6		6		6	
7	7	7	7	7	7	7	7
8		8		8	18	8	18
9		9		9		9	
10		10		10		10	

Insert 7 (7 mod 11) =7

Insert 36 (36 mod 11) =3

Insert 18 (18 mod 11) =7

Insert 62 (62 mod 11) =7

Double Hashing:

$h_i(x) = (\text{Hash}(x) + i * \text{Hash2}(x))$ % HashTable size.

$H_1(\text{Key})$ = keymod table size

$H_2(\text{Key}) = M - (\text{Key mod } M)$

m is the prime number < Table size

g: 37, 90, 45, 22, 17, 49, 55

Table size =10

0	
1	
2	
3	
4	
5	
6	
7	37
8	
9	

Insert 37
=(37 mod 10)
=7

0	
1	
2	
3	
4	
5	
6	
7	37
8	
9	

Insert 90
(90 mod 10)
=0

0	
1	
2	
3	
4	
5	45
6	
7	37
8	
9	

45 mod 10
=5

0	90
1	
2	22
3	
4	
5	45
6	
7	37
8	
9	

Insert 22
(22 mod 10)
=2

0	90
1	17
2	22
3	
4	
5	45
6	
7	37
8	
9	

Insert 17
=(17 mod 10)
=7

$H_2(17) = 7-(17 \bmod 7)$
$=7-3$
$=4.$

0	90
1	17
2	22
3	
4	
5	45
6	
7	37
8	
9	49

Insert 49
(49 mod 10)
=9

0	90
1	17
2	22
3	
4	
5	45
6	55
7	37
8	
9	49

Insert 55
(55 mod 10)
=5

$H_2(55) = 7-(55 \bmod 10)$
$=7-6$
$=1$

Rehashing:

Rehashing is the process of re-calculating the hashcode of already stored entries (key value pair) to more item to another bigger size Hash map when load factor threshold is reached.

Load factor: It is a measure, "till what load, hashmap can allow elements to put in it before its size is increased.

Steps:

1) Create a large table.
2) Create a new hash function.
3) use the new hash fun to add the existing data items from the old table to the new table.

Rehashing Techniques:

1) Linear probing
2) Two-pass file creation
3) Separate overflow area
4) Double hashing
5) Synonym chaining
6) Bucket Addressing
7) Bucket chaining.

Eg: 13, 15, 24, 6 . Table size : 7.

$\boxed{\text{Hash function } h(x) = x \bmod 7.}$

Linear probing with i/p 13, 15, 24 and 6

0	6
1	15
2	
3	24
4	
5	
6	13

(13 mod 7) = 6

(15 mod 7) = 1

(24 mod 7) = 3

(6 mod 7) = 0.

After 23 is inserted.

0	6
1	15
2	23
3	24
4	
5	
6	13

(23 mod 7)
= 2.

∴ Table is 70% full
so new table is created
with size 17.

∴ New hash function is then,

h(x) = x mod 17.

The old table is scanned, and elements
6, 15, 23, 24, 13 are inserted onto the new table.
This entire operation is called as rehashing

Open Addressing hash table after rehashing.

6, 15, 23, 24, 13

0	
1	
2	
3	
4	
5	
6	6
7	23
8	24
9	
10	
11	
12	
13	13
14	
15	15
16	

Rehashing is expensive operation with running time O(N)

Extendible hashing:

Used when the amount of data is too large to fit in main memory and external storage is used.

N records in total to store

M records in one disk block.

In ordinary hashing several disk blocks are examined, to find an element, a time consuming process.

Idea:

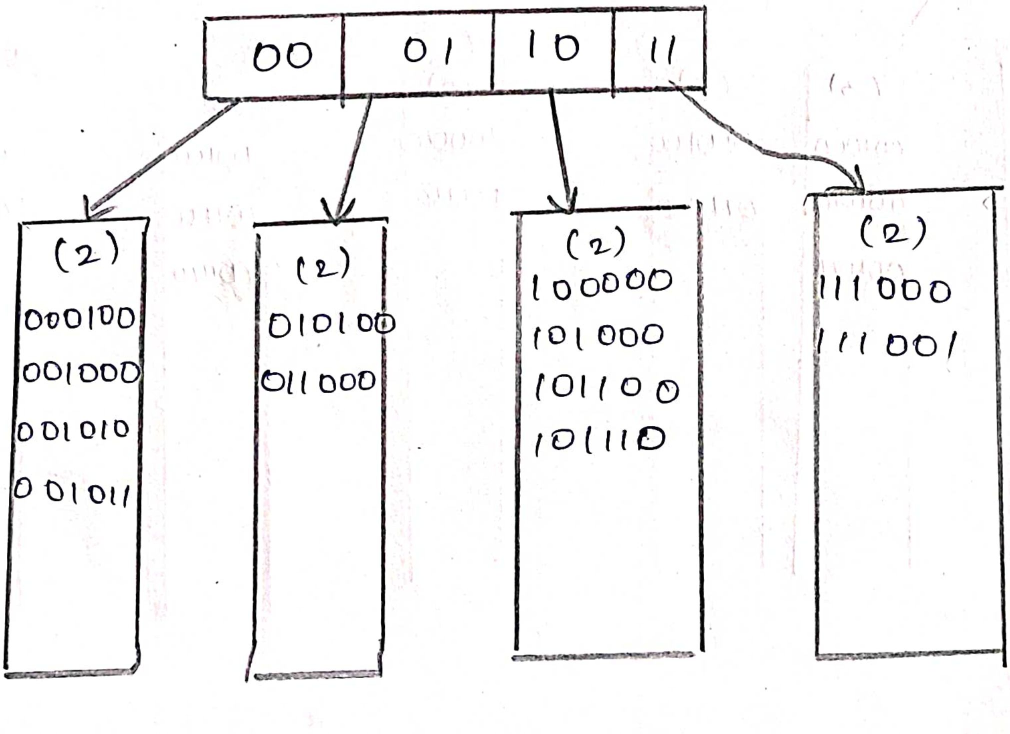
⇒ Keys are grouped according to the first m bits in their code.

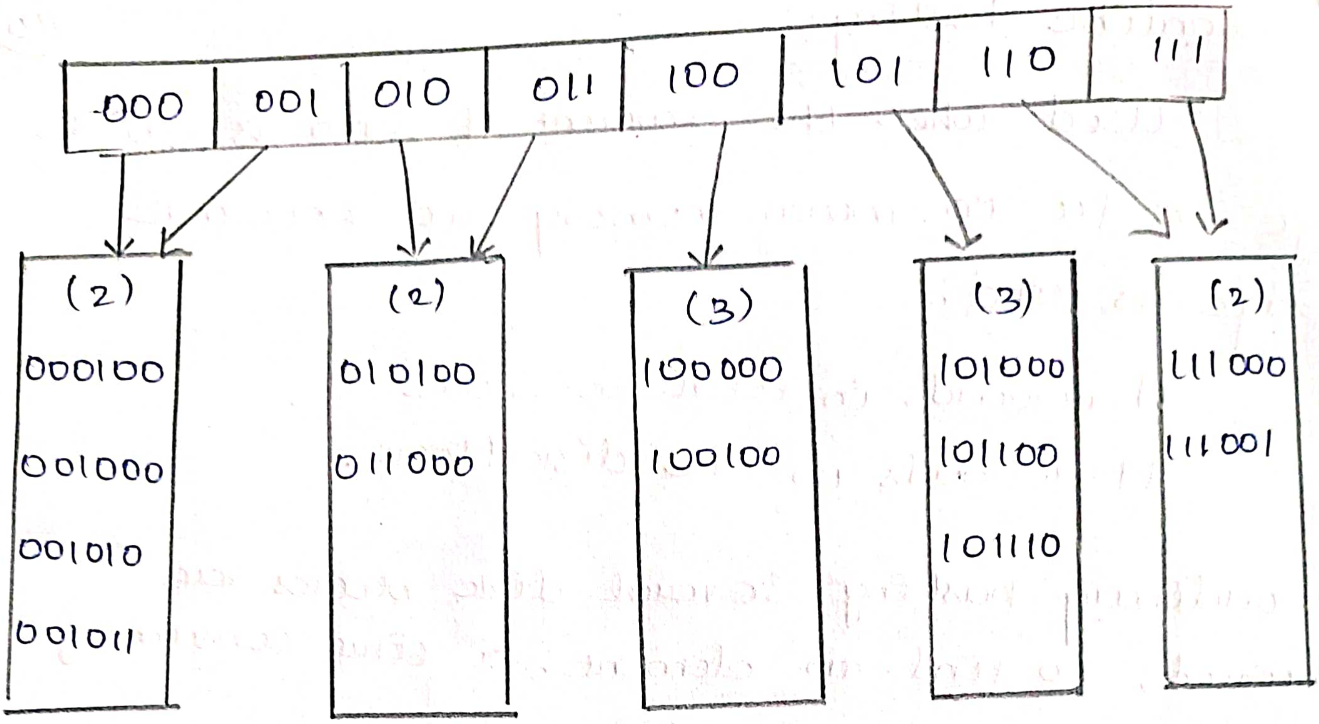
⇒ Each group is stored in one disk block.

Eg: Suppose data consists of several six-bit integers.

Each leaf has up to M=4 elements.

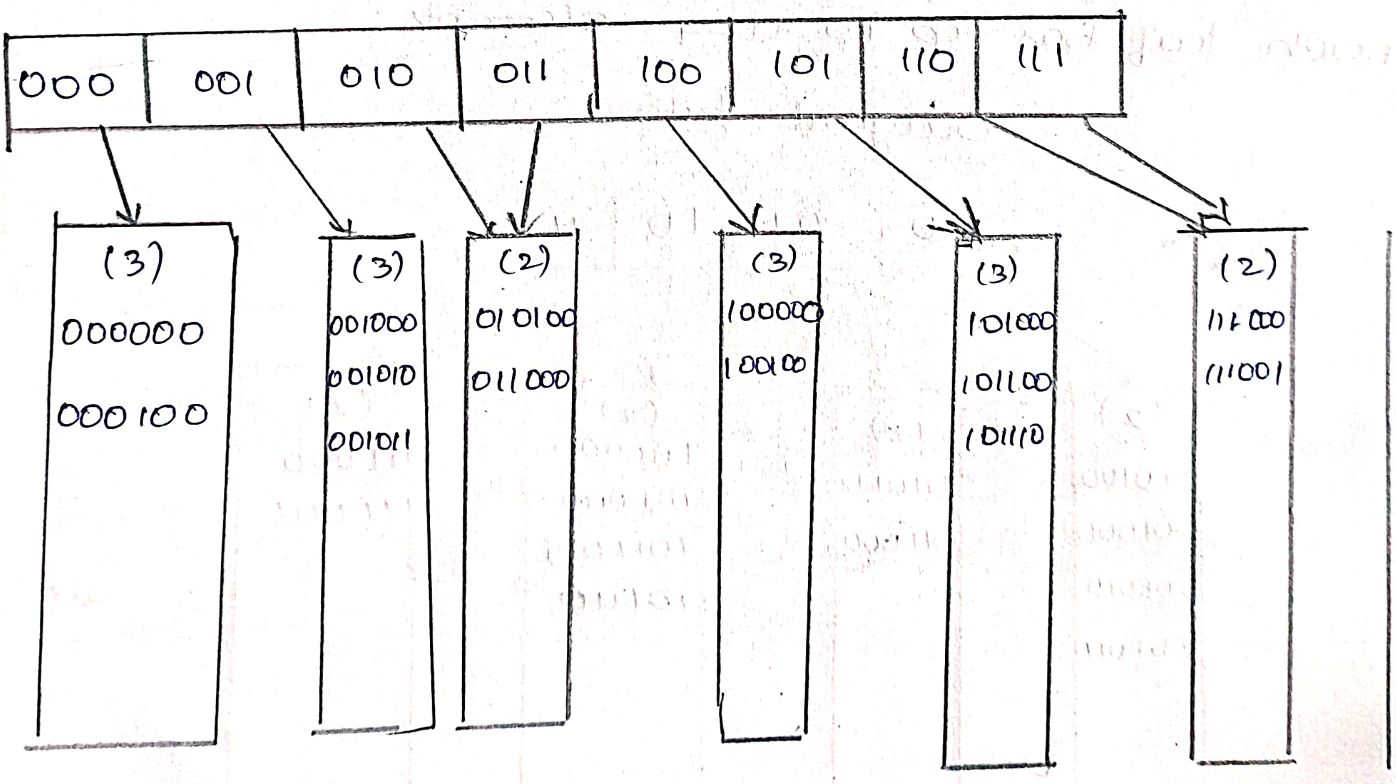
Original data.





Extendible hashing after insertion of 100100 and directory split.

Extendible hashing after inserting of 000000 and leaf split



Unit - V

Searching, Sorting and Hashing Techniques.

Searching: It is an operation (or) a technique that helps finds the place of a given element (or) value in the list.

(*) Linear search (or) Sequential search.

(*) Binary search.

Linear search: (or) Sequential search:

It is a very simple search algorithm. In this type of search, a sequential search is made over all items one by one.

Steps: 1) Starts from the leftmost element of arr[] and one by one compare x with each element of arr[].

2) If x elements matches with an element, return the index.

3) If x doesn't match with any elements, return -1.

0	1	2	3	4	5	6	7
65	20	10	55	32	12	50	99

search(12)

(12) ⇒ Both not match. Move to next element

0	1	2	3	4	5	6	7
65	20	10	55	32	12	50	99

(12)

0	1	2	3	4	5	6	7
65	20	10	55	32	12	50	99

(12)

0	1	2	3	4	5	6	7
65	20	10	55	32	10	50	99

(12)

0	1	2	3	4	5	6	7
65	20	10	55	32	12	50	99

(12)

0	1	2	3	4	5	6	7
65	20	10	55	32	12	50	99

(12)

Both are matching. So we stop comparing.

Advantages:

1) The linear search is simple.

2) It does not require the data in the array to be stored in any particular order.

Binary search:

Binary search, is also known as half-interval search, is a search algorithm that finds the position of a target value within a sorted array.

I/P: 65,20,10, 55, 32, 50, 99.

list

0	1	2	3	4	5	6	7	8
10	12	20	32	50	55	65	80	99

search(12)

(Arranged in ascending order)

list

0	1	2	3	4	5	6	7	8
0	12	20	32	50	55	65	80	99

↑
(12)

8/2 =4
7/2 =3

Search element (12), compared with middle element (50)
12<50. So search only in the left sublist
(10, 12, 20, 32)

list

0	1	2	3
10	12	20	32

5/2 =2

0	1	2	3
10	12	20	32

3/2 =1

(12)

0	1	2	3	4	5	6	7	8
10	12	20	32	50	55	65	80	99

Search(80)

80 > 50

5	6	7	8
55	65	80	99

5	6	7	8
55	65	80	99

7	8
80	99

7	8
80	99

↑
80

Both are matching. so element 80 is found at Index 7.

Advantages:

⇒ faster, because does not have to look at every element.

Disadvantages:

⇒ ~~This alg.~~ requires the list to be sorted.

Sorting:

Sorting refers to arranging data in a particular format. It specifices the way to arrange data in particular order.

⇒ Bubble sort

⇒ Selection sort

⇒ Insertion sort.

⇒ Shell sort

⇒ Radix sort.

Bubble sort:

Bubble Sort is a simple sorting algorithm. This sorting alg. is comparission - based algorithm in which each pair adjacent elements is compared and the elements are swapped if they are not in order.

Eg: 14, 33, 27, 35, 10

14, 33, 27, 35, 10 [14 < 33 Already sorted no swap]

14, 33, 27, 35, 10 [33 > 27, so swap it]

14, 27, 33, 35, 10 [33 < 35. Already sorted]

14, 27, 33, 35, 10 [35 > 10, so swap it]

14, 27, 33, 35, 10 [33 > 10, so swap it]

14, 27, 33, 10, 35 [27 > 10, so swap it]

14, 10, 27, 33, 35 [14 > 10, so swap it]

10, 14, 27, 33, 35 [Last Iteration].

And when there is no swap required, bubble sort learns that an array is completely sorted.

Selection Sort:

Selection sort is a simple algorithm. This sorting algo. in - place comparsion based algorithm in which the list is devided into two parts, the sorted part at the left end and the unsorted path at the right end.

Initally, the sorted part is empty and the unsorted part is the entire list.

14	33	27	10	35	19	42	44

The whole list is scanned sequentially.

14, 33 27 10 35 19 42 44 (swap)

↑min.

we replace 14 with 10.

10 | 33 27 14 35 19 42 44

(sorted) ↑mon

10 33 27 14 35 19 42 44

After swapping two least values are positioned.

10 14 | 27 35 19 42 44 (swap it)

Sorted ↑min

10 14 19 | 33 35 27 42 44 (swap it)

Sorted. ↑min

10 14 19 27 | 35 33 42 44 (swap it)

Sorted. ↑min

10 14 19 27 33 | 35 42 44 (NO need swapping)

Sorted ↑min

10 14 19 27 33 35 | 42 44

sorted. ↑min.

Sorted order is

10	14	19	27	33	35	42	44

Insertion sort:

This is an in-place comparison-based sorting algorithm.

Steps: 1) If it is the 1st element, it is already sorted. return 1.

2) Pick next element

3) Compare with all elements in the sorted sub-list.

4) Shift all the value elements in the sorted sub-list that is greater than the value to be sorted.

5) Insert the value.

6) Repeat until list is sorted.

Original	34	8	64	51	32	21	position moved.
After P=1	8	34	64	51	32	21	1
After P=2	8	34	64	51	32	21	0
After P=3	8	34	51	64	32	21	1
After P=4	8 8 8	34 34 32	51 32 34	32 51 51	64 64 64	21 21 21	3
After P=5	8 8 8 8	32 32 32 21	34 34 21 32	51 21 34 34	21 51 51 51	64 64 64 64	4

No. of elements : 6

n-1 : 6-1 ⇒ 5 passes.

Merge Sort:

↳ Merge sort is sorting techniques based on devide and conquer techniques.

↳ Complexity O(n log n)

↳ It devides the array into equal halves and then combines them in a sorted manner.

Merge sort works:

14	33	27	10	35	19	42	44

Devide into equal array (4)

14	33	27	10

35	19	42	44

Devide into array (2)

14	33

27	10

35	19

42	44

Devide into single array

14

33

27

10

35

19

42

44

Combine each element : 14,33

14	33

10	27

19	35

42	44

10	14	27	33

19	35	42	44

10	14	19	27	33	35	42	44