Unit 1

C- Programming Fundamentals.

Datatypes - 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 Ritchie

⇒ It is also a high level programming language

⇒ It is a case sensitive language.

⇒ (ie) it differentiates two variables having same letters but different cases (upper and lower case).

Keywords, Variables and Constants :

Keywords :

⇒ Standard words in C.

⇒ Reserved words with special meaning

⇒ The meaning of keywords cannot be changed.

Keywords in C

alien	auto	break
case	char	const
continue	default	do
double	else	enum
extern	float	for
goto	if	int
long	register	return
short	signed	sizeof
static	struct	switch
typedef	union	unsigned
void	volatile	while .

Identifiers :

⇒ collection of alphanumeric characters.

⇒ First character must not be numeric.

⇒ are the names given to the variables, functions or constants.

⇒ Name of the identifier must not be the keyword.

Validity of variable names:

⇒ Following rules should be followed while deciding the variable names.

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

↳ Special characters (such as $, #, %) are not allowed in the variable name except underscore.

↳ The variable name is case sensitive.

↳ A variable name can be in capital letters.

↳ The keywords are not valid variable names.

↳ The length of the variable name can be any but only first 31 characters are recognized.

↳ Blank space or special characters, commas, use of quotes are not allowed in the variable name.

↳ Arithmetic operators should not be used in the variable names.

↳ Variables names should be Informative.

It should describe the purpose of it by its name.

⇒ The valid variables are.

- → Count
- → tax_id
- → INDEX
- → Xyz
- → brick01

⇒ The invalid variable names are

- → -file
- → char
- → #id
- → 1A
- → valid name

Constants :

⇒ The specific alphabetical or numerical value that never gets changed during the procesing of the instructions is called as constant.

⇒ constant can be alphabetical, numeric or special symbol.
⇒ constants are given names and refered by names.
⇒ constant names helps in accessing them easily.
⇒ Generally, all the letters of (eg) PI the constant names are capital.
⇒ commonly used constant.
⇒ once value is assigned it doesnot get changed.

Constants	Variables
1. constants cannot be changed.	Values of the variable may get changed during processing.
2. Storage location is given a name.	Storage location is given names.
3. Named constants the constants are given the names and are referred by the given name.	The variables are given the names and are referred by the name in the instruction.
4. Eg. PI = 3.142	Eg. Name = "AAA", EnglishMarks = 50

Header Files:

⇒ contains standard library functions.

⇒ For using these library functions directly in the program, header files are to be included at the beginning of the program.

Following are the header files supported by C.

alloc.h	assert.h	bcd.h	dir.h
complex.h	conio.h	ctype.h	fcstream.h
dirent.h	dos.h	errno.h	math.h
fcntl.in	fstream.h	generic.h	share.h
float.h	graphics.h	iomanip.h	string.h
io.h	limits.h	malloc.h	
locale.h	mem.h	setjmp.h	
process.h	signal.h	stddef.h	
stdarg.h	stdio.h	stream.h	
stdlib.h	stdiostr.h	values.h	
time.h	strstrea.h	bios.h	

⇒ Commonly used header files are

⇒ stdio.h → Standard Input/output header file. Functions for printf, scanf, fprintf, fscanf are defined.

⇒ conio.h

⇒ math.h → These functions deals with input and output functions

⇒ alloch

conio.h → Console input Output header file.

→ Clrscr() function is defined. (O/p screen gets cleared)

math.h → all the functionalities related to the mathematical operations are defined.

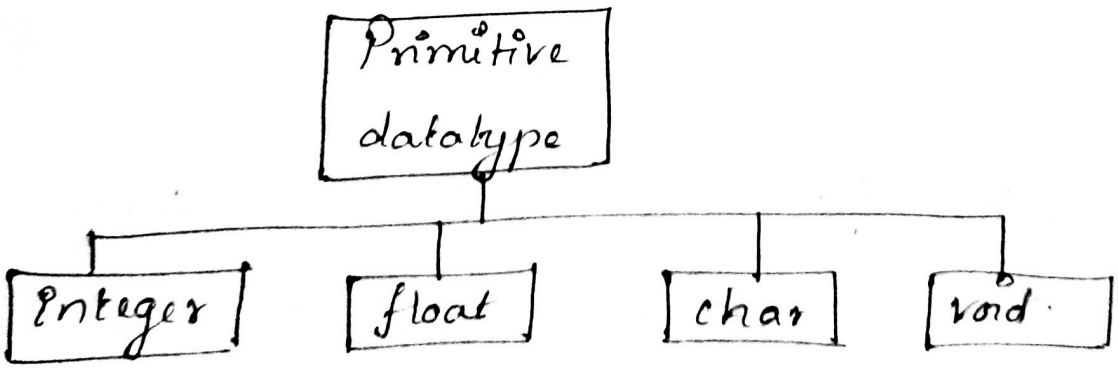
- pow → computing power
- sqrt → computing square root

alloc.h → allocating memory dynamically. malloc() is used in the program.

Data Types:

⇒ Specify the type of data entered in the program.

⇒ There are some predefined set of data types called as primitive data types.



Datatypes.

⇒ Primitive datatypes are the fundamental data types.

1. Integer type : whole numbers without fraction.

2. float type : real numbers with fraction

3. char type : character value.

4. void type : no value associated with the function with no return value.

1. integer type :

	Range
⇒ int or signed int (2 bytes)	→ -32768 to 32767
⇒ unsigned int (2 bytes)	→ 0 to 65535
⇒ short int or signed short int (1byte)	→ -128 to 127
⇒ unsigned short int (1 byte)	→ 0 to 255
⇒ long int or signed long int (4 bytes)	→ -2,147,483,648 to 2147483647
⇒ unsigned long int (4 bytes).	→ 0 to 4,294,967,295

2. Float type :

	Range
⇒ float (4 bytes)	→ 3.4E -38 to 3.4E+38
⇒ double (8 bytes)	→ 1.7E -308 to 1.7E+308
⇒ Long double (10 bytes)	→ 3.4E-4932 to 1.1E+4932.

3. char type.

	Range
char or signed char (1 byte)	-128 to 127
unsigned char (1 byte)	0 to 255

Operations:

Expressions using operators:

⇒ For handling expressions the operators are used. C supports the following set of operators.

Type	Operator	Meaning	Example.
Arithmetic	+	Addition / Unary plus	c = a+b
	-	Subtraction / Unary minus	d = -a c = a-b
	*	Multiplication	c = a*b
	/	Division	c = a/b
	%	Mod	c = a % b.
Relational	<	Less than	a < 4
	>	Greater than	b > 4
	<=	Less than equal to	a <= 4
	>=	Greater than equal to	a >= 4
	==	Equal to	a == 4
	!=	Not equal to	a != 4
Logical	&&	And	0 && 1
	||	Or	0 or 1
Assignment	=	Is assigned to	a = 5
Increment	++	Increment by 1	++i or i++
Decrement	--	Decrement by 1	--x or x--

Conditional operator:

=> The conditional operator is [?:]

Syntax:

condition ? expression 1 : expression 2

True condition ← ↳ False condition

Eg:

a > b ? true : false

If a > b, then true otherwise false.

Precedence of Operators:

=> Precedence means the priority of certain Operation.

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

1. In case, if there is a tie between operations of same priority, then the preference will be given to the operator which occurs first.

2. In case of parenthesis, if there are more than one set of parenthesis inner most parenthesis, will be performed first, followed by operations within second.

(eg) $a = 3/3 * 4 + 3/8 + 3.$ (/)

$a = 1 * 4 + 3/8 + 3$ (*)

$= 4 + 3/8 + 3$ (/)

$= 4 + 0 + 3$ (+)

$= 7$

Input and Output Operations:

⇒ In C, `Scanf` is used for inputting data

`printf` is used for outprinting or printing the data on console (output screen).

⇒ There are standard library functions.

Syntax

scanf (formatspecifier, variables);

printf (format specifier, variables);

Example:

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

- `&val` → variables
- `"%d"` → formal specifier
- `%d` → val is integer variable
- `&` symbol → means (ampersand) 'address of'
- → Any variable is referred by its address

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

→ no need of & here because value of val variable is printed which is already stored in some statement.

Decision making and conditional statements:

⇒ Various control structures are,

1. if statement
2. while statement
3. do while statement
4. switch case statement
5. for loop

1. if statement

⇒ There are 2 types of if statement

① → simple if statement

② → compound if statement

simple if

⇒ if statement followed by single statement

Syntax:

```
if (condition)
    statement
```

Example

```
if(a<b)
printf("a is smaller than b");
```

Compound if

⇒ if statement is followed by group of statements.

Syntax

```
if (condition)
{
  statement 1;
  ...
  ...
}
```

Example:

```
If (a<b)
{printf("a is smaller than b");
printf("b is larger than a");}
```

if ... else :

Syntax

```
if (condition)
 Statement ;
else
 Statement ;
```

Example:

```
If(a<b)
 printf("a is smaller than b");
else
 printf(" b is smaller than a");
```

Compound if ... else

Syntax

```
if (condition)
 {
  Statement 1 ;
  ...
 }
else
 {
  Statement 2;
  ...
 }
```

Example

```
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 ... else:

Syntax

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

Example

```
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 and
        c");
```

While Statement:

⇒ while statement executes repeatedly until the condition is false.

⇒ The while statement can be simple while or compound while.

Simple while :

Syntax

```
while (condition)
    statement ;
```

Example

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

Compound while :

Syntax

```
while (condition)
{
statement 1 ;
...
}
```

Examples

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

do ... while

⇒ used for repeated execution.

↳ In while statement, the condition is checked before executing the statement whereas in do.. while statements, the statement is executed first and then the condition is tested.

↳ There will be atleast one execution of statements in do... while loop.

Syntax

```
do
{
Statement 1;
...
} while (condition);
```

↳ while condition is terminated by a semicolon.

Example

```
do
{ print("a is less than b");
a++;
} while(a<10);
```

Switch.. case statement:

⇒ From multiple cases if one case is to be executed at a time, then switch case statement is executed.

Syntax

```
switch (condition)
{
case caseno :
        Statements;
        break;
.....
default :
        Statements;
}
```

Example

```
printf("\n Enter choice");
scanf("%d", &choice);
switch(choice)
{
case 1 : printf("1 is
                selected");
break;
case 2 :
  printf("2 is selected");
break;
case 3 :
  printf("3 is selected");
break;
```

for loop:

=> Repeated execution of statement occurs using for loop

Syntax:

```
for (initialization; termination; step count)
statement
   or
{
statement 1;
statement 2;
...
}
```

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

Example:

```
for( i=0; i<10; i++)
{
for(j=0; j<10; j++)
c[i][j] = a[i][j] + b[i][j].
```

Functions:

=> If certain set of ~~for~~ instructions are required frequently, then these instructions are wrapped within a function.

⇒ Functions are handled using three methods.

1. Function declaration
2. Function definition
3. Function call.

Function declaration:

```
return-type Function-name (data-type parameter);
```

Function definition

```
return-type Function-name (data-type Parameter)
{
// Function body
}
```

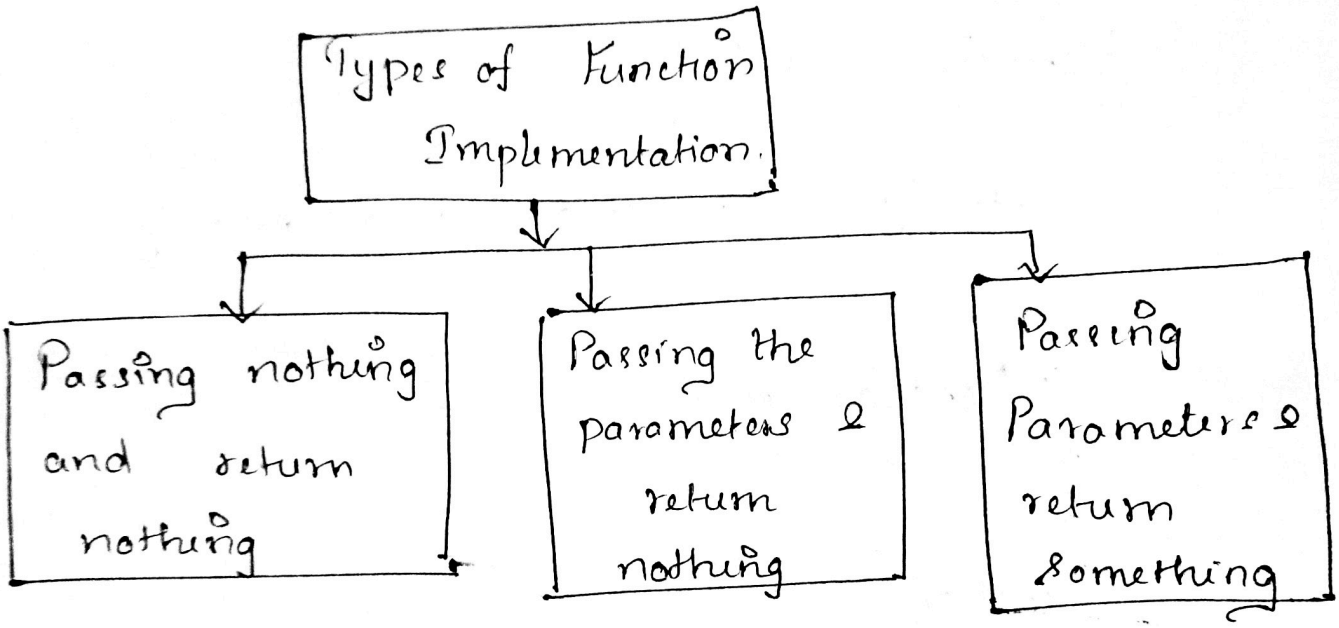
Function call

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

Example:

```
void main()
{
  void sum(); Function declaration
  sum();      Function call
}
  void sum()  Function definition.
```

```
{
  int a, b, c ;
  printf ("\n Enter the two numbers");
  scanf ("%d %d", &a, &b) ;
  c = a+b;
  printf ("\n Addition of two numbers is %d", c);
}
```



Parameter passing Method :

Type 1 : Passing nothing and return nothing

Type 2 : Passing the parameter and return nothing.

Type 3 : passing the parameters and returning from the function.

Type 1 : Passing nothing and return nothing

Example

```
Void main()
{
 Void sum();
  sum();
}
 Void  sum()
{ int a, b, c;
 printf("\n Enter the numbers");
 Scanf("%d %d", &a, &b);
 c=a+b;
 printf("\n Sum is %d", c);
}.
```

Type 2 : Passing the parameter and return nothing

Example (call by value).

```
main()
{
 int a, b;
 void sum(int a, int b);
 printf("\n Enter the two numbers");
  Scanf("%d", &a, &b);
   Sum(a, b);
}
```

```
void sum ( int x, int y )
{ int c;
  c = x + y;
  printf ("\nSum is %d", c);
}.
```

⇒ Parameters a and b are passed to x and y respectively.

⇒ There are two methods of parameter passing.

1) call by value
2) call by reference.

⇒ The above examp2 is passing parameters by call by value. This is called, call by value because the values are passed.

↑

(2) call by value:

(2) call by reference:

⇒ In call by reference, the parameters are taken by reference.

⇒ Pointer variables are passed as parameter

Example : 1

```
main()
{
int a, b;
void sum(int *, int *);
printf("\n Enter the two numbers");
scanf("%d %d", &a, &b);
sum(&a, &b);
}
void sum(int *x, int *y)
{
 int c;
c = *x + *y;
printf("%d", c);
}
```

Example 2

```
void swap(int *a, int *b)
{
int temp;
temp = *a;
*a = *b;
*b = temp;
}
```

```
void swapval(int a,
             int b)
{int temp;
 temp = a;
 a = b;
 b = temp;
}
```

```
int main()
{
  int x, y;
  printf("\n Enter the first number:");
  scanf("%d", &x);
  printf("\n Enter Second number:");
  scanf("%d", &y);
  printf("\n Before swapping x = %d and
                    y = %d", x, y);
  swap_val(x, y);
  printf("\n After swapping x = %d and
                    y = %d", x, y);
  printf("\n Before swapping x = %d and y = %d",
  swap(&x, &y);                          x, y);
  printf("\n After swapping x = %d and y = %d",
                                         x, y);
}
```

Output

```
Enter first number : 10
Enter Second number : 20
Before swapping x = 10 and y = 20
After swapping x = 20 and y = 10
Before swapping x = 10 and y = 20
After swapping x = 20 and y = 10.
```

Recursive Functions:

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

Properties:

⇒ There are two fundamental properties of recursion.

(1) There must be atleast one condition in recursive function which donot involve the call to recursive routine.

↳ This is called the way out of the sequence of recursive calls.

↳ This is called base class property.

(2) The invoking of each recursive call must reduce to some manipulation and must go closer to base class condition.

Factorial

```
if (n==0)  → base case
    return 1;
else
    return n * fact(n-1);  → On each call the computation goes closer to base case.
```

Factorial of a number using recursive function

Pgm:

```
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>
void main(void)
{
 int n,f;
 int fact(int n); / declaration /
 clrscr();
 printf("\n Enter the number to find the factorial");
 scanf("%d", &n);
 f = fact(n); / Function call /
printf("\n The factorial of %d is %d", n,f);
getch();
}
int fact(int n)
{
 int x,y;
 if(n<0)
 printf(" Negative parameter in the factorial function");
 exit(0);
}
 if (n==0)
 return 1;
```

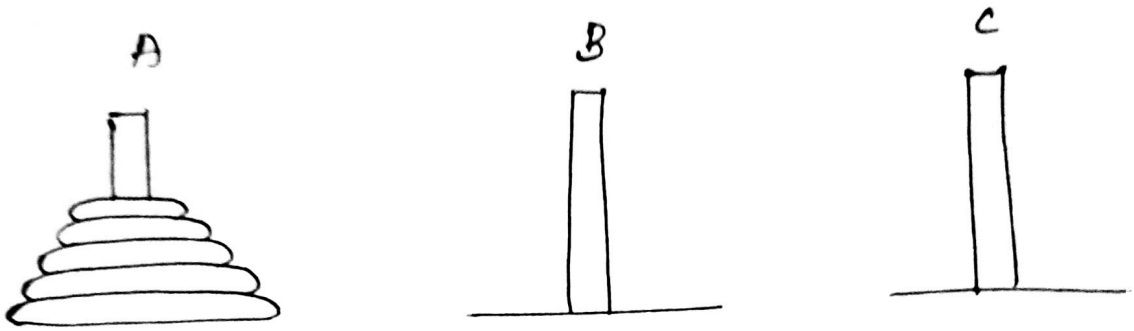
```
x = n-1;
y = fact(x);   → recursive call.
return (n*y);
}
```

Fibonacci Series:

```
# include < stdio.h >
# include < conio.h >
void main(void)
{
 int n, num;
 int fib(int n);
 clrscr();
 printf("\n Enter location in fibonacci series:");
 scanf("%d", &n);
 num = fib(n);
 printf("\n The number at %d th position is %d
        in fibonacci series", n, num);
getch();
}
int fib(int n)
{
 int x, y;
 if(n<=1)
   return n;
```

```
x = fib(n-1);
y = fib(n-2);
return (x+y);
}
```

Towers of Hanoi



⇒ There are three pegs named A, B and C

⇒ Five disks of different diameters are placed on peg A.

⇒ The arrangement of disks is such that each smaller disk is placed on the larger disk

Towers of Hanoi - Problem.

⇒ Move the five disks from peg A to Peg C using peg B as an auxillary.

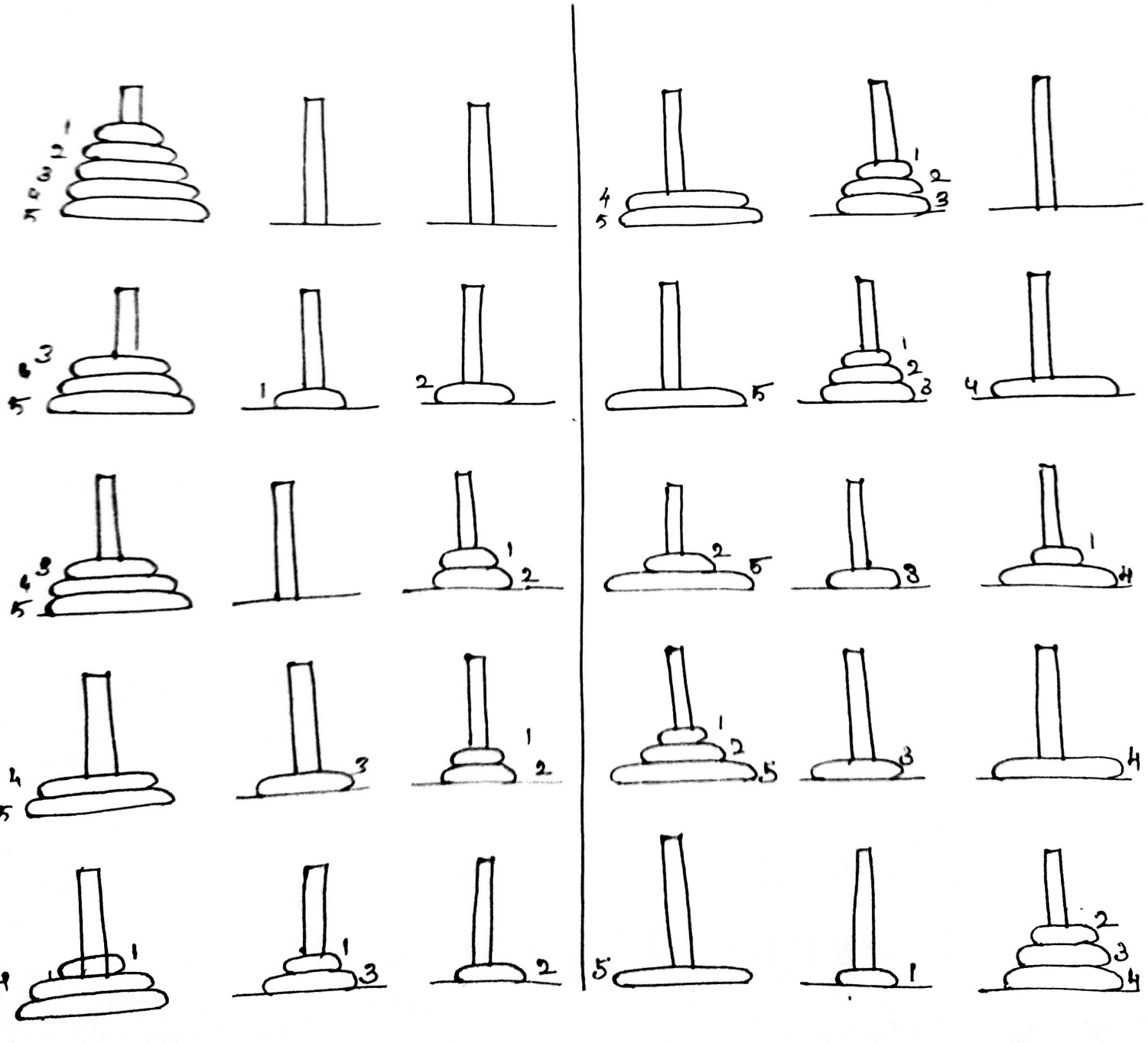
Conditions:

(i) Only the top disk on any peg may be moved to any other peg.

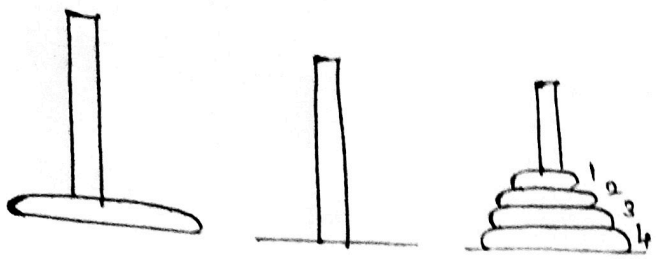
(ii) A larger disk should never rest on the smaller one.

Solution :

1. Move top n-1 disks from A to B using C as auxillary.
2. Move the remaining disk from A to C
3. Move the n-1 disks from B to C using A as auxillary.



5



Similarly the remaining disk can be moved from A to C

Program:

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
void main (void)
{
int n;
void towers (int n, char from, char to, char aux);
clrscr();
printf("\n Towers of Hanoi");
printf("\n Enter the total number of disks");
scanf("%d", &n);
towers(n, 'A', 'C', 'B');
getch();
}
void towers (int n, char from, char to, char aux)
{
  if (n==1)
  {
```

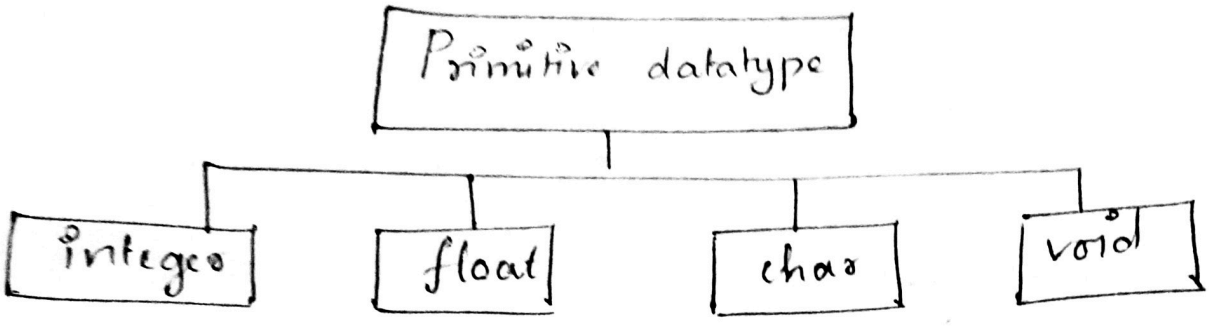
```
printf("In Move disk 1 from %c peg to %c peg",
                                    from, to);
return;
}
towers(n-1, from, aux, to);
printf("In Move disk %d from %c peg to %c
                    peg", n, from, to);
towers(n-1, aux, 1, from);
}.
```

Output:

Datatypes

⇒ Datatypes specify the type of data we enter in our program.

⇒ In C there are some predefined set of data types which are also called as primitive data types.



⇒ Primitive datatypes are fundamental data types

(1) integer type → These datatypes are used to store whole number (ie) number without fraction.

→ Size (4 bytes)

→ Format specifier (%d)

→ Range: -2147483648. to 2147483647

→ int keyword to declare the integer variable.

(2) float datatype → used to store decimal and exponential values.

→ used to store decimal numbers with single precision.

→ Size (4 bytes)
→ format specifier (%d)

(3) Character datatype:
⇒ used to store only a single character.
⇒ Size of the character is 1 byte.
⇒ Range (-128 to 127) or (0 to 255)
⇒ Size (1 byte)
⇒ Format specifier (%c)

(4) void datatype:
→ used to specify that no value is present.
→ It has no value and no operations
→ It is used to represent nothing.
→ It is used in
↳ function returntypes
↳ function arguments
↳ pointers as void.

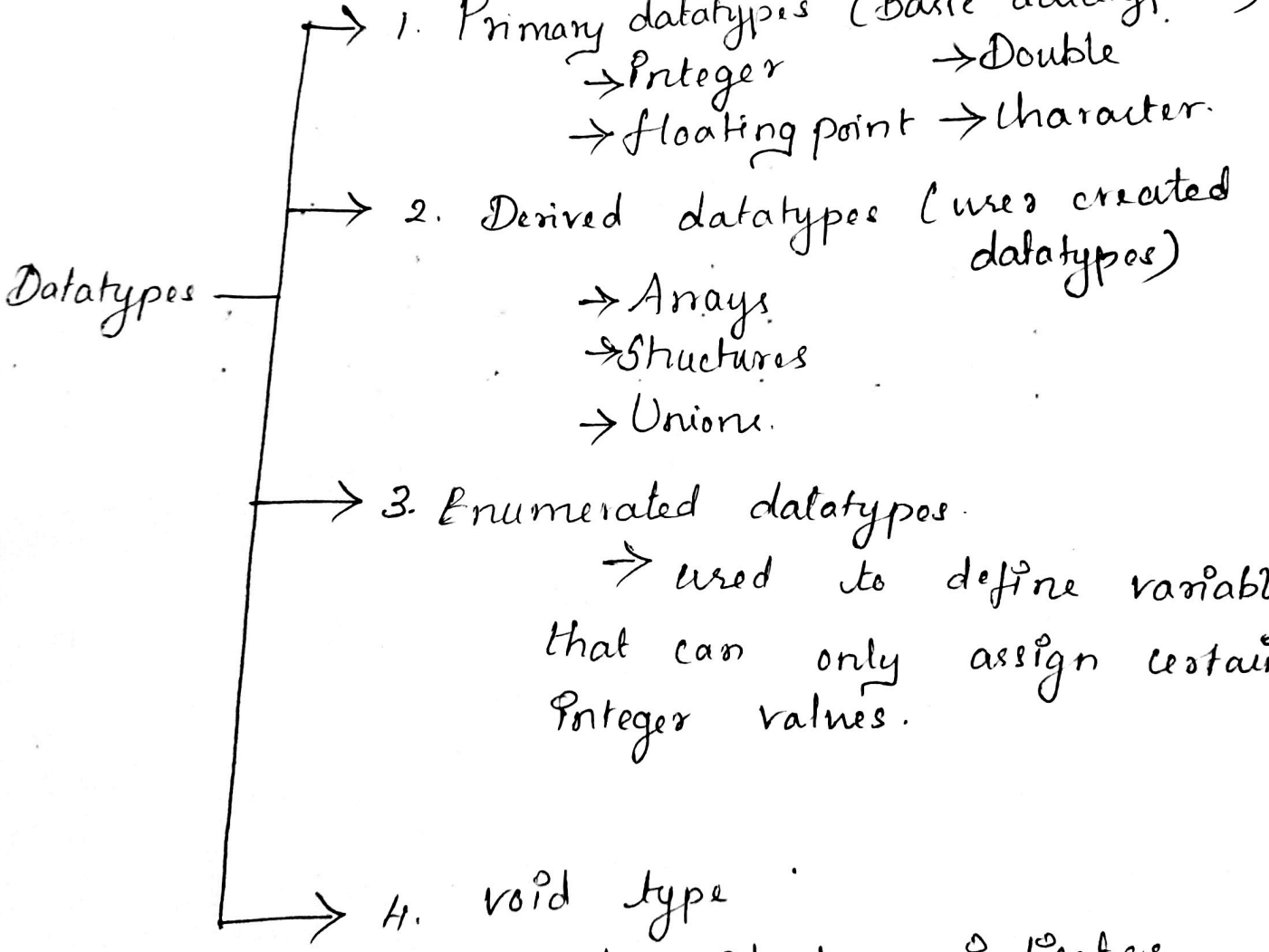
(5) double datatype:
→ used to store fractional numbers containing one or more decimals.
→ Sufficient for storing 15 decimal digits. Size (8 bytes) format specifier (%lf)

→ 5.20 (8 bytes)

format specifier (%lf)

In C, Datatypes are classified as,

1. Primary data types (Basic datatypes)
2. Derived data types (user defined datatypes)
3. Enumeration datatypes
4. void data type.



Basic Datatypes (Primary Datatypes)

- Integer
  - Signed
    - int
    - short int
    - long int
  - Unsigned
    - int
    - short int
    - long int
- Floating point
  - float
  - double
  - long double
- character
  - char
  - signed char
  - Unsigned char

Modifiers in C

- **Short** → Small integer values (2 bytes)
- **long** → Store larger number (8 bytes)
- **signed** → user can store negative and positive values. (int & char)
- **unsigned** → user can store only positive values (int & char)

⇒ An array is defined as finite ordered collection of homogenous (→ similar data type) data stored in contiguous memory locations.

finite – data range defined.
ordered – data stored in contiguous memory locations

Arrays:

⇒ An array is defined as the collection of similar type of data items stored at contiguous memory location.

⇒ Arrays are the derived datatype in c.

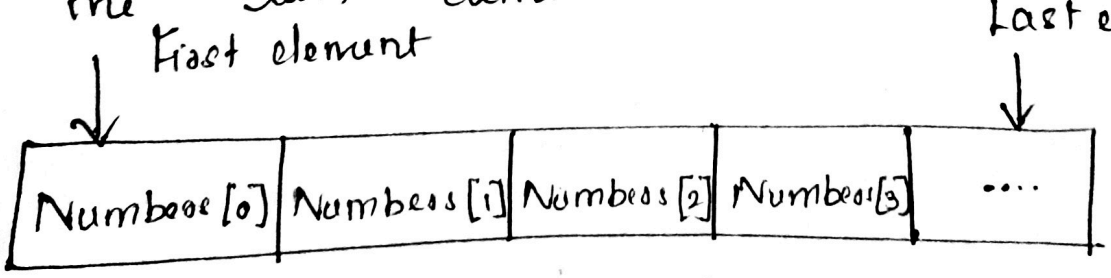
⇒ It stores the primitive type of data such as int, char, double, float etc.

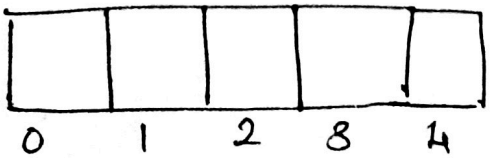
⇒ It also has the capability to store the collection of derived datatypes such as pointers, structures etc.

⇒ Each elemement in an array can be randomly accessed by using its index number.

⇒ Array can also be defined as a collection of variables of the same type.

⇒ The lowest address corresponds to the first element and the highest address to the last element.



⇒ 

int A[5];

array

array length.

Subscript operator.

0 to (size-1)

Advantages:

1. Code optimization → less code to access the data
2. Ease of traversing → Using for loop, the elements of an array are retrieved easily.
3. Ease of sorting → To sort the elements of the array only few lines of code is needed.
4. Random Access → We can access any element randomly using the array.

Disadvantages:

1. Fixed size → Whatever the size of the array is declared, it doesn't exceed the limit.
   → Array does not grow dynamically.

Array - Operations:

1. Traversing → Process each and every element in the array sequentially.
2. Searching → Searching an element to find out whether the element is present or not.
3. Sorting → Arranging the array elements in a particular sequence.
4. Inserting → To insert the element into the array.
5. Deleting → To delete the element from the array.

Types of C Arrays.

⇒ There are two types of C Arrays. They are

1. One dimensional Array.
2. Multi dimensional Array (Two dimensional, Three dimensional Array etc).

One dimensional Array.

⇒ An array that has only one subscript specification that is needed to specify a particular element of an array.

⇒ A one dimensional array is a structured collection of components (often called array elements) that can be accessed individually by specifying the position of a component with a single index value.

Syntax:

data-type arr-name [arr-size];

int A [n];

Rules for declaring one Dimensional Array:

1. An array variable must be declared before being used in a program.

2. The declaration must have a datatype (int, float, char, double etc), variable name and subscript.
3. The subscript represents the size of the array. If the size is declared as 10, programmers can store 10 elements.
4. An array index starts from 0.
5. Each array element is stored in a separate memory location.

Initialization of 1D array:

⇒ An array can be initialized at the following states.

1. At compiling time (static initialization)
2. Dynamic initialization.

1. Static initialization:

⇒ The array of the elements is initialized at the time, the program is written or array declaration.

Syntax:

datatype array-name [array-size] = { list of array elements };

eg.

```
int n[5] = {0, 1, 2, 3, 4};
```

2. Run time initialization:

⇒ Run time initialization means the array can be initialized at run time.

Types of C Arrays.

⇒ There are two types of C Arrays. They are

1. One dimensional Array.
2. Multi dimensional Array (Two dimensional, Three dimensional Array etc).

One dimensional Array:

⇒ An array that has only one subscript specification that is needed to specify a particular element of an array.

⇒ A one dimensional array is a structured collection of components (often called array elements) that can be accessed individually by specifying the position of a component with a single index value.

Syntax:

data-type arr-name [arr-size];

int A [5];

Rules for declaring one Dimensional Array:

1. An array variable must be declared before being used in a program.

⇒ That means array elements are initialized after the compilation of the program.

⇒ An array can be initialized at run time using scanf() function.

⇒ This approach is usually used for initializing large arrays or to initialize arrays with user specified values.

⇒ To input elements in an array, we can use a for loop or insert elements at a specific index.

1. State Initialization

```
#include <stdio.h>
void main()
{ int i;
  int a[3] = {2, 3, 4};
  for (i=0; i<3; i++)
  { printf("%d\t", a[i]);
  }
}
```

⇒ Run time Initialization

```
#include <stdio.h>
void main()
{ int a[5];
  printf("Enter array elements");
  for(int i=0; i<5; i++)
  scanf("%d", &a[i]);
  printf("Array elements are");
  for(int i=0; i<5; i++)
  printf("%d", a[i]);
  int sum=0;
  for(int i=0; i<5; i++)
    sum+=a[i];
  printf("sum=%d", sum);
}
```

⇒ compile time initialization

2. Multidimensional Array:

⇒ Multidimensional arrays allows to store data in a table like format, where each row and column can be accessed using an index.

Syntax:

datatype array_name[size1][size2]...[sizeN];

Accessing elements

int element = arr[1][2];

Initializing multidimensional arrays.

int arr[2][3] = { {1, 2, 3},
{4, 5, 6} };

2D arrays.

int a[3][3].

	col 0	col 1	col 2
row 0	a[0][0]	a[0][1]	a[0][2]
row 1	a[1][0]	a[1][1]	a[1][2]
row 2	a[2][0]	a[2][1]	a[2][2].

```
#include<stdio.h>
int main()
{
int r, c, a[100][100], b[100][100], sum[100][100], i, j;
printf("Enter the number of rows:");
scanf("%d", &r);
printf("Enter the number of columns:");
scanf("%d", &c);
printf("\n Enter elements of 1st matrix:\n");
for (i=0; i<r; i++)
for (j=0; j<c; j++)
{
printf("Enter elements a%d%d", i, j);
scanf("%d", &a[i][j]);
printf("\n Enter elements of 2nd matrix:\n");
for(i=0; i<r; i++)
for(j=0; j<c; j++)
{ printf("Enter elements b%d%d", i, j);
scanf("%d", &b[i][j]);
}
for(i=0; i<r; i++)
for(j=0; j<c; j++)
{ sum[i][j] = a[i][j] + b[i][j];
}
```

```
printf("\n Sum of two matrices : \n");
    for(i=0; i<r; i++)
        for(j=0; j<c; j++)
        {
            printf("%d", sum[i][j]);
        }
    return 0;
}
```

Output :

```
Enter the number of rows : 2
Enter the number of columns : 3
Enter elements of 1st matrix :
Enter element a11 : 2
Enter element a12 : 3
Enter element a13 : 4
Enter element a21 : 5
Enter element a22 : 2
Enter element a23 : 3
Enter element b31 : -4
Enter element b32 : 5
Enter element b33 : 3
Enter element b21 : 5
Enter element b22 : 6
Enter element b23 : 3
Sum of two matrices :
-2  8  7
10  8  6
```

Unit - II

C - Programming - Advanced Features.

Structures - Union - Enumerated data types - Pointers : Pointers to Variables, Arrays and Functions - File handling - Preprocessor directives.

Structures :

⇒ Structure is a collection of variables of different types under a single name.

⇒ To create structure variables its datatype must be defined.

⇒ To define a structure, struct keyword is used.

Syntax :

```
struct structure_name {
    datatype member1;
    datatype member2;
    ....
};
```

(eg)

```
struct employee
{
    char name[20];
    int emp_no;
    float salary;
};
```

⟶ should end with semicolon.

Access Members of a structure:

⇒ There are different kinds of data items in the structure.

⇒ To access any member of a structure, member access operator (.) is used.

⇒ Keyword struct is used at the beginning while defining a structure in C.

⇒ To access the structure a variable is created

⇒ A structure can be defined as a single entity holding variables of different data types that are logically related to each other.

⇒ All the data members inside a structure are accessible to the functions defined outside the structure.

⇒ To access the data members in the main function, we need to create a structure variable.

Syntax with variable.

```
struct structName
{
 datatype1  member_name1;
 datatype2  member_name2;
} struct_var1, struct_var2;
```

→ structure variables at the end of structure definition.

⇒ Structure variables are declared at the end of the structure definition, right before terminating the structure.

```
struct structName
{
datatype1  member_name1;
datatype2  member_name2;
  :
};
int main()
{
struct structName struct_var1, struct_var2;
```

⇒ A compilation error will be thrown when the data members of the structure are initialized inside the structure.

⇒ Structure variables are created to initialize a structure's data member.

⇒ This variable can access all the members of the structure and modify their values.

```
struct rectangle
{
  int length;
  float breadth, float area;
};
int main()
{
struct rectangle rect;
```

structure variable

```
rect.length = 10;
rect.breadth = 6;
```

→ initializing with structure variable

```
rect.area = rect.length * rect.breadth;
printf("\n Area of the rectangle is %f",
                                rect.area);
return 0;
}
```

(or)

```
printf("Enter the length & breadth
                of the rectangle :\n");
scanf("%d", &rect.length);
scanf("%f", &rect.breadth);
```

Note:

⇒ Structure members can be initialised in 3 ways:

1. Using Assignment Operator
2. Using Initializer List.
3. Using Designated Initializer List.

1. Using Assignment Operator

```
struct structure_name str;
str.member1 = value1;
str.member2 = value2;
:
```

2. Using initializer list:

```
struct structure_name str = { value1,
                              value2, value3};
```

3. Using Designated initializer List

```
struct structure_name str = {.member1=
                             value1,
                             .member2 = value2,
                             .member3 = value3};
```

typedef:

⇒ is a keyword that is used to provide existing datatypes with a new name (ie) it is used to redefine the name of already existing data types.

Syntax:

typedef existing-name alias-name;

⇒ It is used for creating a new data using the existing type.

Syntax

typedef data-type name;

10.

Structure using Array:

⇒ It is possible to store a structure that has an array element (ie) (an) array in which each element is a structure.

```
struct structName
{
 type element1;
 type element2;
 .....
 type elementn;
} array name[size];
```

(Eg)

```
struct student
{
 int rollno;
 char name[25];
 float marks;
} stud[100];
```

Example:

```
#include <stdio.h>
#include <conio.h>
void main()
{
 struct student
 {
   int rollno;
   char name[25];
   float marks;
 } stud[100];
 int n, i;
 clrscr();
 printf("Enter total number of students\n");
 scanf("%d", &n);
 for(i=0;i<=n;i++)
 {
 printf("\nEnter the student details %d", i+1);
 printf("Name:\n");
 scanf("%s", &stud[i].name);
 printf("RollNo:\n");
 scanf("%d", &stud[i].rollno);
 printf("Marks:\n");
 scanf("%f", &stud[i].marks);
}
```

```
printf("Student details: \n");
for(i=0; i<=n; i++)
printf("\n Roll Number: %d \n", stud[i].rollno);
printf("\n Name: %s \n", stud[i].name);
printf("\n Marks: %f \n", stud[i].marks);
}
```

Union

⇒ Union is a user created datatype similar to that of structure whereas all the members share a common memory location.

⇒ The size of the union corresponds to the length of the largest member.

⇒ Since the member share a common location, they have the same starting address.

⇒ The real purpose of Union is to prevent memory fragmentation for arranging for standard size of data in the memory.

```
Union  Union-name
{
  type  element1;
  type  element2;
  ..... .....
  type  elementn;
};
```

⇒ Variables are then declared as

```
Union  Union-name  x, y, z;
```

(eg)

```
union  student
{
  int  rollno;
  float  marks;
};
union student stud;
```

⇒ Also the below format can be used.

```
union  union-name
{
type  element1;
type  element2;
.....
type  elementn;
} var1, var2, var3;
```

```
union  student
{
 int  rollno;
 float  marks;
} x, y, z;
```

Program:

```
#include<stdio.h>
void main()
{
union data
{
  int a;
  char b;
 float c;
} var;
printf("\n size of (var) is %d", sizeof(var));
printf("\n size of (var.a) is %d", sizeof (var.a));
printf("\n size of (var.b) is %d", sizeof (var.b));
printf("\n sizeof (var.c) is %d", sizeof (var.c));

var.a = 10;
printf("\n value of var.a is %d", var.a);
var.b = 'b';
printf("\n value of var.b is %c", var.b);
var.c = 15.52;
printf("\n value of var.c is %f", var.c);

}.
```

	Structure	Union
1.	The amount of memory required to store the structure variable is sum of size of all its members.	The amount of memory required is equal to the memory required by the largest member.
2.	Each member have their own memory space.	one block is used by all the member of the union.
3.	keyword struct defines a structure.	keyword union defines a union.

4.	struct stud
	{
	int rollno;
	float marks;
	};
	}
		union stud
		{
		int rollno;
		float marks;
		};
		}

5.	Any member can be retrieved at any time.	While retrieving data the type that is retrieved must be the type recently used
6.	More members can be initialized at once.	A union can be initialized with type of first member.

Enumerated data types

⇒ used to represent group of constants which means unchangeable values.

⇒ keyword enum is used. to represent enumerated data types.

Syntax

⇒

enum enum_name { List of enum items separated by commas };

(eg)

enum Level { LOW, MEDIUM, HIGH };

⇒ To access the enum we need to create a variable of it.

(eg)

```
enum Level { LOW, MEDIUM, HIGH };
enum Level myvar;
enum Level myvar = MEDIUM;
```

⇒ The 1st item has the value 0,
2nd item has the value 1
3rd item has the value 2 and

so on.

Program:1

```
#include<stdio.h>
enum Level{LOW, MEDIUM, HIGH};
int main()
{
enum Level myvar = MEDIUM;
printf("%d", myvar);
return 0;
}
```

O/P → 1.

Program2:

```
#include <stdio.h>
enum level {LOW, MEDIUM, HIGH};
                =25      =50    =75
int main()
{
  printf("%d", myvar);
}.
```

O/P

50.

Pointers:

⇒ A pointer is a variable that stores the memory address of another variable as its value.

⇒ A pointer is a variable created with * operator.

Program:

```
void main()
{
int myage = 43; int *ptr = &myage;
printf("%d", myage); ⇒ 43
printf("%p\n", ptr); ⇒ address
printf("%p\n", &myage); ⇒ address.
printf("%d\n", *ptr); ⇒ 50
}
```

pointer variable in declaration

↳ dereference operator.

& → reference operator

* → dereference operator.

O/P

43
0x7ffe5367e044
0x7ffe5367e044
43.

Arrays and functions

⇒ In C programming we can pass entire arrays to functions.

⇒ Passing array elements to a function is similar to passing variables to a function

```
#include <stdio.h>
void display (int age1, int age2)
{
 printf("%d\n", age1);
 printf("%d\n", age2);
}

 int main()
 {
 int age[] = { 2, 8, 4, 12};

 display (age[1], age[2]);

 return 0;

 }
```

O/P:

8
4.

File handling

Files:

A file can be classified into two types based on the way the file stores the data.

1. Text files
2. Binary files.

Text files:

⇒ contains the data in the form of ASCII characters and is generally used to store a stream of characters.

⇒ Each line in a text file ends with a new line character ('\n')

⇒ It can be read or written by any text editor.

⇒ They are generally stored with .txt file extension

⇒ Text files can also be used to store the source code.

Binary Files:

⇒ A binary file contains data in binary form. ie 0's and 1's.

⇒ Binary files can be created only from within a program and their contents can only be read by a program

⇒ More secure as they are not easily readable.

⇒ They are generally stored with .bin file extension.

C File operations

1. Creating a new file. fopen() with attributes as "a" or "a+" "w" or "w+"
2. opening an existing file fopen()
3. Reading from a file fscanf or fgets
4. writing to a file fprintf or fputs
5. Moving to a specific location in a file. fseek(), rewind()
6. Closing a file. fclose()

fopen() → open a file.

File pointer:

⇒ reference to a particular position in the opened file.

⇒ FILE macro is used to declare the file pointer variable.

⇒ FILE macro is defined inside <stdio.h> header file.

Syntax

```
FILE * filepointer_name;
```

fopen() function contains

⇒ filename

⇒ access mode.

opening modes:

⇒ r — returns null if not opened.
⇒ rb — binary mode
⇒ w
⇒ wb — write in binary mode
⇒ a
⇒ ab
⇒ r+
⇒ rb+
⇒ w+
⇒ wb+
⇒ a+
⇒ ab+

fscanf() ⇒ use formatted string & variable argument list to take input from a file.

fgets() ⇒ Input the whole line from the file

fgetc() ⇒ Reads a single character from the file.

fread() ⇒ reads the specified bytes of data from a binary file.

fgetw() ⇒ Reads a number from a file.

fprintf() ⇒ Formatted string and variable argument list to print output to the file.

fputs ⇒ prints the whole line in the file and a newline at the end.

fputc ⇒ prints a single character into the file.

fputw ⇒ prints a number to a file.

fwrite ⇒ write the specified amount of bytes to the binary file.

Need for file handling in C:

⇒ Reusability.

⇒ Larger storage capacity

⇒ Saves time

⇒ Portability

File Handling :

⇒ In C, we can create, open and read and write to files by declaring a pointer of type FILE and use fopen() function.

FILE *fptr ; ⟶ pointer variable.
↳ data type

fptr = fopen(filename, mode);
↳ function to open a file
eg filename.txt

w - write to file
a - Append
r - Read from file.

Create a file :

⇒ use 'w' mode in fopen() function.

```
FILE *fptr;
fptr = fopen("filename.txt", "w");
fclose(fptr);
```

↳ Closing the file.

↳ Makes sure that the
⇒ changes are saved properly.
⇒ other programs can use the file
⇒ clean up unnecessary memory space.

Write to a file:

⇒ To insert content to a file, you can use fprintf() function. and add the pointer variable fptr.

Eg.

```
FILE *fptr;
fptr = fopen("filename.txt", "w");
fprintf("fptr, "text");
fclose(fptr);
```

```
fprintf(fptr, "Hello");
```

Append content to a file:

⇒

```
FILE *fptr;
fptr = fopen("filename.txt", "a");
fprintf(fptr, "Hi everyone!");
fclose(fptr);
```

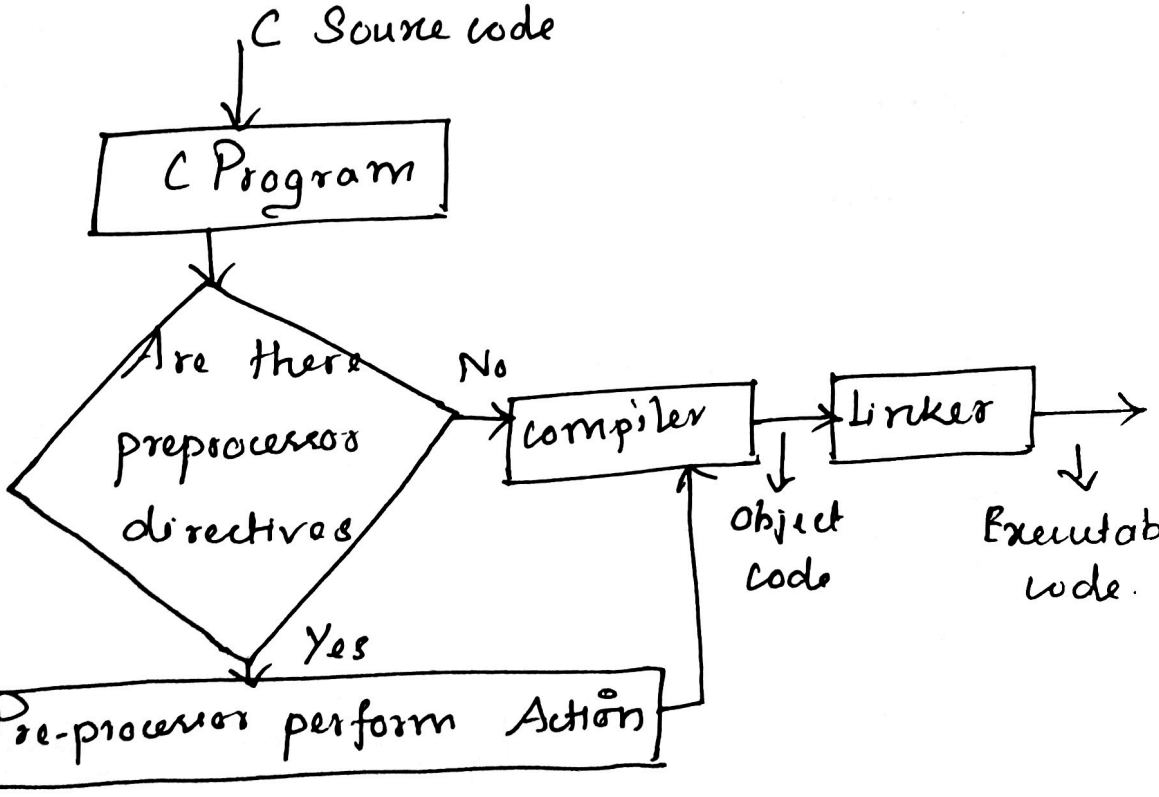
```
#include <stdio.h>
int main()
{
char name[50];
int marks, i, num;
printf(" Enter number of students:");
scanf("%d", &num);
FILE *fptr;
fptr = (fopen("C:\\student.txt", "a"));
if (fptr==NULL)
{
printf("Error!");
exit(1);
}
for(i=0; i<num; i++);
{
printf(" Enter name: %d", i+1);
scanf("%s", name);
printf(" Enter marks:");
scanf("%d", &marks);
fprintf(fptr," \n Name: %s \n
                Marks = %d \n", name, marks);
}
```

```
    fclose(fptr);
    return 0;
}
```

Preprocessor directives.

Preprocessor directives in C are text processing tool or text substitution tool.

⇒ All the preprocessor directives begins with # symbol.



⇒ There are mainly four types of preprocessor directives.

1. Macros:

⇒ replaces every identifier with a string.

(eg)

```
#define PI 3.1415
```

2. File Inclusion:

⇒ #include directive is used for file inclusion.

(eg)

```
#include <file name>
```

3. Conditional Compilation:

⇒ switch ON or OFF of a particular line or group of lines in a progoam. The ON or OFF works only for that line of code.

It can also be used to skip any part of the code as needed.

(eg) #if, #else, #endif etc.

```
#ifdef Age
printf("Age is %d", Age);
#endif
```

4. Line control:

⇒ provide information about the errors or mistakes in the program.

(eg) line-number "file-name"

line line-number "file-name"

undef ⇒ remove definitions related to a macro

#ifdef ⇒ check if a macro is defined or not.

ifndef ⇒ check if #define is defined or not

#if ⇒ evaluating conditions/ expression.

#else ⇒ execute the code if the

#elif ⇒ condition satisfied

#endif ⇒

if exps
// code
else // else code
#endif

#error ⇒ show an error

pragma ⇒ present extra information to the compiler.

Unit - 3

Linear Data Structures

Abstract Data Types (ADTs) - List ADT - Array-Based Implementation - Linked list - Doubly Linked list - Circular Linked list - Stack ADT Implementation of stack - Applications - Queue ADT - Priority Queues - Queue Implementation - Applications.

Abstract Data Types (ADTs):

⇒ ADT is the mathematical specification of the data, a list of operations that can be carried out includes the specification of what it does but excludes the specification of how it does.

⇒ Operations on set ADT

- ↳ Union
- ↳ Intersection
- ↳ Size
- ↳ Complement.

⇒ ADT is an extension of the modular design. The basic idea is that

the implementation of these operations is written once in the program. and any other part of the program that needs to perform an operation on the ADT, can do so by calling the appropriate function.

⇒ Examples of ADT,

- ↳ Stack
- ↳ Queue
- ↳ List
- ↳ Trees
- ↳ Heap
- ↳ Graph.

Benefits of using ADT

⇒ code is easier to understand.

⇒ Provides modularity and reusability.

⇒ Implementation of ADTs can be changed without requiring changes to the program that uses the ADTs.

List ADT:

⇒ List is a linear collection of ordered elements.

⇒ General form of the list size N is $A_0, A_1 \ldots A_{N-1}$.

Where A_1 – First element.

⇒ If the element at position 'i' is A_i then its successor is A_{i+1} and its predecessor is A_{i-1}.

Operations performed on List ADT

⇒ Insert
⇒ Delete
⇒ Find
⇒ Next
⇒ Previous
⇒ PrintList
⇒ MakeEmpty.

Implementation of List ADT:

⇒ Array Implementation
⇒ Linked List Implementation.

Array Based Implementation:

⇒ An array is a collection of homogeneous data element described by a single name.

⇒ Each element of an array is referenced by subscript or index.

⇒ In array implementation, elements

of list are stored in continuous cells of an array.

Advantages:

⇒ Searching an array for the individual element can be very efficient.

⇒ Fast, random access of elements.

Limitations:

⇒ Maximum size must be known in advance, even in case of dynamically allocated.

⇒ The size of an array cannot be changed after its declaration, the size is fixed.

⇒ Data are stored in continuous memory blocks.

⇒ Running time for insertion and deletion is slow.

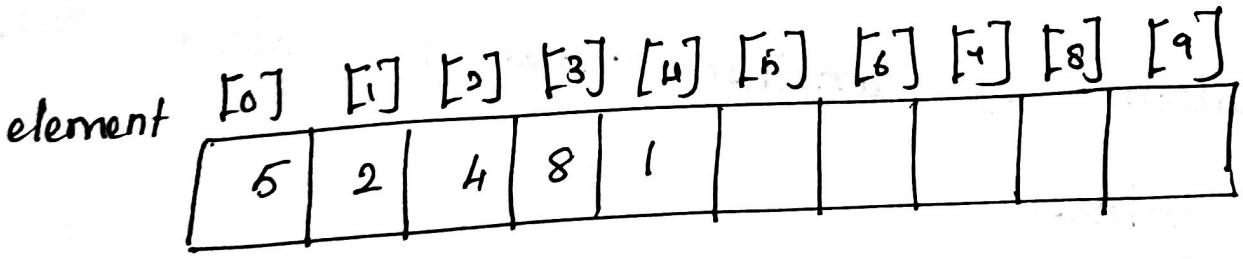
⇒ Memory is wasted, as the memory remain allocated to the array throughout the program execution even few nodes are stored.

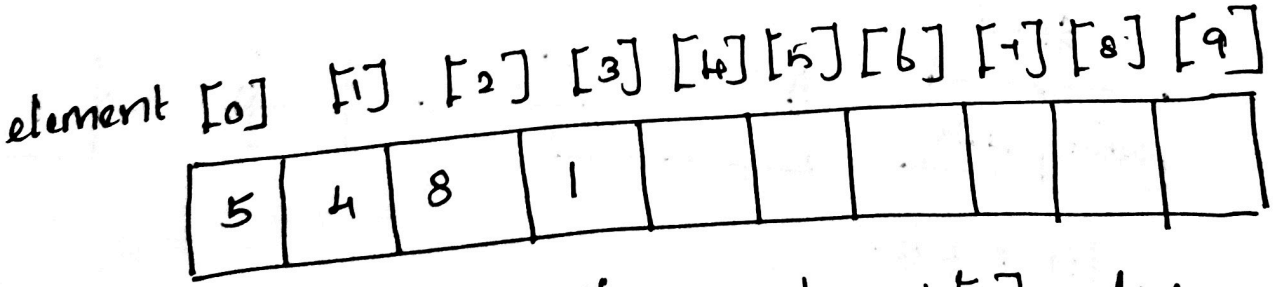
Program:

creation of List.

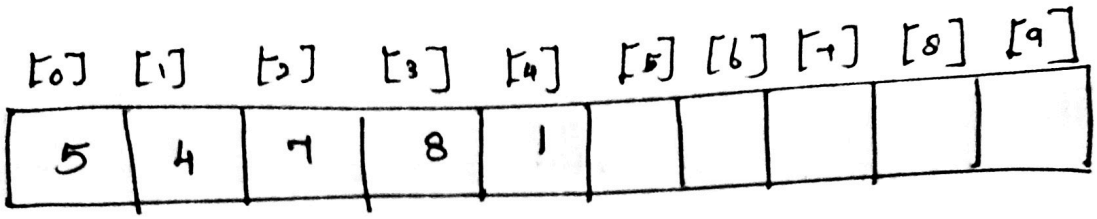
```
void create()
{
int i, n, list[];
printf("\n Enter the number of
        elements to be added in the
        list:\t");
scanf("%d", &n);
printf("\n Enter the array elements:\t");
for(i=0; i<n; i++)
scanf("%d", &list[i]);
}
```

Insertion and Deletion of Linear list.





2 removed from element [1], list size = 4.



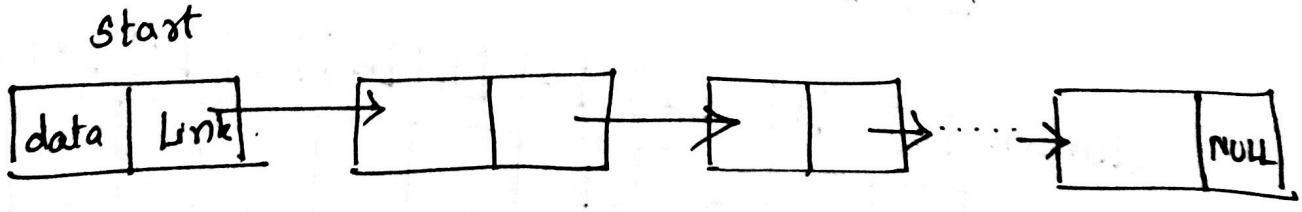
7 inserted in element [2]. list size = 5.

Linked List:

⇒ In a linked list representation, each element of an instance of a data object is represented in a cell or node.

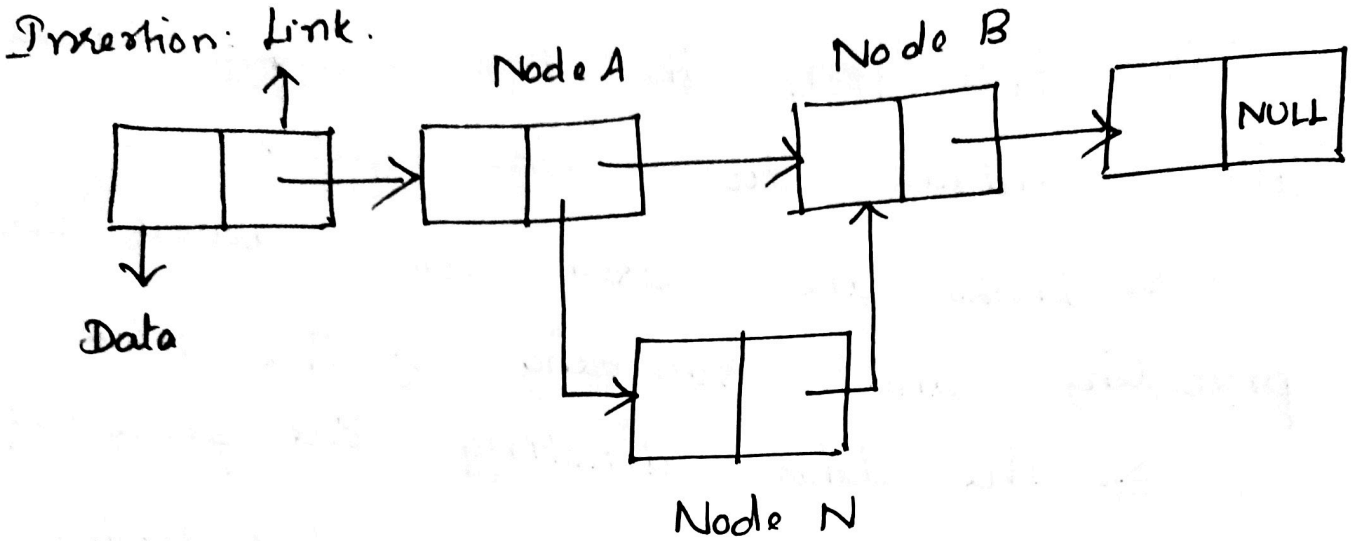
⇒ Each node keeps explicit information about the location of other relevant nodes.

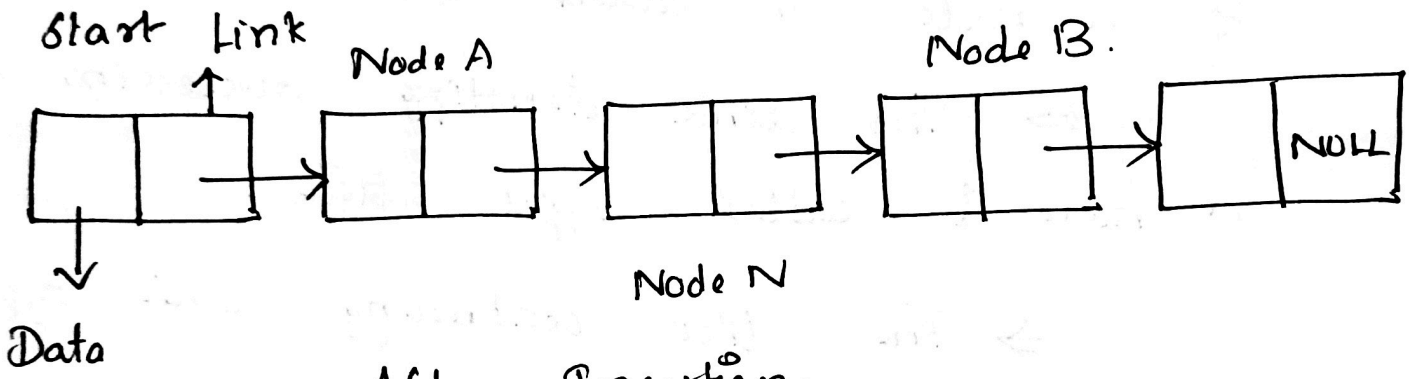
⇒ The explicit information about the location of another node is called link or pointer.



First node.

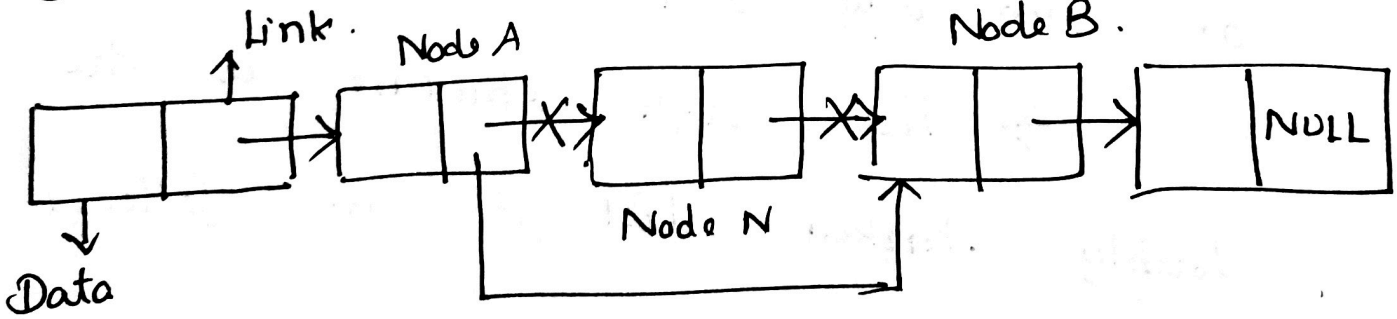
Insertion and Deletion of a singly Linked List.

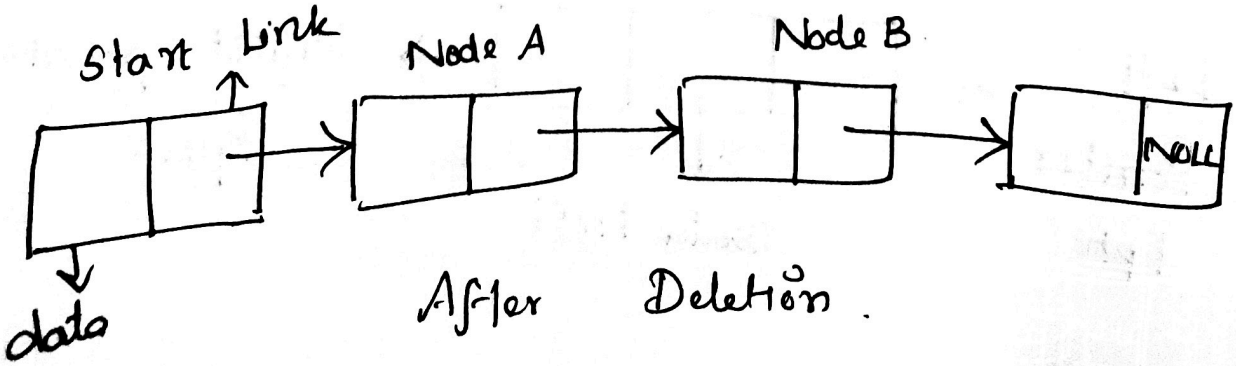




After Insertion.

Deletion :





After Deletion.

Doubly Linked List:

⇒ Each node in the doubly linked list contain two link fields.

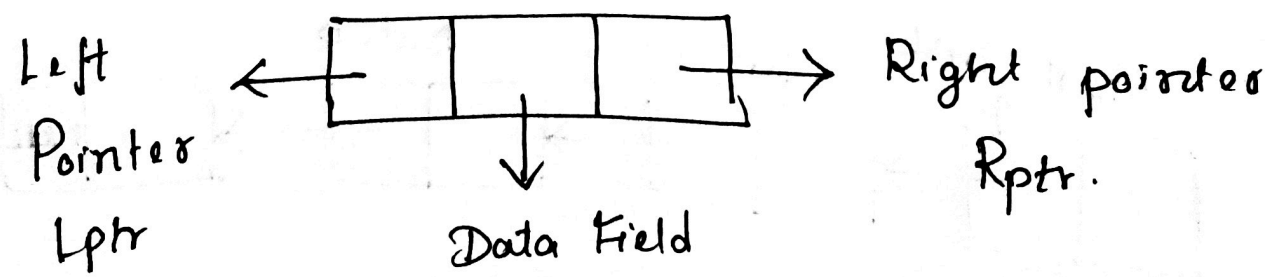
⇒ Links are used to denote the proceeding and succeeding of the node.

⇒ The link denoting the proceeding of a node is called the left link.

⇒ The link denoting succeeding of a node is called right link.

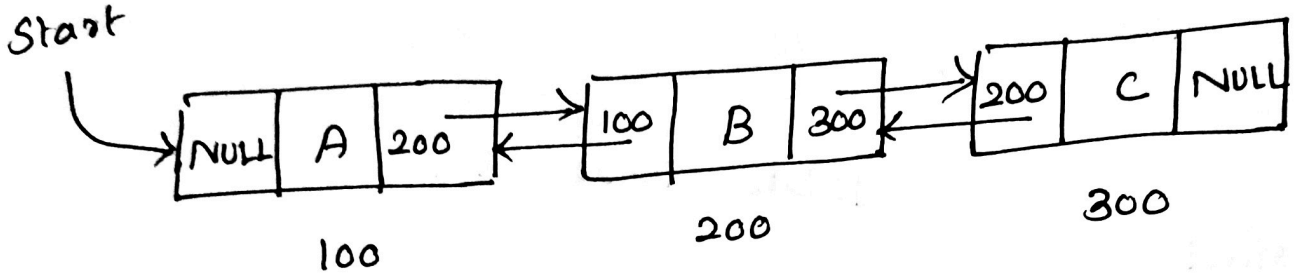
⇒ The list containing this type of node is called a Doubly Linked list or Two way list.

⇒ The node structure in the doubly linked list is as follows.

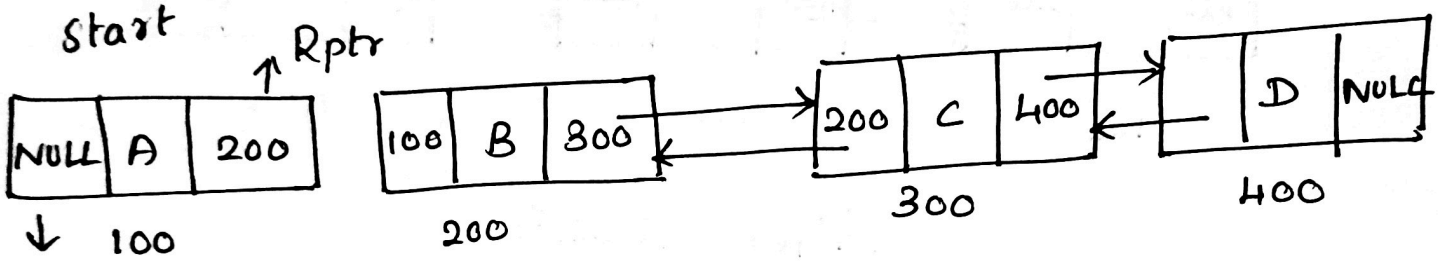


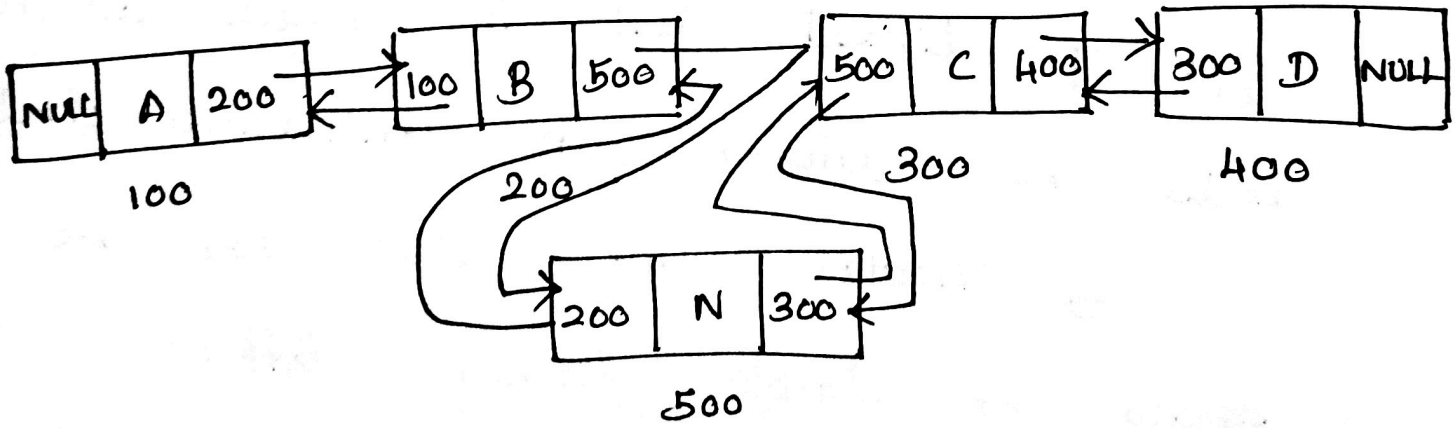
Lptr → contains the address of the before node.

Rptr → contains the address of the next node.



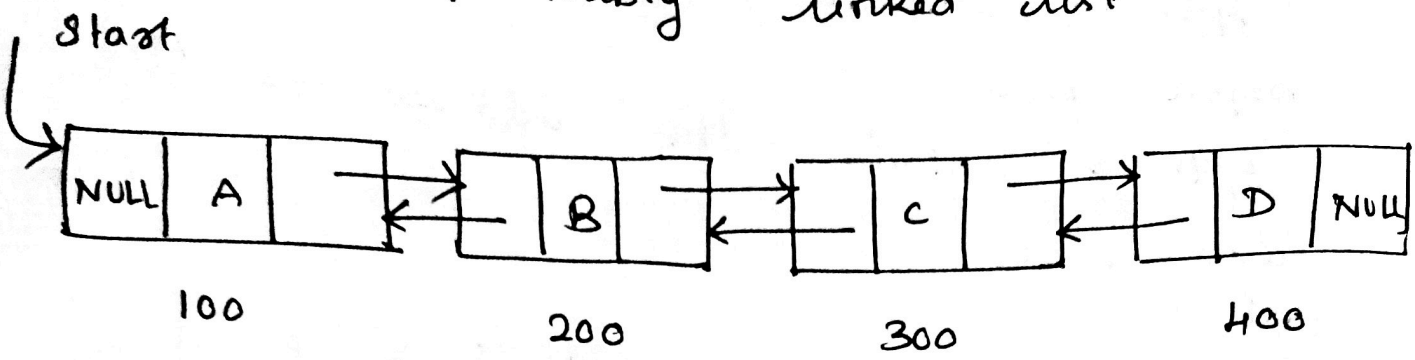
Insertion.



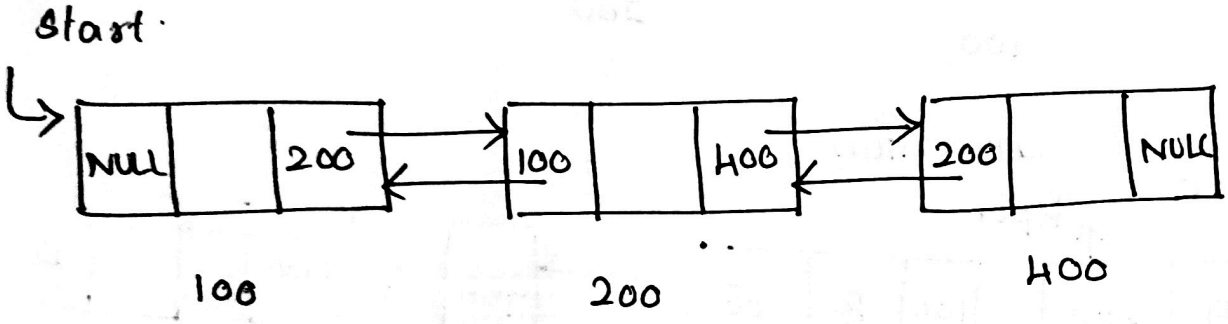


After Insertion.

Deletion of doubly linked list.



Delete Node C.



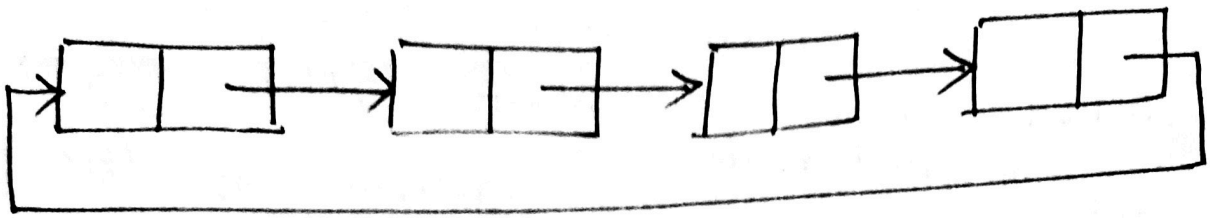
After deletion.

Circular Linked List:

⇒ Circular Linked list is a special type of linked list in which all the nodes are linked in continuous circle.

⇒ Circular linked list can be singly or doubly linked list.

Circular Singly Linked List :



⇒ Circular linked list are the one in which the link fields of the last node of the list contain the address of the first node instead of a null pointer.

Advantages

⇒ Every node is accessibly from a given node.

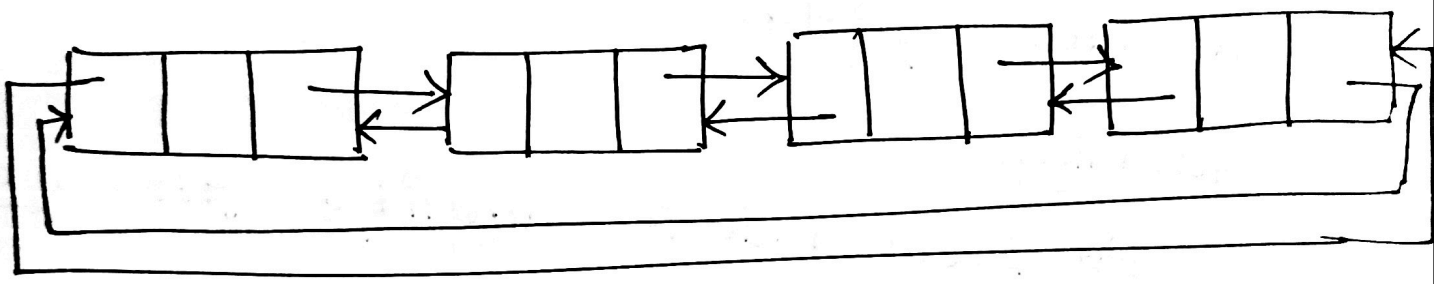
⇒ Certain operations like concatenation and splitting becomes more efficient in a circular list.

Disadvantage :

⇒ Without some conditions in processing it is possible to get into an infinite loop.

Circular doubly linked List:

⇒ These are one type of doubly linked list in which the right pointer field of last node contain the address of the first node and the left pointer field of the first node contains the address of the last node of the list instead of containing null pointer.



Circular doubly linked list.

Advantages:

⇒ Circular linked lists are frequently used instead of ordinary linked list because in circular list all nodes contained a valid address.

⇒ Every node is accesible from a given node.

Linked list:

⇒ is a linear data structure that includes a series of connected nodes. Here each node stores the data and the address of the next node.

① First node → given the special name called head

② last node → is identified because its next portion points to null.

next

Types:

1) Singly linked list
2) doubly linked list
3) circular linked list.

Representation of Linked list:

Each node consists of

⇒ a data item

⇒ Address of another node.

⇒ Both data item and the next node reference are wrapped in the structure

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

Linked List Applications

1) Dynamic memory allocation
2) Implementation in stack and queue.
3) Hash tables, graphs.

Operations

⇒ Insertion
⇒ Deletion
⇒ Traversal
⇒ Search

Insertion:

1) Insert at the beginning
2) Insert at the end
3) Insert at the middle or after a node.

(1) Insert at the beginning:

1) Allocate memory for new node
2) Store data

3) change next of new node to point to head.

4) change head to point to recently created node.

```
struct node * newnode;
newnode = malloc (sizeof (struct node));
newnode -> data = 30;
newnode -> next = head;
head = newnode;
```

(2) Insert at the end:

1) Allocate memory for new node
2) Store data
3) Traverse to last node
4) change next of last node to recently created node.

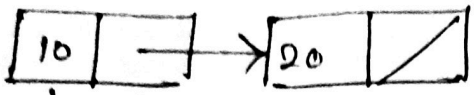
```
struct node * newnode;
newnode = malloc (sizeof (struct node));
newnode -> data = 110;
newnode -> next = NULL;
struct node * temp = head;
while (temp -> next != NULL)
{ temp = temp -> next; }
temp -> next = newnode;
```

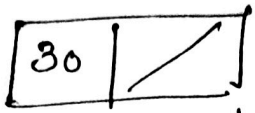
3) Insert after a node:

```
struct node * newnode;
newnode = malloc (sizeof (struct node));
newnode -> data = 50;
struct node * temp = head;
if (head == NULL)
{
head = newnode;
}
else
{
printf("\n Enter the key value:");
scanf("%d", &key);
struct node * temp = head;
do
{
if (temp -> data == key)
{
newnode -> next = temp -> next;
temp -> next = newnode;
}
else
    temp = temp -> next;
} while (temp != NULL);
}
```

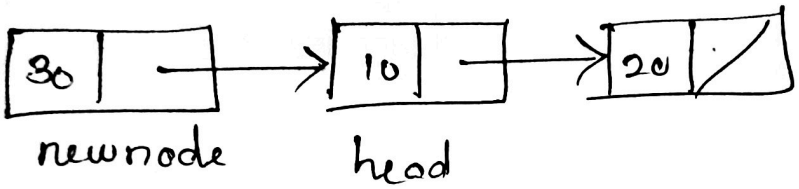
Insert at the beginning.

consider the list



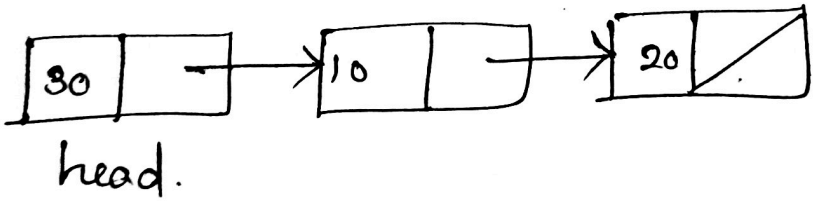
To insert node  in the beginning

① newnode → next = head



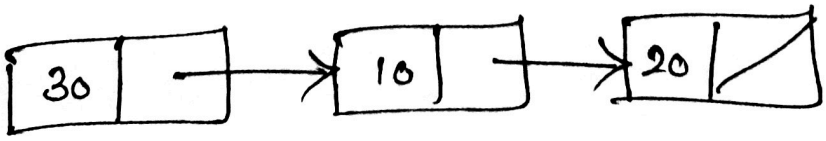
~~newnode = head~~ ;

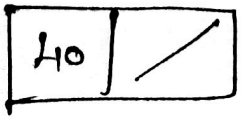
② head = newnode ;

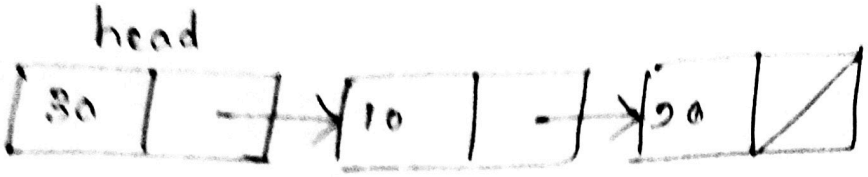


Insert at the end.

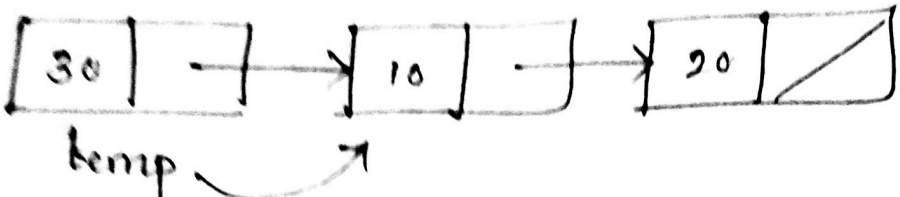
consider the list



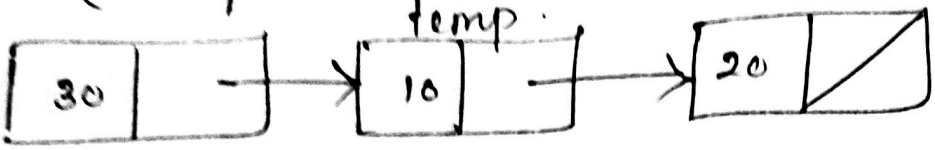
To insert node  in the end.



struct node *temp = head;



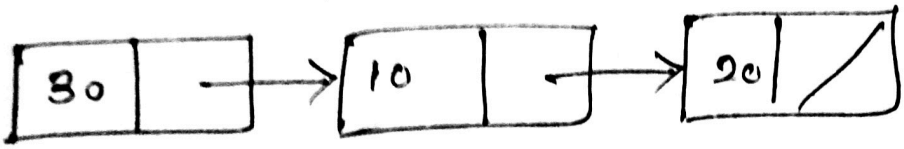
(temp → next != NULL) ⇒ true



↑ temp = temp → next

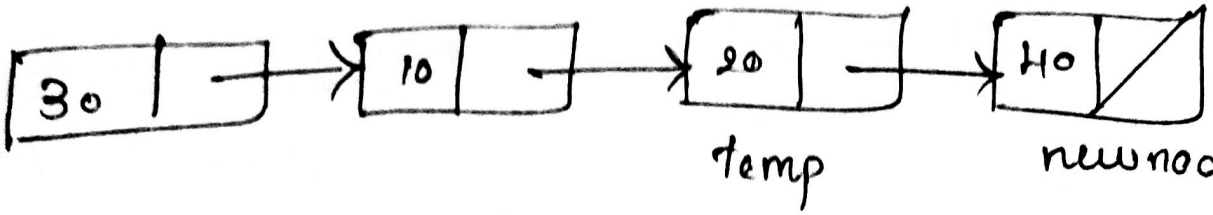
Again temp → next != NULL ; ⇒ True

temp = temp → next;



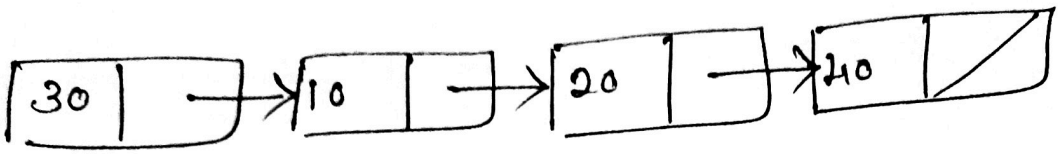
Again temp → next != NULL ; ⇒ False

So, temp → next = newnode

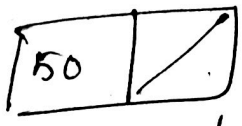


Insert after a node

consider the list

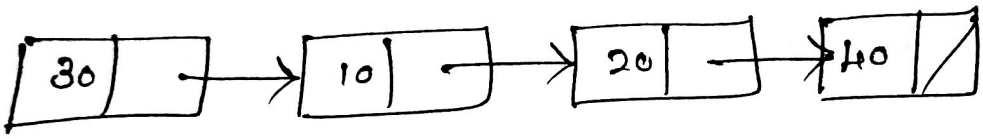


Let key = 20

To insert node 

newnode.

struct node * temp = head



head

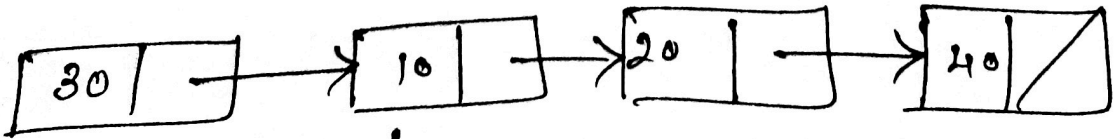
temp

20

if (temp → data == key) ⇒ no

temp = temp → next ;

(temp != NULL) True.

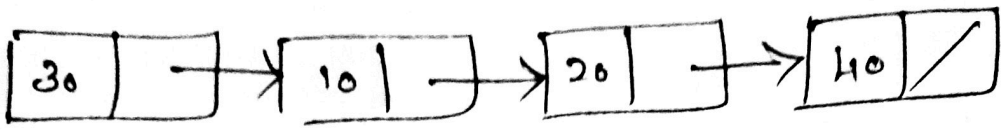


temp

if (temp → data == key) ⇒ no

temp = temp → next ;

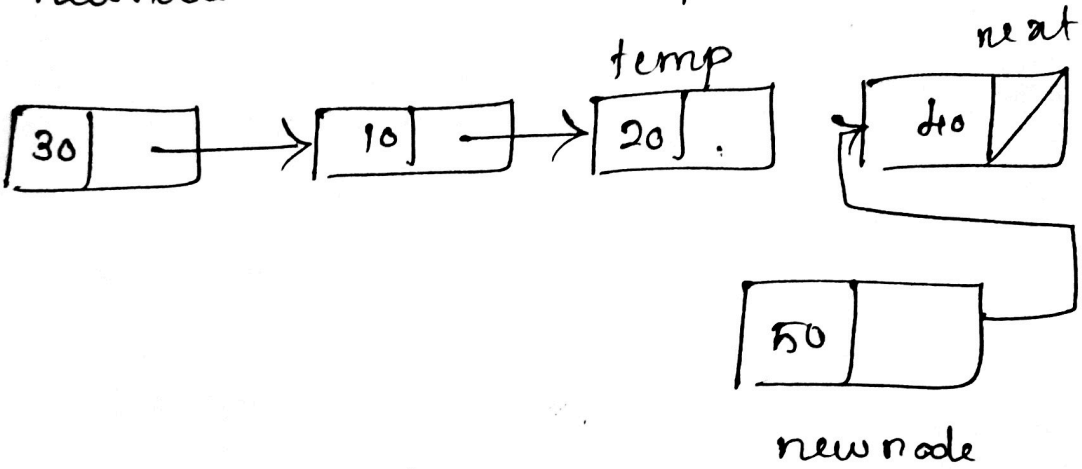
(temp != NULL) true



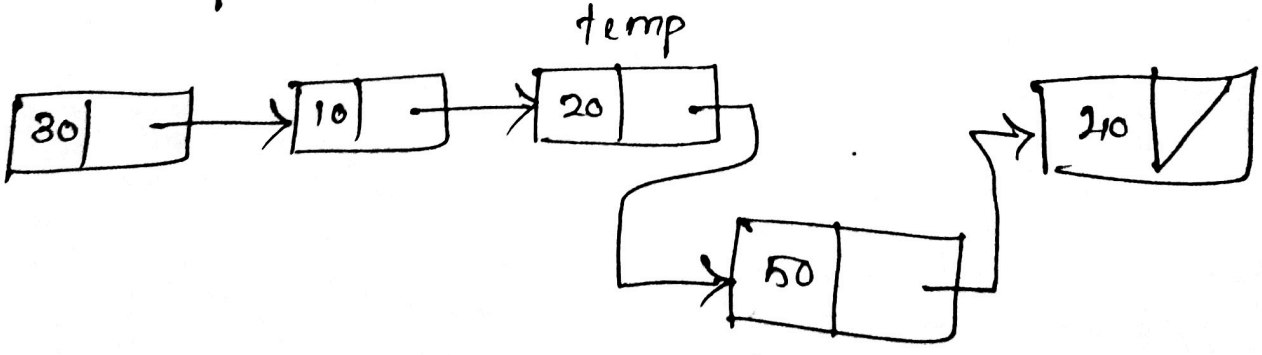
temp

if (temp → data == key) ⇒ yes.

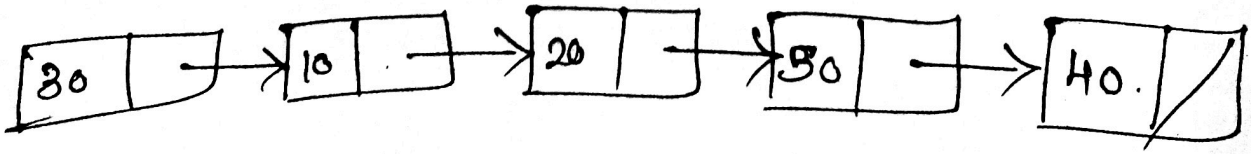
newnode → next = temp → next ;



. temp → next = newnode



Now the list becomes,



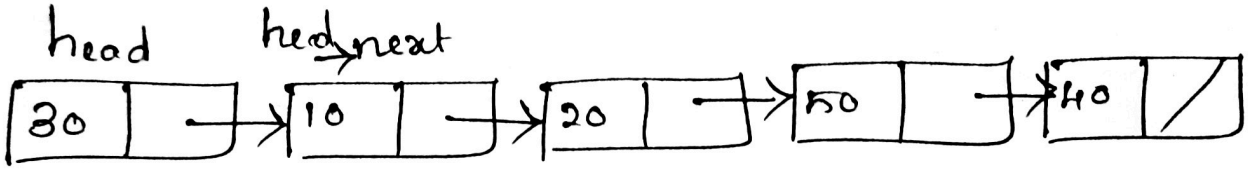
Deletion from a linked list

1. Delete from the beginning
2. Delete from the end.
3. Delete from position / after a node.

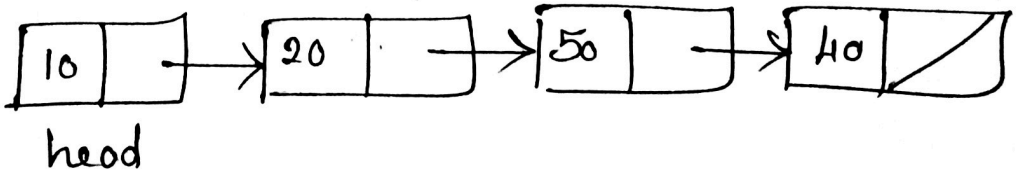
1) Delete from beginning:

head = head → next

consider the list,



head = head → next;

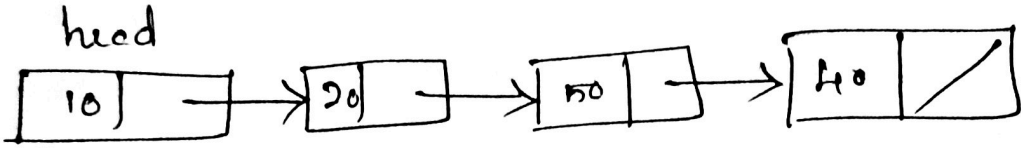


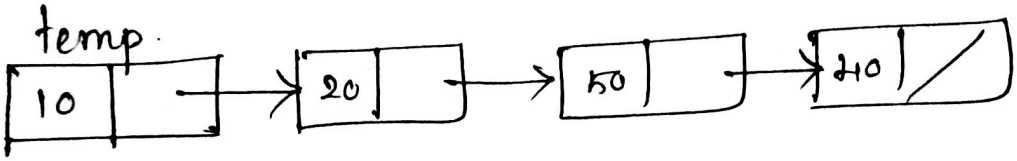
2) Delete from end.

1) Traverse to record last element
2) change its next pointer to null.

```
struct node * temp = head;
while ( temp -> next -> next != NULL)
{
  temp = temp -> next;
}
temp -> next = NULL;
```

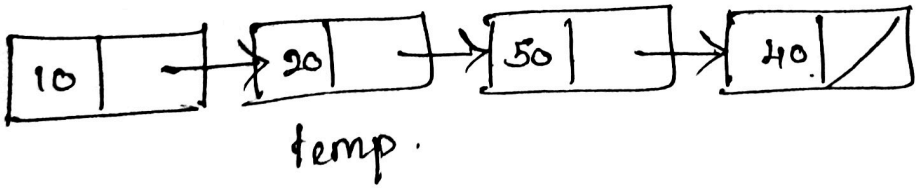
consider the list,



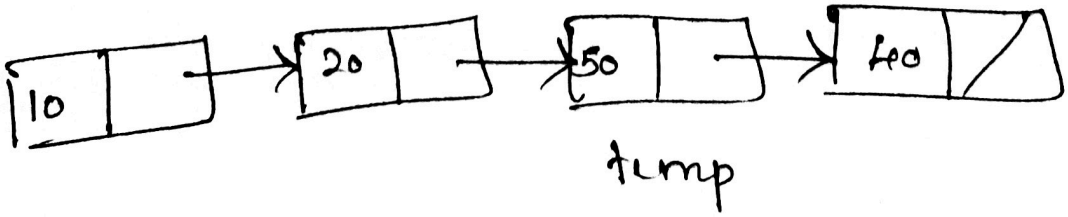


temp -> next -> next != NULL => True.

temp = temp -> next;



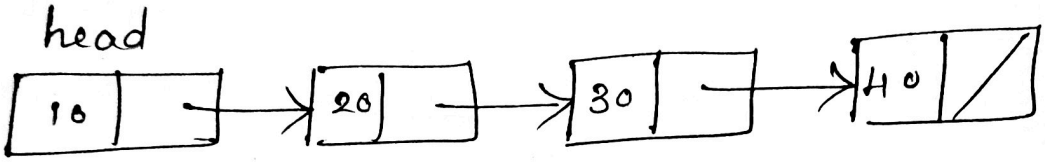
temp -> next -> next != NULL => True



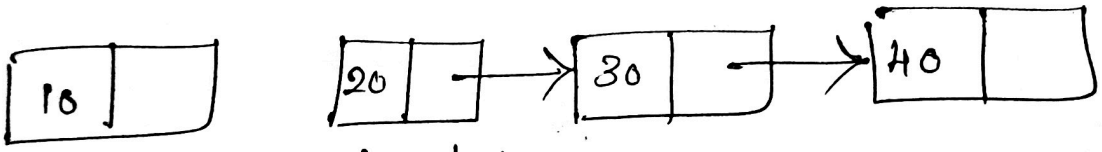
temp = temp -> next;

```
        temp -> next = temp -> next -> next;
    }
    else
        temp = temp -> next;
} }
```

consider the list



head -> data == key



temp

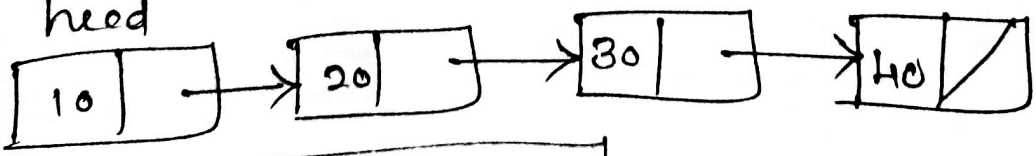
temp = head

head

head = head -> next

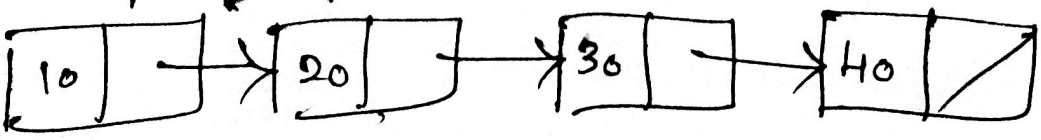
else

let the list be

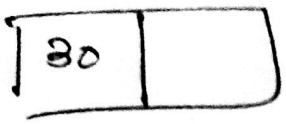


temp

temp = head

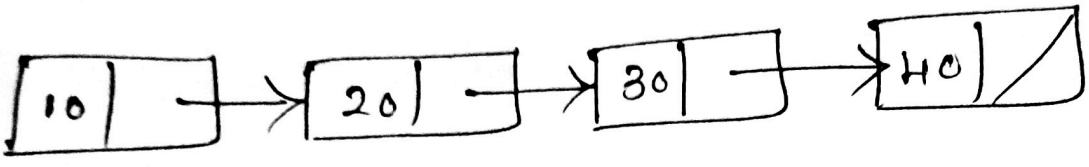


consider the node to be deleted is



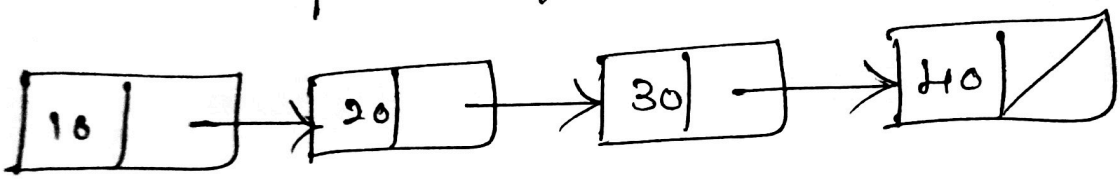
temp → next != NULL ⇒ True

temp → next → data == key ⇒ 20 ⇒ No.



temp

temp = temp → next;



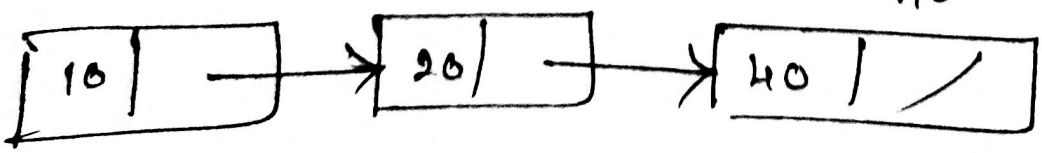
temp

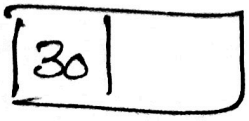
temp → next != NULL ⇒ True.

temp → next → data == key ⇒ 30 ⇒ Yes.

temp → next = temp → next → next

20 40



so node 

gets deleted.

Doubly linked list:

⇒ Each node in linked list consists of 3 components or fields.

⇒ prev - address of the previous node

⇒ data - data item

⇒ next - address of next node.

Representation of single node in DLL

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

Insertion on DLL

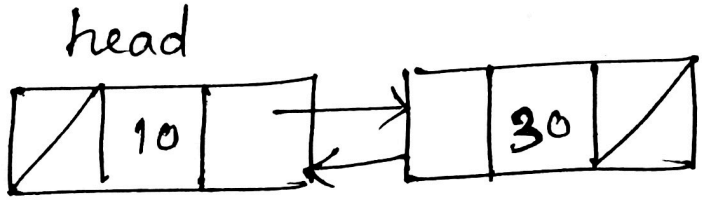
1) Insertion at the beginning.
2) Insertion in between nodes
3) Insertion at the end

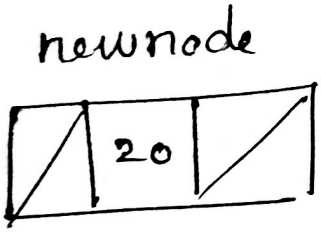
Insertion at the beginning.

```
struct node * newnode;
newnode = malloc(sizeof(struct node));
newnode → data = 20;
newnode → next = head;
```

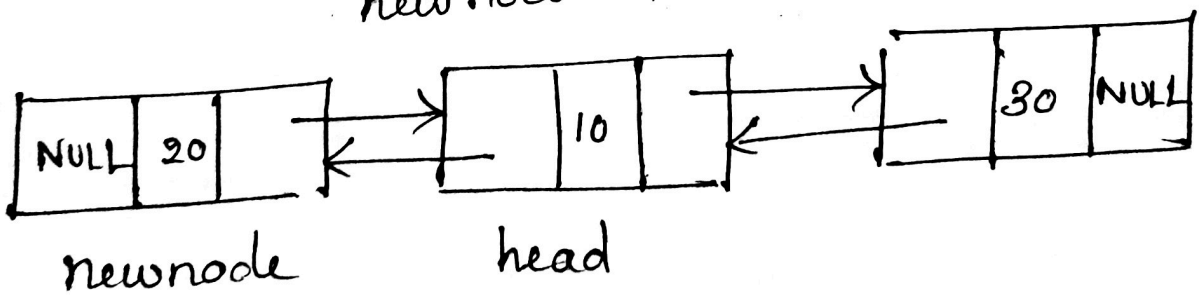
John

```
newnode -> prev = NULL;
if (head != NULL)
head -> prev = newnode;
head = newnode;
```

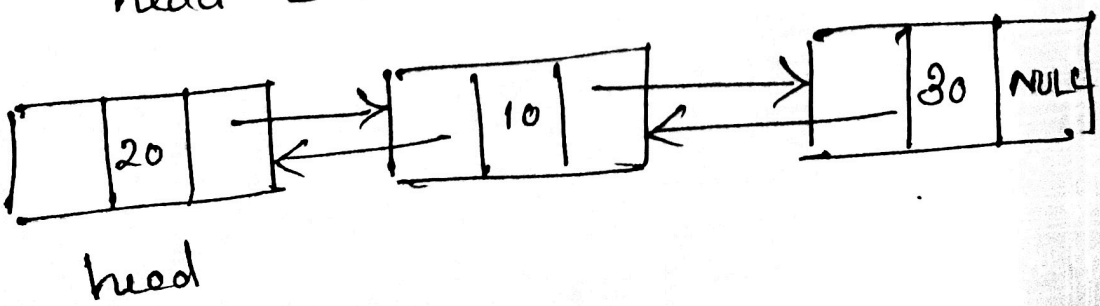




newnode -> next = head;



head = newnode



Stack ADT:

⇒ Stack is an important data structure which stores its elements in an ordered manner.

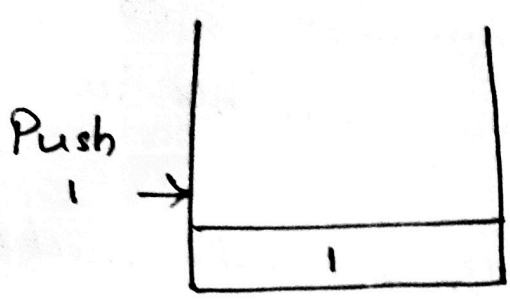
⇒ A stack is a linear data structure in which elements in a stack are added and removed only from one end which is called the TOP.

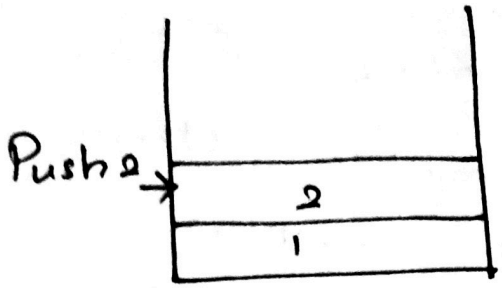
⇒ Hence a stack is called a LIFO (Last In first Out) data structure as the element that was inserted last is the first one to be taken out.

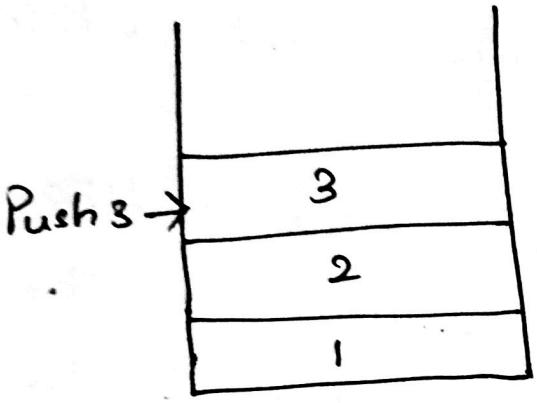
Array Implementation of stacks.

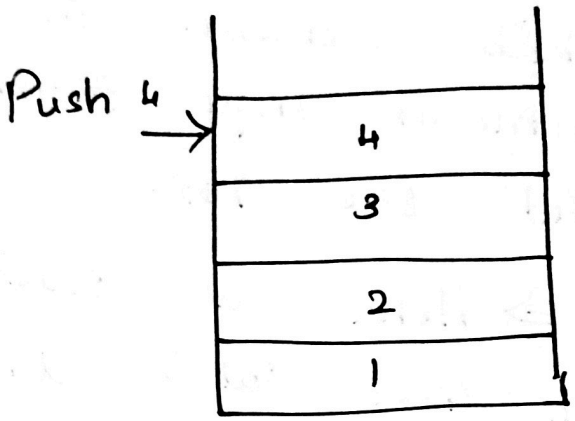
⇒ It contains only one pointer top pointer pointer pointing to the topmost element of the stack.

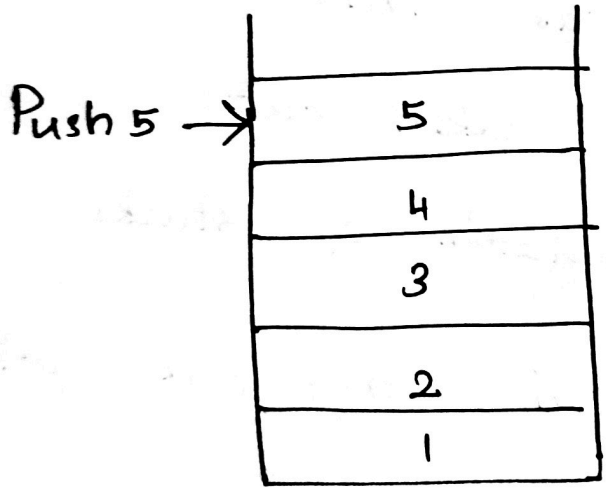
⇒ It can store elements of limited size.











Operations on stack:

- ⇒ Push()
- ⇒ pop()
- ⇒ IsEmpty()
- ⇒ IsFull()
- ⇒ Peek()
- ⇒ count()
- ⇒ change()
- ⇒ display()

Push operation:

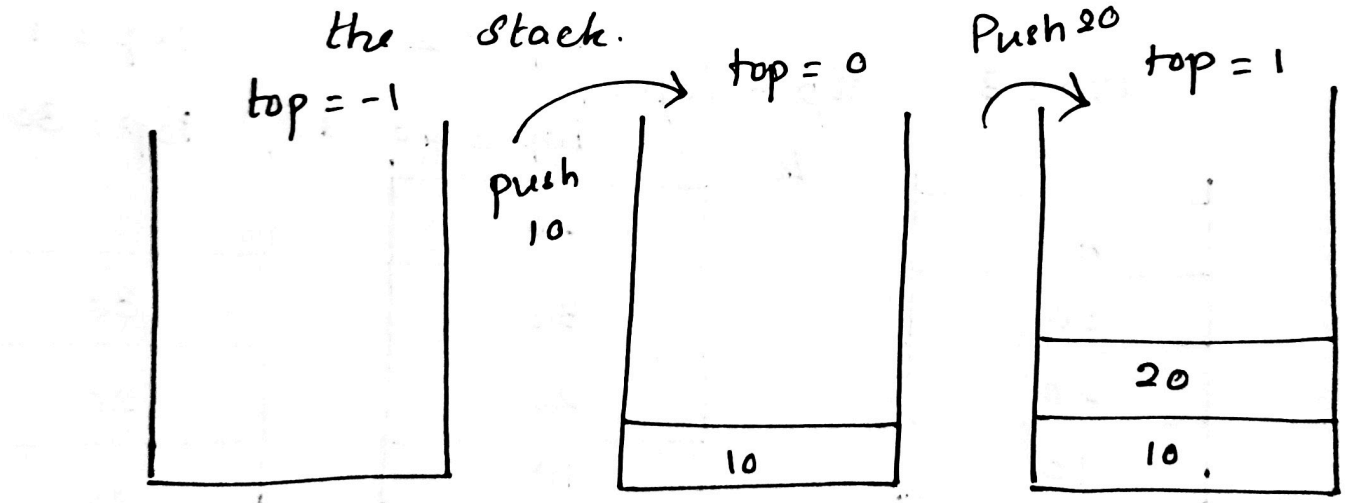
⇒ check whether the stack is full
If full → overflow occurs

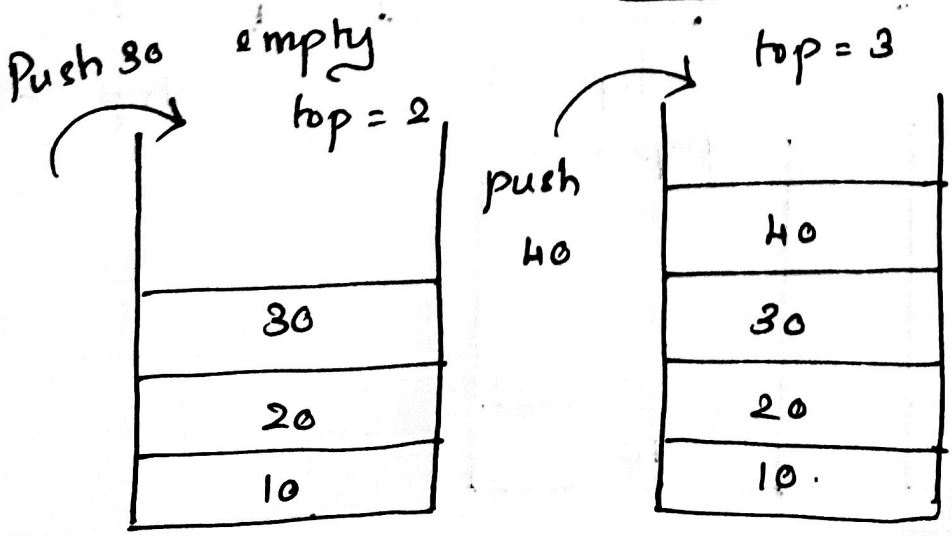
⇒ First top is -1 which checks that the stack is empty.

⇒ The value of top is incremented by 1 when a new element is pushed in a stack.

$$top = top + 1$$

⇒ The element will be inserted until we reach the max size of the stack.





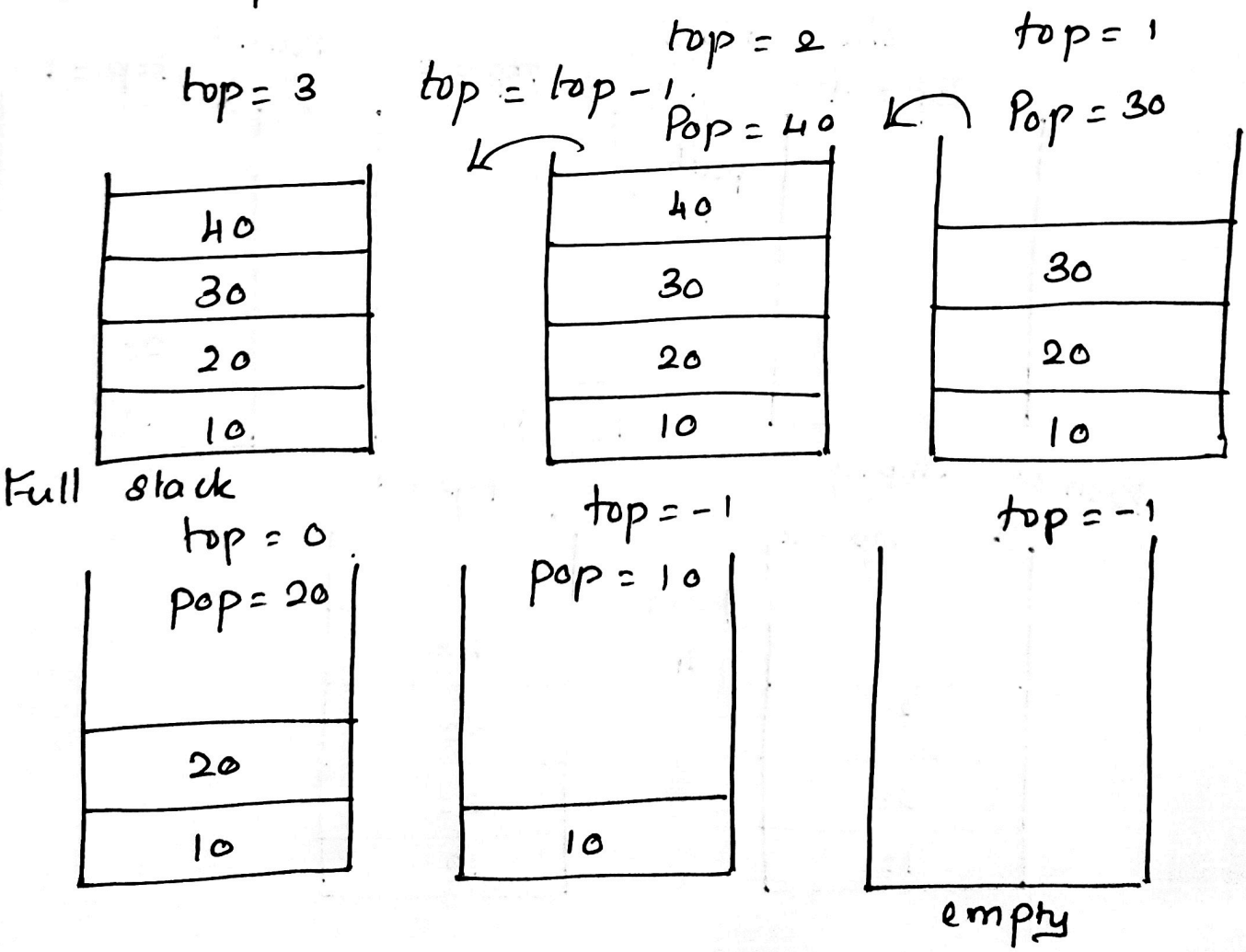
Pop operation:

⇒ Before deleting the element from the stack, check whether the stack is empty.

⇒ If the stack is empty then stack underflow condition occurs.

⇒ If the stack is not empty, we first access the element pointed by the top.

⇒ once the pop operation is performed, the top is decremented by 1.



Applications of stack:

⇒ Stack is used for balancing a symbol.

⇒ Stack is also used for reversing a string.

⇒ Stack can be used for expression conversion.

- ↳ Infix to prefix
- ↳ Infix to postfix
- ↳ Prefix to infix
- ↳ prefix to postfix
- ↳ postfix to infix.

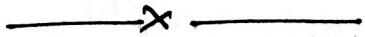
⇒ Evaluation of Arithmetic Expression

⇒ Balancing symbols

⇒ Processing function calls.

⇒ Backtracking.

⇒ Reversing a data.



Evaluating postfix expression.

Infix expression	stack	Postfix expression
(A + B / C + D * (E - F) ^ G)	(	A
(A + B / C + D * (E - F) ^ G)	(+	
(A + B / C + D * (E - F) ^ G)	(+	AB
(A + B / C + D * (E - F) ^ G)	(+ /	ABC
(A + B / C + D * (E - F) ^ G)	(+ /	ABC/+
(A + B / C + D * (E - F) ^ G)	(+	ABC/+D
(A + B / C + D * (E - F) ^ G)	(+	ABC/+D
(A + B / C + D * (E - F) ^ G)	(+*	ABC/+D
(A + B / C + D * (E - F) ^ G)	(+*	ABC/+D
(A + B / C + D * (E - F) ^ G)	(+*(	ABC/+DE
(A + B / C + D * (E - F) ^ G)	(+*(	ABC/+DE
(A + B / C + D * (E - F) ^ G)	(+*(-	ABC/+DEF
(A + B / C + D * (E - F) ^ G)	(+*	ABC/+DEF-
(A + B / C + D * (E - F) ^ G)	(+*^	ABC/+DEF-
(A + B / C + D * (E - F) ^ G)	(+*^	ABC/+DEF-G
(A + B / C + D * (E - F) ^ G)	empty.	ABC/+DEF-G^*+.

Linked list implementation of stack.

```
Struct Node
{
int data ;
Struct node * next;
};
Node * top = NULL;
```

①
```
Struct Node * newnode = (Struct Node *)
                        malloc (sizeof (struct
                                    node));
newnode -> data = 10;
```

③
```
newnode -> next = NULL;
newnode -> next = top ;
top = newnode;
```

Steps to element into a stack.

(1) Create a new node using dynamic memory allocation and assign value to the node.

(2) check if the stack is empty or not ie if (top == NULL)

(3) if it is empty then set the next pointer of the node to NULL.

(4) If it is not empty, the newly created node should be linked to the current top element of the stack (ie)

```
newnode -> next = top;
```

(5) Make sure that the top of the stack should always be pointing to the newly created node.

```
top = newnode;
```

pop operation

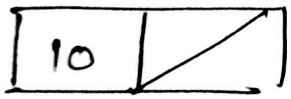
```
struct Node
{
 int data;
 struct node *next;
};
node *top = NULL;
int pop()
 {
  if (top==NULL)
   {
    printf("\n Empty stack");
   }
  else
 {
```

```
        struct node * temp = top;
        int temp→data = top→data;
        top = top→next;
        free(temp);
    }
}
```

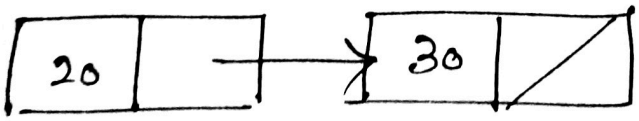
push operation

```
if (top == NULL)
```

newnode

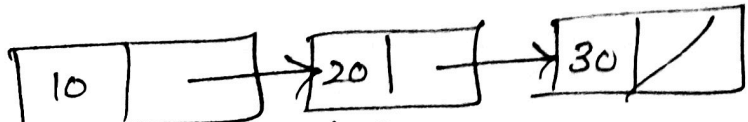



```
if (top != NULL)
```



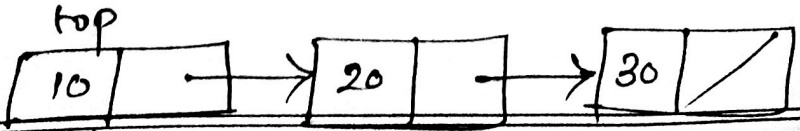
top

```
newnode → next = top;
top = newnode.
```



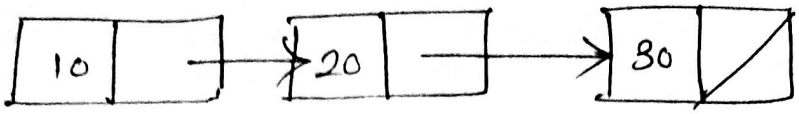
newnode

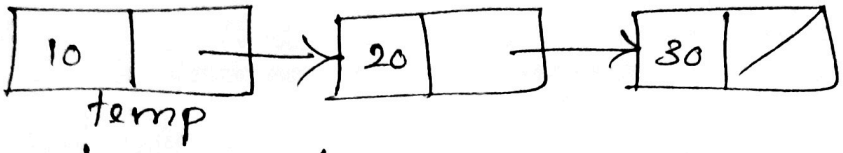
top



top

pop operation

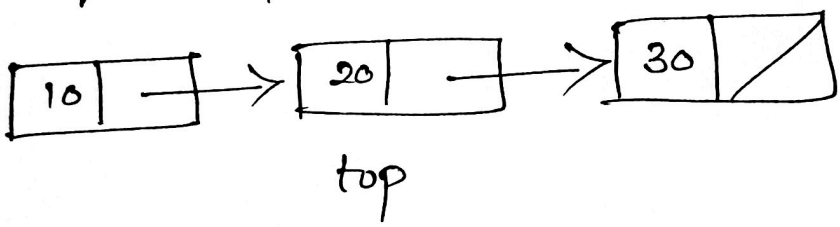




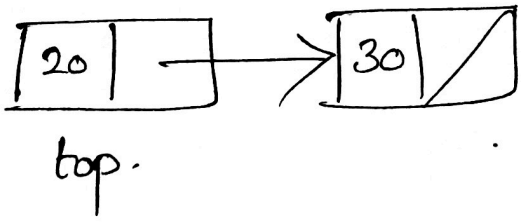
node * temp = top

temp → data = top → data

top = top → next ;



free (temp)



Linked list implementation of Queue

1) create a newnode

newnode = (struct node *) malloc (sizeof (struct node));

2) Now the two conditions arise (a) either the queue is empty or the queue contains atleast one element

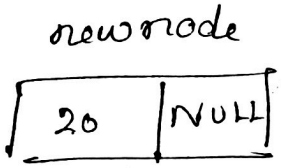
3) if the queue is empty, then the new node added will be both front and rear and the next pointer of front and rear will point to NULL.

4) if the queue contains atleast one element then the condition front == NULL becomes false. So make the next pointer of rear point to newnode and point the rear pointer to the newly created node

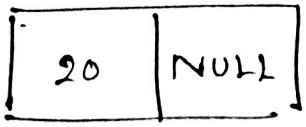
5) Hence the newnode is added to the queue.

```
struct node
{
  int data;
  struct node *next;
};
struct node *front = NULL;
struct node *rear = NULL;
```

```
void enqueue(int val)
{
struct node *newnode = malloc(sizeof(struct node));
newnode -> data = val;
newnode -> next = NULL;
if(front == NULL && rear == NULL)
    front = rear = newnode;
else
{
  rear -> next = newnode;
  rear = newnode;
}
}
```



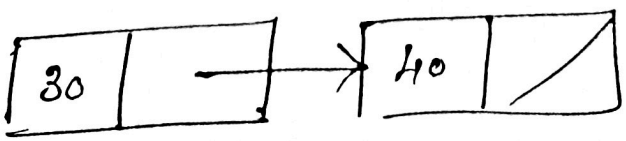
if (front == NULL && rear == NULL)



front = rear = newnode

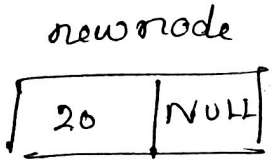
front
rear

else

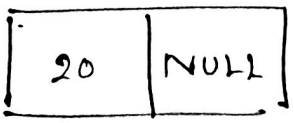


front rear

```
void enqueue(int val)
{
struct node *newnode = malloc(sizeof(struct node));
newnode -> data = val;
newnode -> next = NULL;
if(front==NULL && rear==NULL)
    front = rear = newnode;
else
{
  rear -> next = newnode;
  rear = newnode;
}
}
```



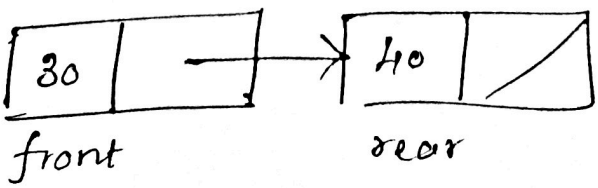

```
if(front == NULL && rear==NULL)
```



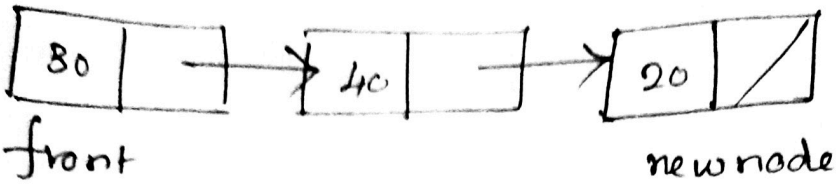
front = rear = newnode

front
rear

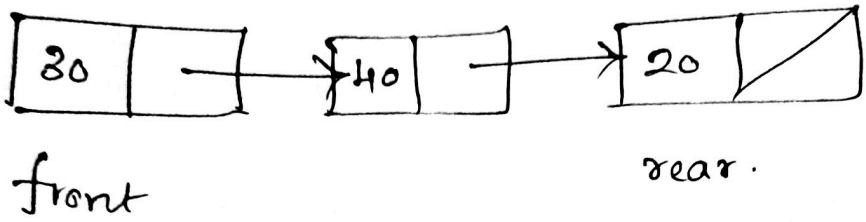
```
else
```



rear → next = newnode;



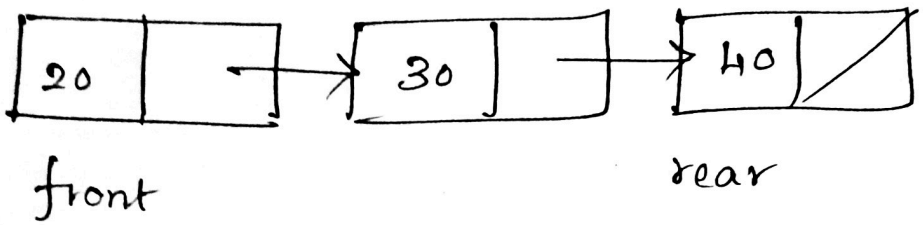
rear = newnode;



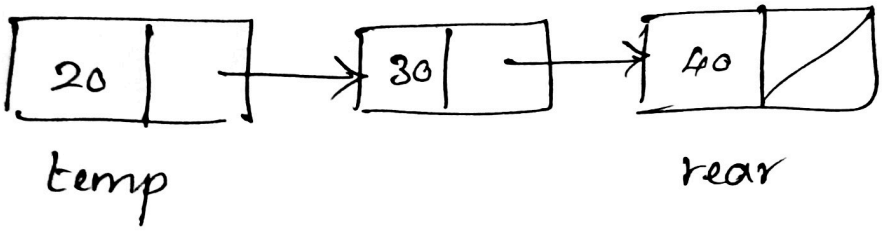
Dequeue:

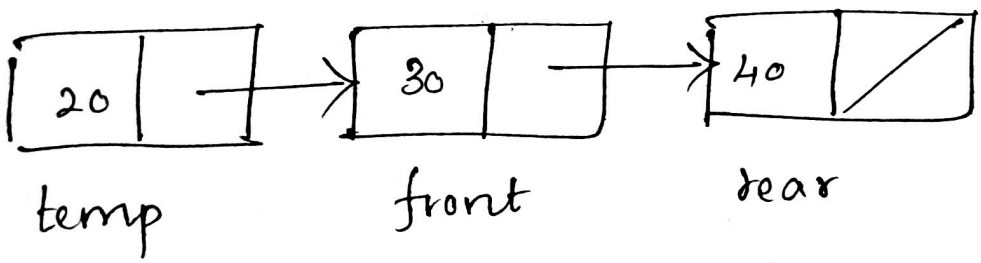
```
void dequeue()
{
struct node * temp;
if (front == NULL)
printf(" Empty Queue:");
else
{
  temp = front;
 front = front → next;
 if (front == NULL)
```

```
        rear = NULL;
        free(temp);
    }
}
```

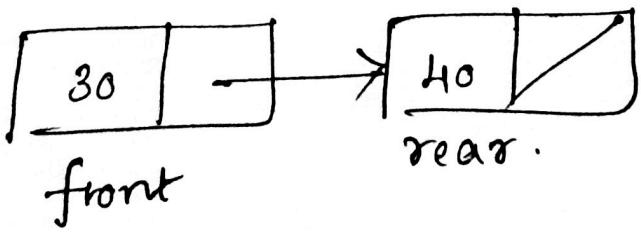


temp = front

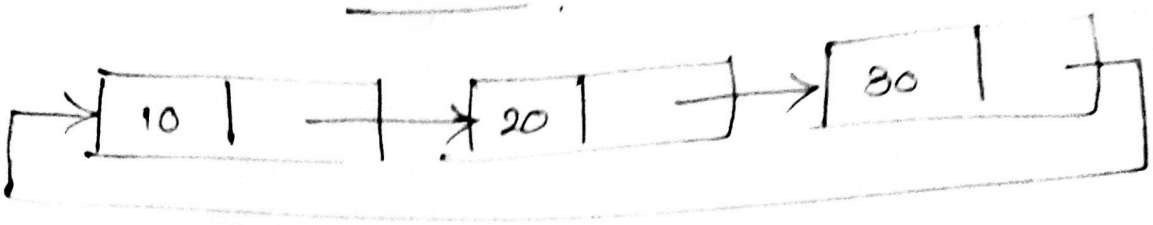




free (temp)

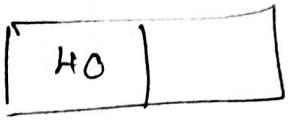


Circular Linked List

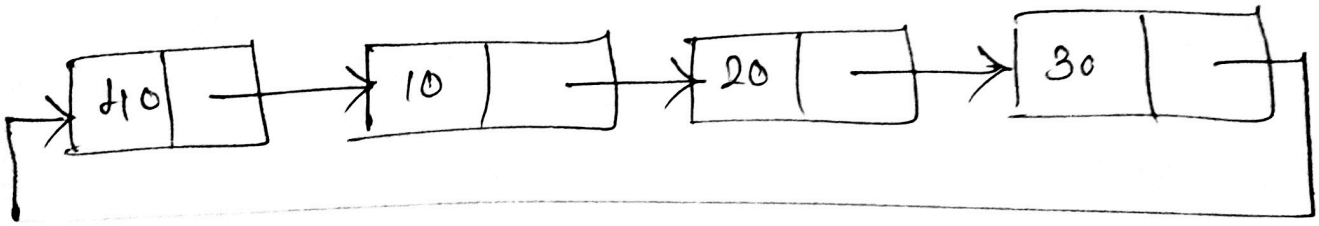


Insertion

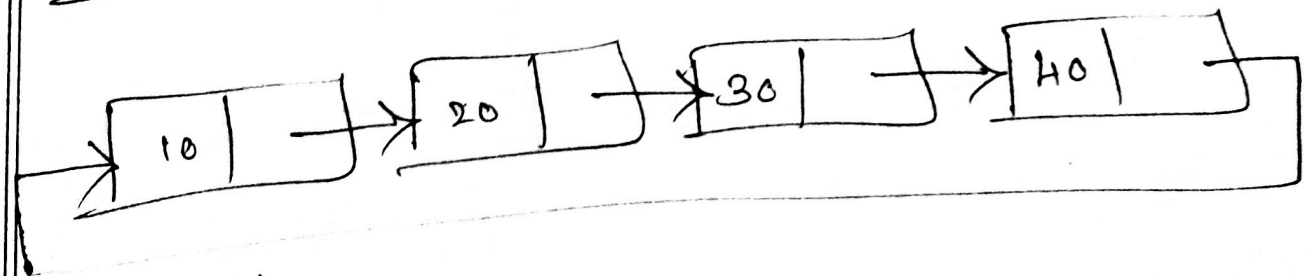
- beginning
- end
- after



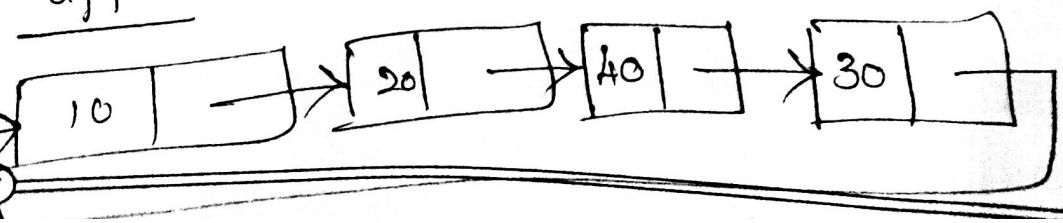
beginning



end



after



Queue ADT :

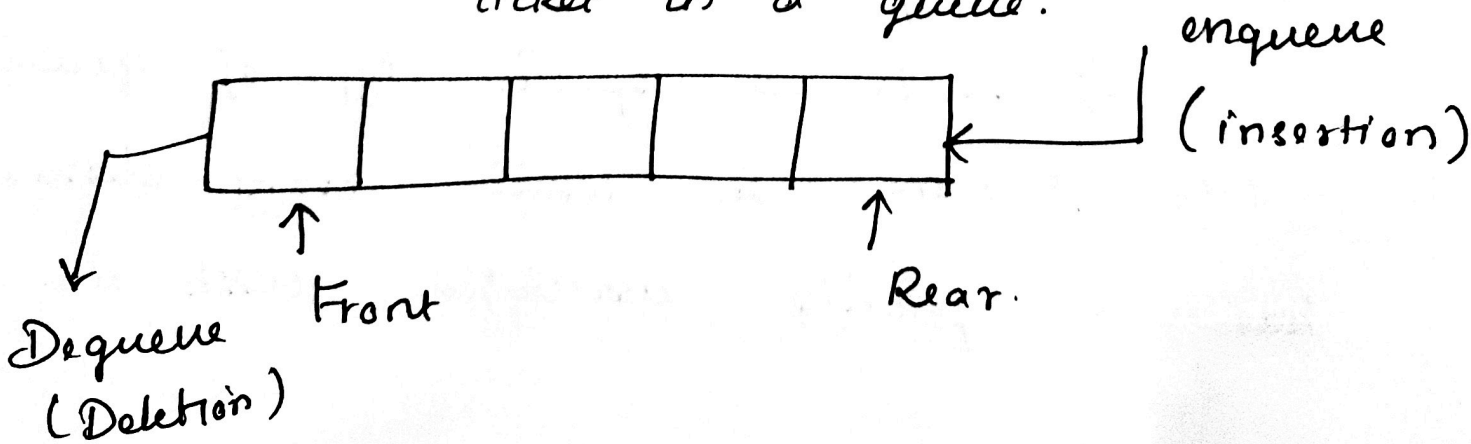
⇒ A Queue can be defined as an ordered list which enables insert operations to be performed at one end called REAR end and delete operations to be performed at another end called FRONT.

⇒ Queue is opened at both the ends. one end is always used to insert data (enqueue) and the other is used to remove data (dequeue).

⇒ Queue is referred to be as First In First Out.

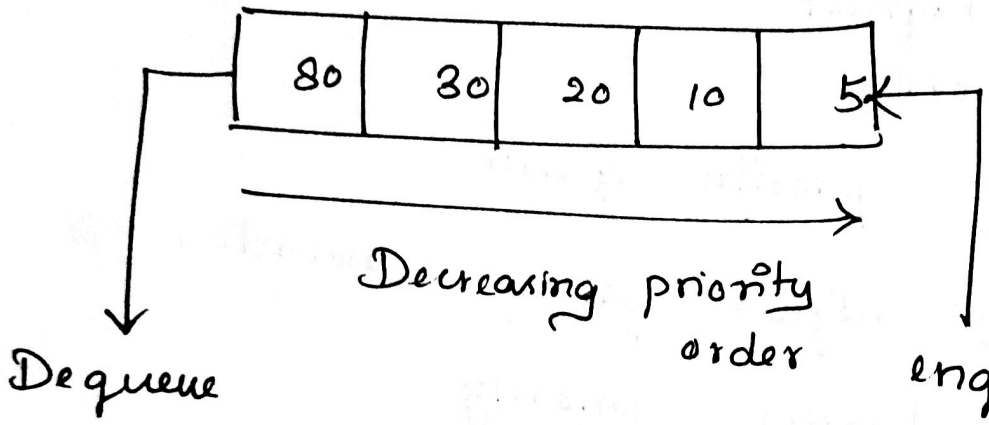
(ie) the data item stored first will be accessed first.

(eg) people waiting for train ticket in a queue.



⇒ Insertion in the priority queue takes place based on the arrival while deletion in the priority queue occurs based on the priority

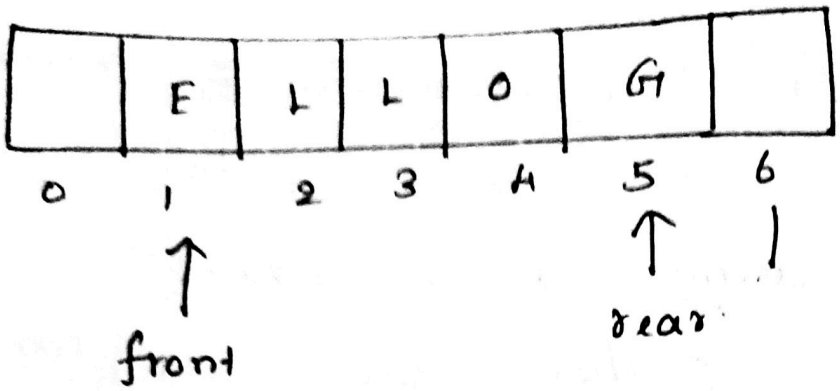
⇒ priority queue is used to implement the CPU scheduling algorithms.



Types:

1) Ascending Priority queue. → Only the smallest element is deleted first.

2) Descending priority queue → Only the largest element can be deleted first.



⇒ Queue after deleting an element from queue.

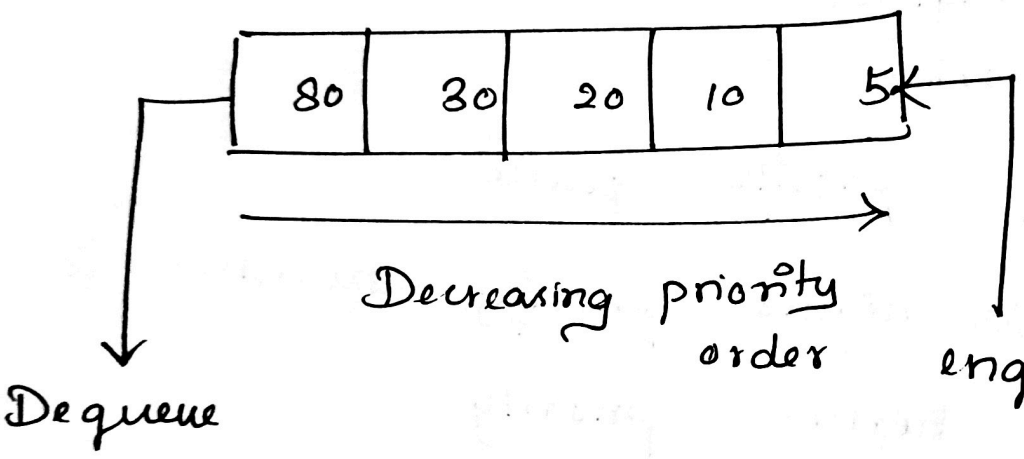
Applications of Queue:

⇒ Jobs submitted to a printer are placed on the queue.

⇒ lines in the ticket counters are queues as first come first served.

⇒ Insertion in the priority queue takes place based on the arrival while deletion in the priority queue occurs based on the priority.

⇒ priority queue is used to implement the CPU scheduling algorithms.



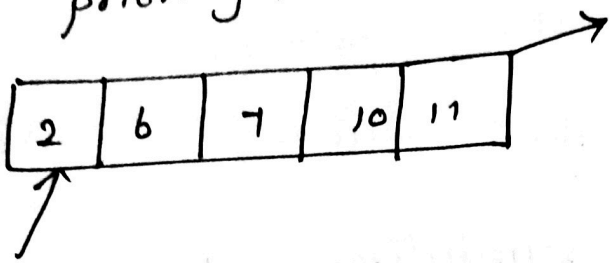
Types:

1) Ascending Priority queue. → Only the smallest element is deleted first.

2) Descending priority queue → Only the largest element can be deleted first.

Ascending priority queue:

⇒ lower priority number is given as higher priority.

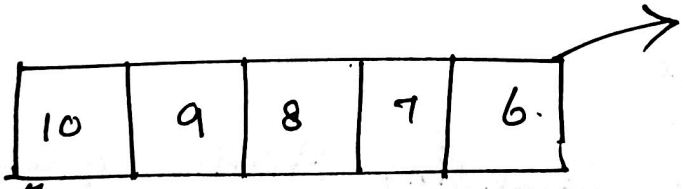


element with lowest priority.

element with the highest priority.

Descending priority queue:

⇒ higher priority number is given as higher priority.



element with the lowest priority.

element with the highest priority.

Operations:

⇒ Insertion

⇒ Deletion

⇒ peek.

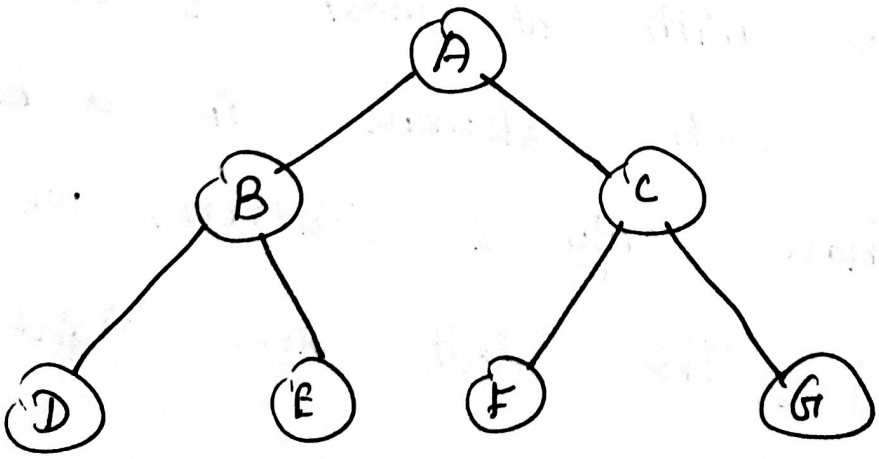
Unit - 4

Non linear Data structures.

Trees - Binary Trees - Tree Traversals - Expression Trees - Binary Search tree - Hashing - Hash Functions - Separate chaining - Open Addressing - Linear probing - Quadratic probing - Double hashing - Rehashing

Trees

⇒ A tree is a non linear data structure consisting of a collection of nodes such that each node of the tree stores a value and a list of reference to other nodes.



A → Root node

B → parent of D and E

D & E → are the siblings

D, E, F & G → leaf nodes

A & B → ancestors of F

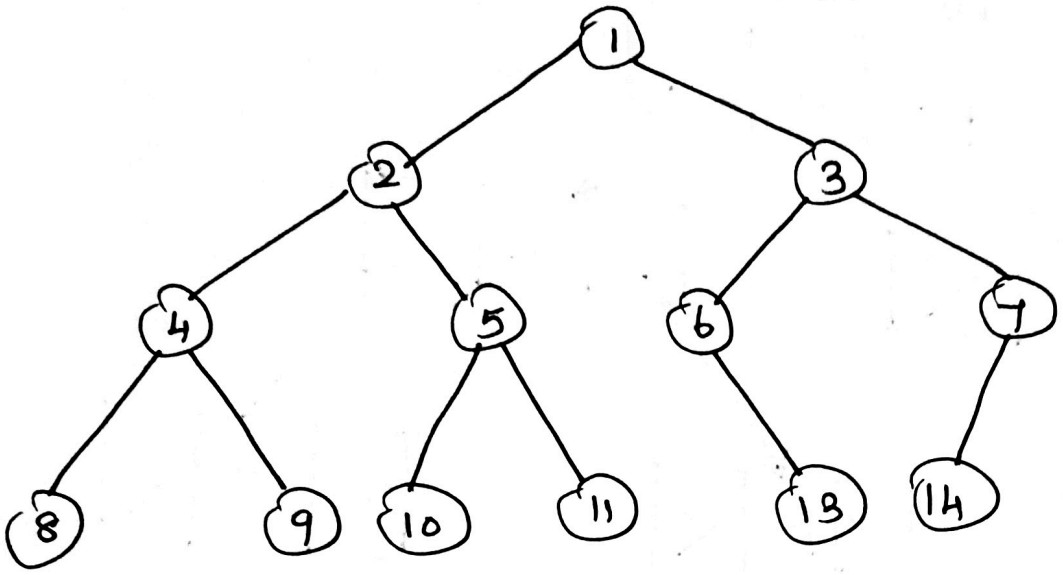
Syntax

```
struct Node
{
 int data;
 Struct Node * left-child;
 Struct Node * right-child;
};
```

Binary Trees:

⇒ Binary Tree is defined as a Tree data structure with at most 2 children.

⇒ Since each element in a binary tree can have only 2 children, we typically name them the left and right child.



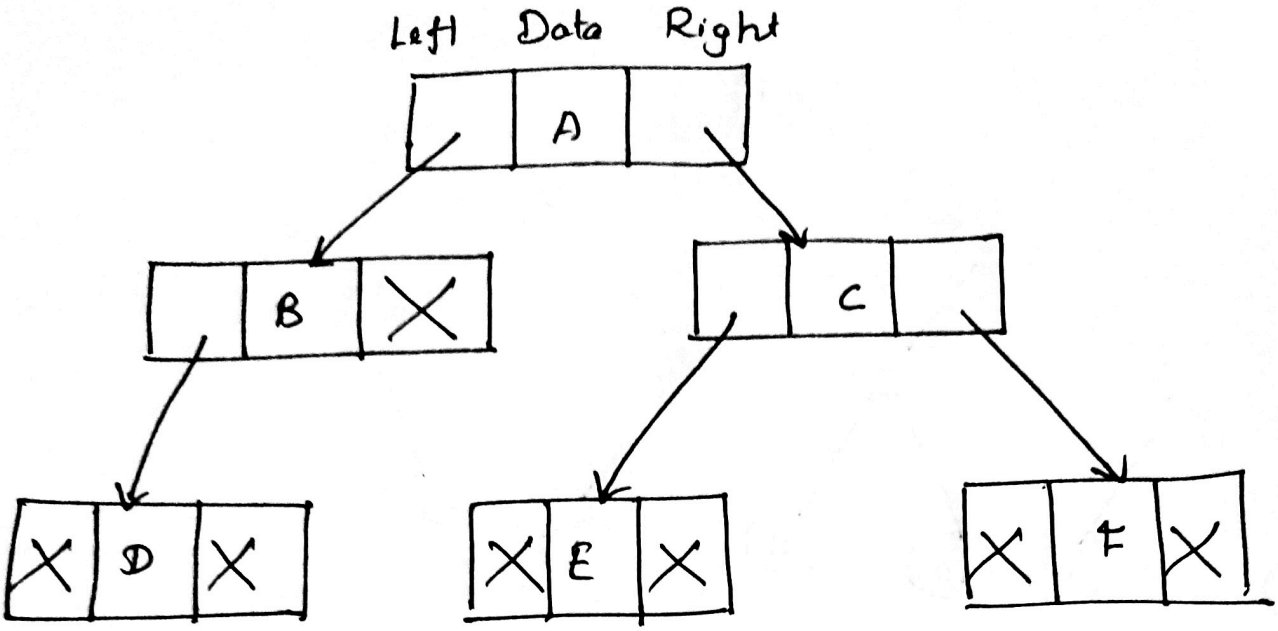
Binary Tree Representation:

⇒ A Binary tree is represented by a pointer to the topmost node of the tree.

⇒ If the tree is empty, then the value of the root is NULL.

⇒ Binary Tree node contains the following parts.

↳ Data
↳ pointer to left child.
↳ Pointer to right child.



Types of Binary Trees:

⇒ There are various types of binary tree and each of these binary trees has unique characteristic.

Full Binary Tree.

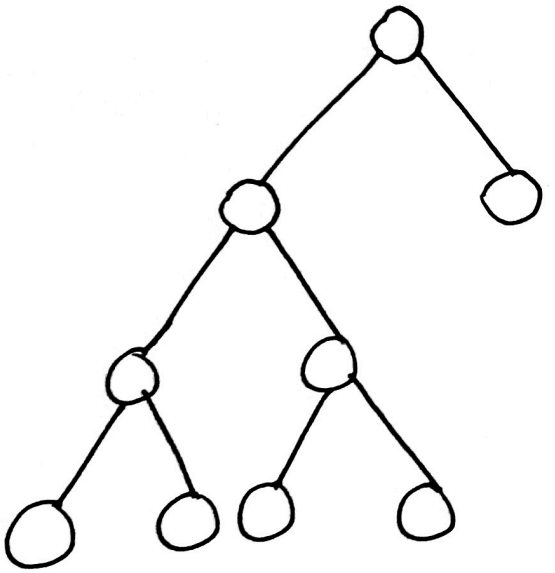
⇒ It is a special kind of binary tree that has either zero children or two children.

⇒ It means that all the nodes in that binary tree should either have two child nodes of its parent node or the parent node is itself the leaf node or the external node.

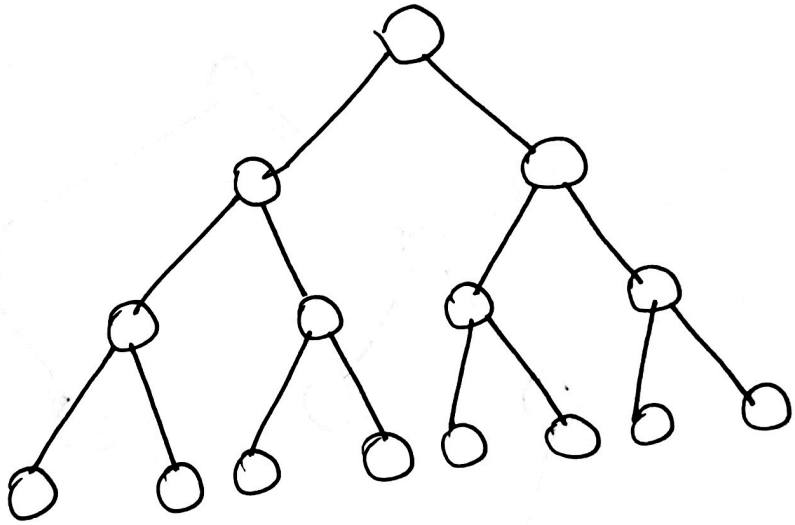
Complete binary Tree

⇒ A complete binary tree is another specific type of binary tree where all the tree levels are filled entirely with nodes, except the lowest level of the tree. Also in the last or the lowest level of this binary tree, every node should possibly reside on the left side.

Full Binary Tree.



Complete Binary Tree.



Perfect Binary Tree:

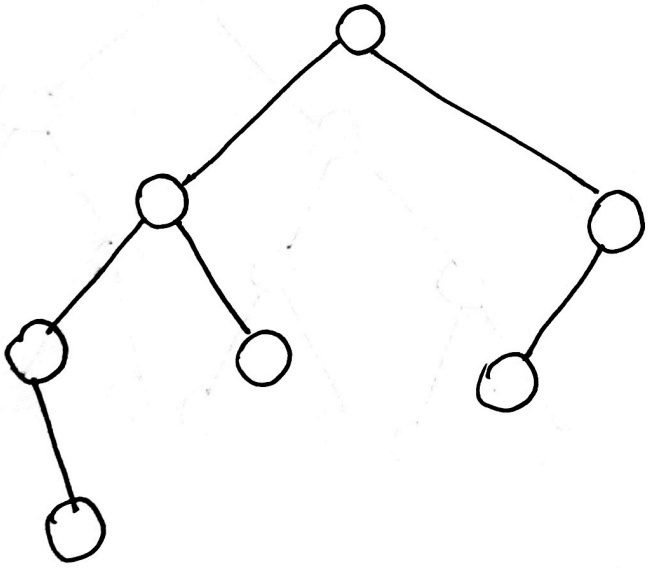
⇒ A binary tree is said to be perfect if all the internal nodes have strictly two children and every external or leaf node is at the same level.

or same depth within a tree.

⇒ A perfect binary tree having height 'h' has $2^h - 1$ node.

Balanced Binary Tree:

⇒ A balanced binary tree, also referred to as a height balanced binary tree, is defined as a binary tree in which the height of the left and right subtree of any node differs by not more than 1.



Degenerate Binary Tree:

⇒ A binary tree is said to be a degenerate binary tree or pathological binary tree if every internal node has only a single child.

Tree

⇒ hierarchical data structure.

⇒ collection of elements called nodes connected to each other through edges such that there exist one path b/w any two nodes.

⇒ last node in each path ⇒ leaf nodes / external nodes.

⇒ a node having atleast a child node ⇒ internal node.

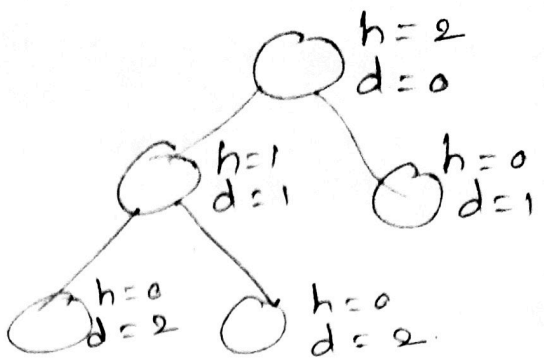
⇒ edge ⇒ link b/w any two nodes.

⇒ root ⇒ topmost node in a tree.

⇒ Height of a node ⇒ longest path from node to a leaf node.

⇒ Depth of a node ⇒ number of edges from root to the node.

⇒ Height of a tree ⇒ height of the root node or the depth of the deepest node.



Degree of a node

⇒ total number of branches of that node.

Binary Tree

⇒ each parent node can have at most two children.

Node

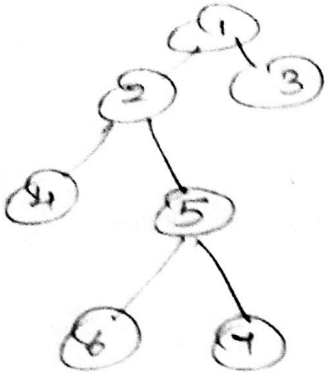
→ data item
→ address of left child
→ address of right child.

Types

→ Full BT
→ Perfect BT
→ Complete BT
→ Degenerate or pathological Tree
→ Skewed BT
→ Balanced BT.

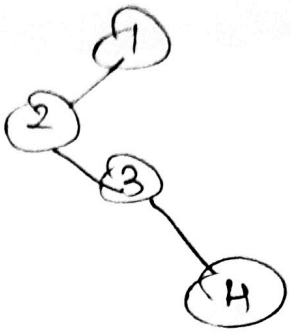
1. Full BT

Parent node → 2 or no children



Degenerate

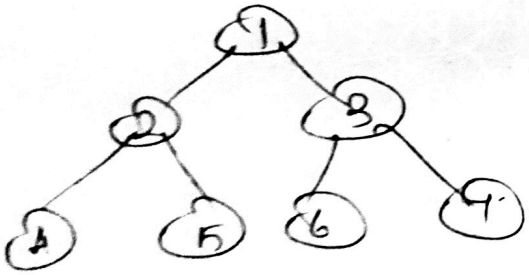
→ have single child either left or right.



② Perfect Binary Tree

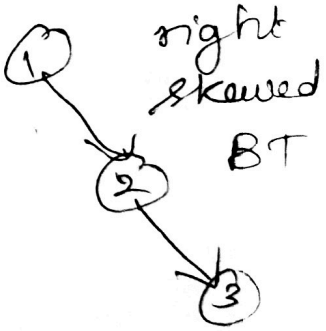
every internal node → have 2 child nodes

⇒ leaf nodes are at same level.

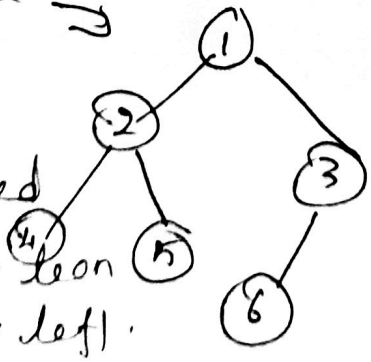


Skewed

→ tree dominated either by left or right node



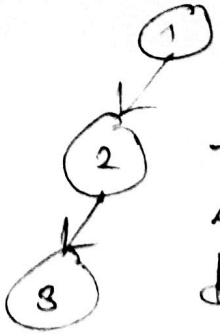
③ Complete Binary Tree ⇒



→ every level completely filled

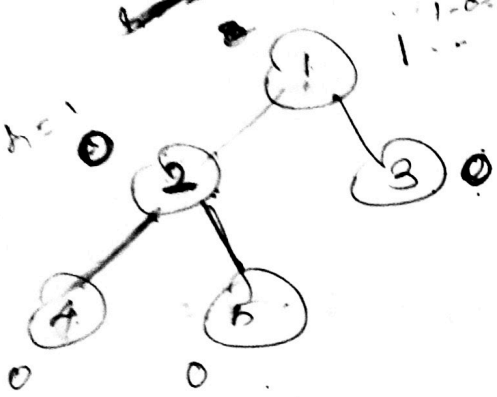
→ leaf element → lean towards left.

→ last leaf should not have right sibling.

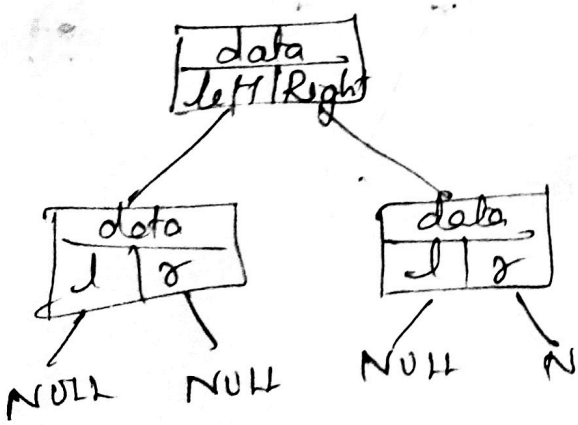


6. Balanced Binary Tree.

→ difference b/w the height of left and right sub tree for each node is 0 or 1.




```
struct node
{
  int node;
  struct node *left;
  struct node * right;
};
```



Tree Traversal

⇒ Inorder TT
⇒ Preorder TT
⇒ Postorder TT.

Traversing → visiting every nodes in the tree.

Inorder

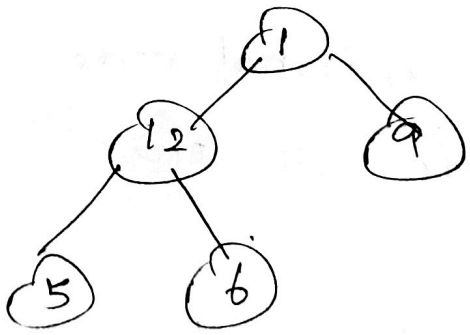
L Root Right.

Pre Order

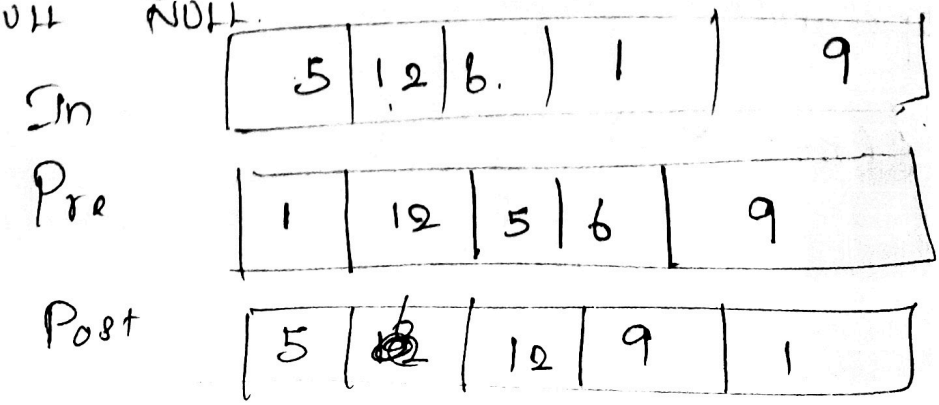
Root Left Right

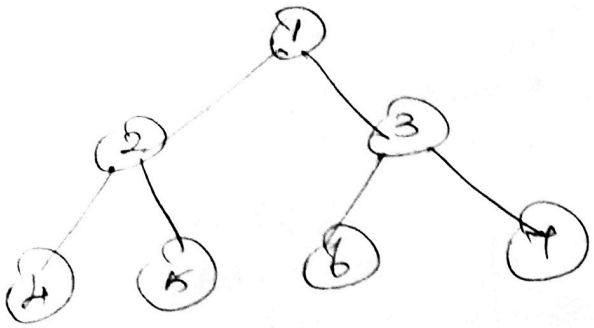
Post Order

left Right Root.



Left Root Right





In

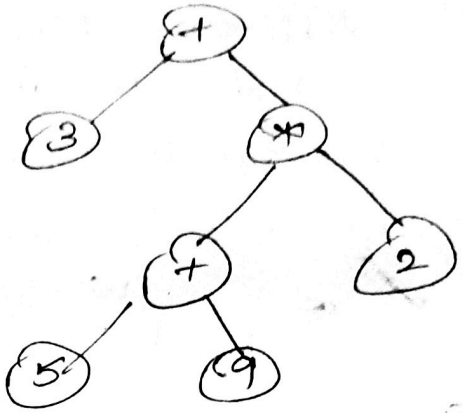
4	2	5	1	6	3	7

Pre

1	2	4	5	3	6	7

Post

4	5	2	6	7	3	1



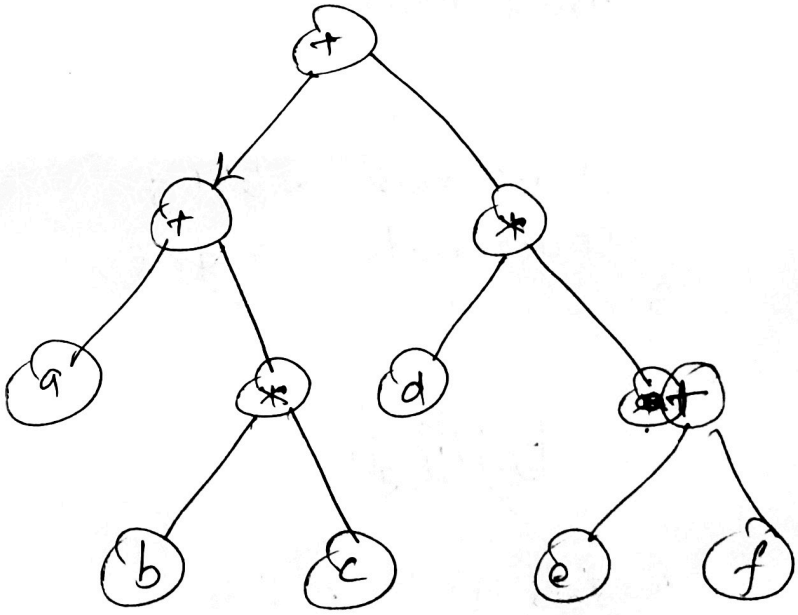
a + (b * c) + d * (e + f)

Expression Trees

⇒ each internal node corresponds to the operator.

⇒ each leaf node corresponds to the operands.

3 + ((5+9) * 2)



Prophyles

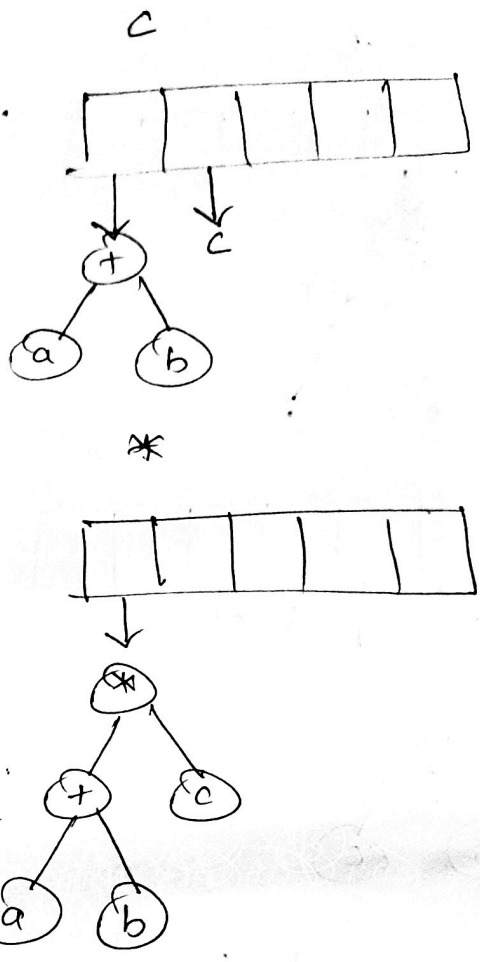
construction of expression tree

→ stack is used in building an expression tree.

→ character ⇒ operand add into stack

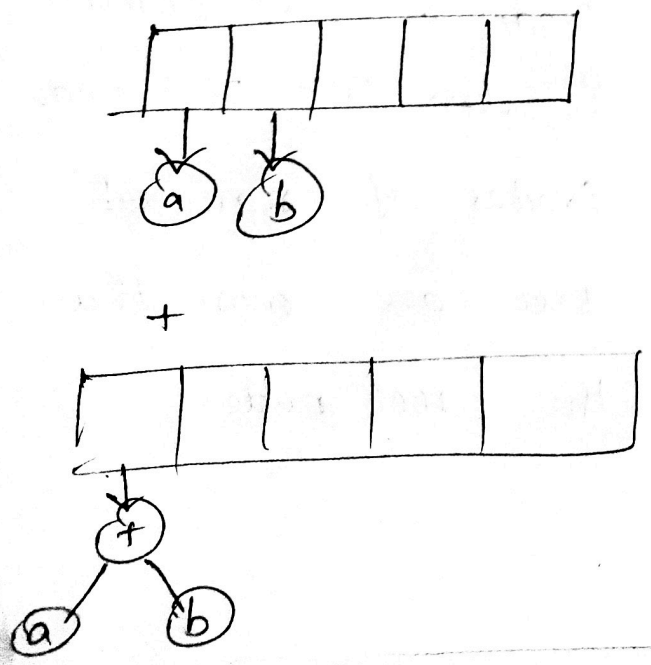
→ character ⇒ operator → pop both the values from the stack and make both its children.

→ Finally stack's lone element will be the root of expression tree.



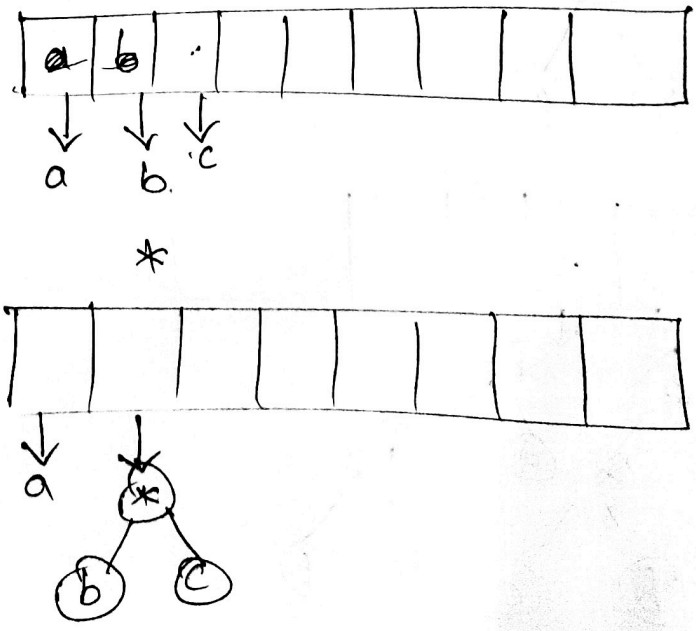
post fix expression

a b + c *

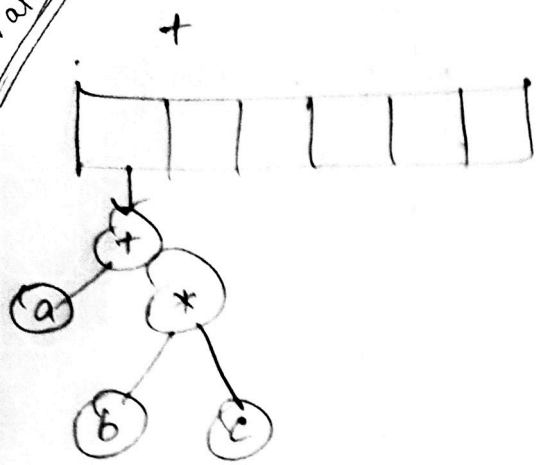


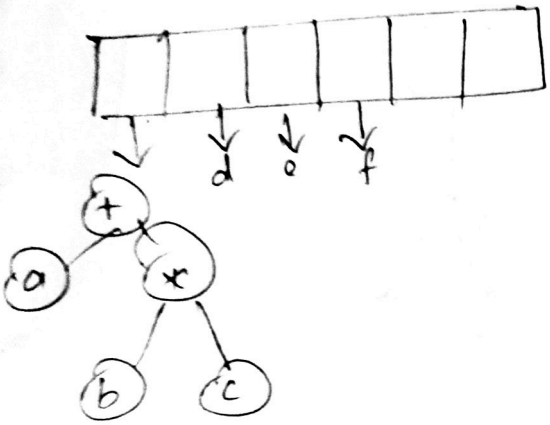
Postfix

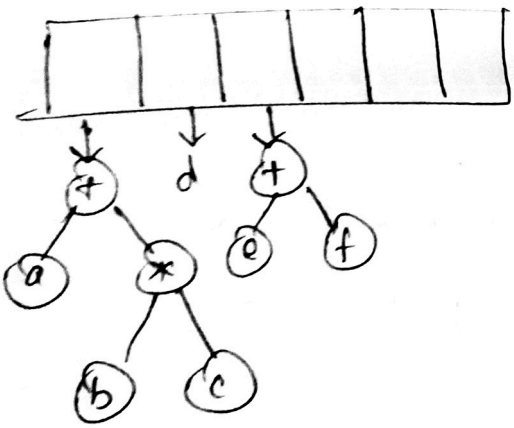
a b c * + d e f + * +

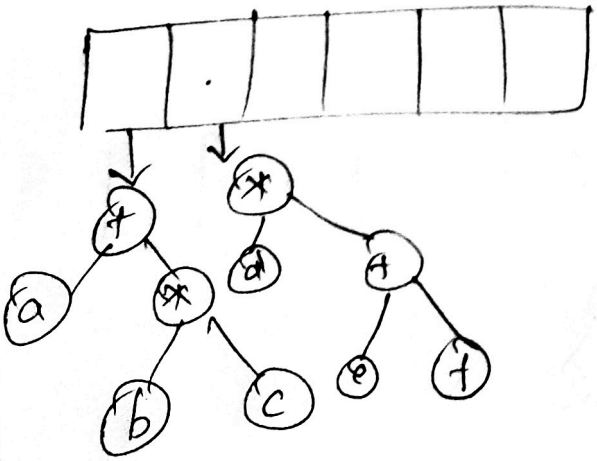


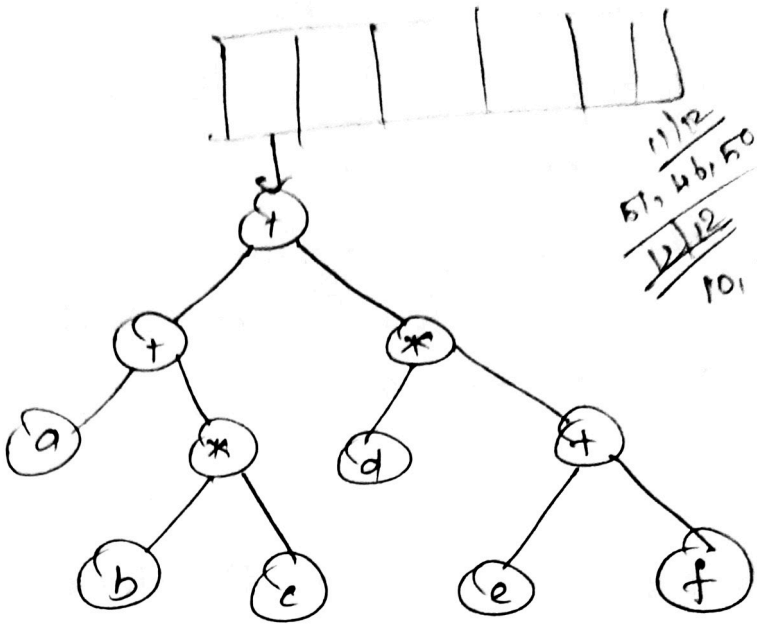
Krivalas











11/12
51, 46, 50, 55.

12/12
101, 46, 2

Binary Search Trees

⇒ Binary → mostly 2 child

⇒ Search → Search the elemen in O(log n) time.

⇒ properties

⇒ Nodes of left sub tree is less than root node

⇒ Nodes of right sub tree are more than the root node.

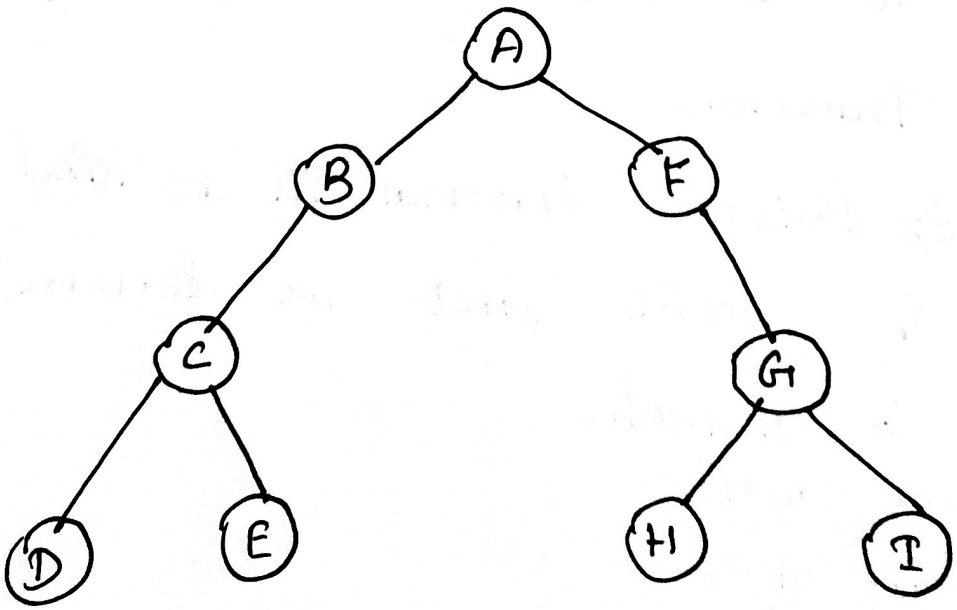
Tree Traversal:

⇒ Tree traversal means visiting each node of the tree. The tree is a non-linear data structure and therefore its traversal is different from other linear data structures.

⇒ There is only one way to visit each node / element in linear data structures i.e starting from the first value and traversing in a linear order.

Types of Tree Traversal.

⇒ Preorder traversal
⇒ Inorder traversal
⇒ Postorder traversal.

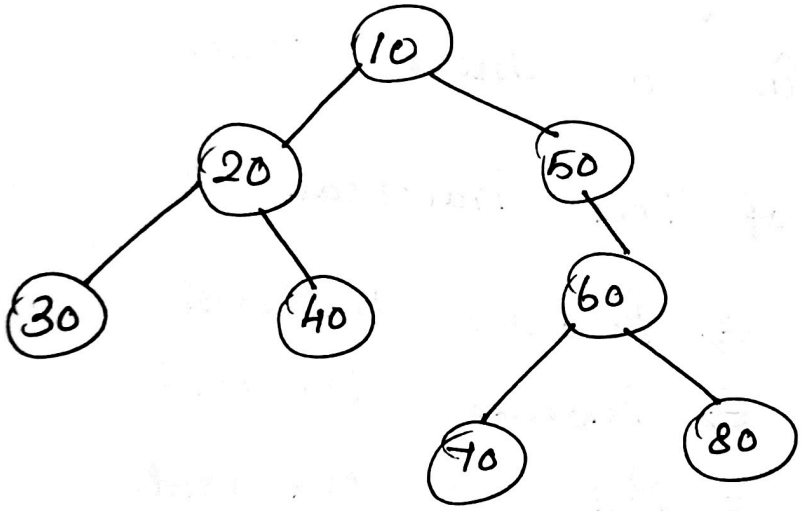


⇒ Root node first
⇒ left subtree next
⇒ right subtree.

A → B → C → D → E → F → G → H → I.

Inorder Traversal:

⇒ In an Inorder traversal, first visit the left subtree, then the root node and then the right sub tree.

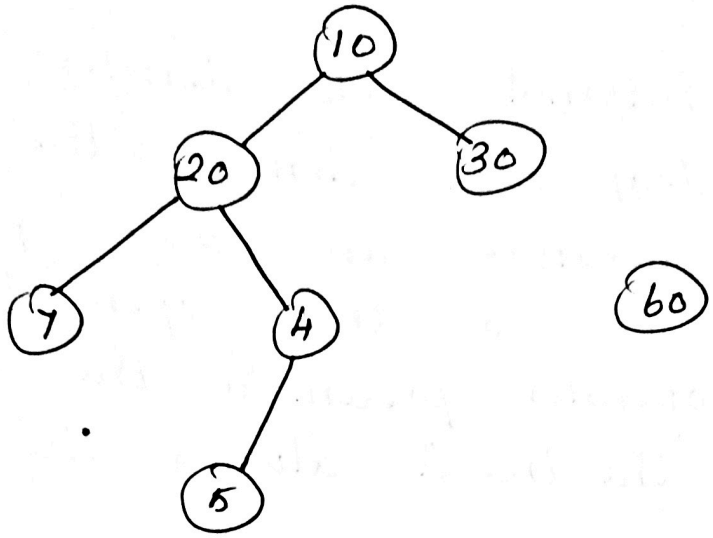


Ans: 30 → 20 → 40 → 10 → 50 → 70 → 60 → 80.

Postorder Traversal:

⇒ Postorder traversal is a kind of traversal in which first we traverse the

⇒ left subtree
⇒ right subtree
⇒ root node.
⇒



Ans: $7 \rightarrow 5 \rightarrow 4 \rightarrow 20 \rightarrow 60 \rightarrow 30 \rightarrow 10$.

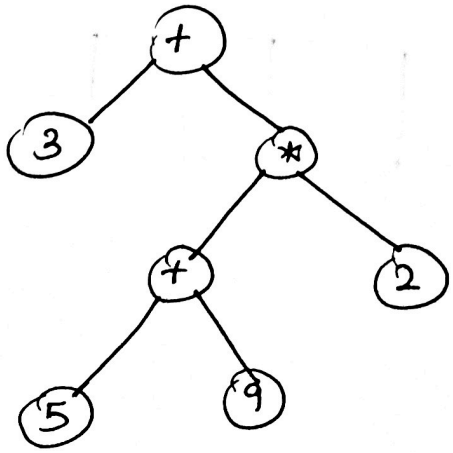
Expression Trees:

⇒ The expression tree is a tree used to represent the various expressions.

⇒ The tree data structure is used to represent the expressional statements.

⇒ In this tree, the internal node always denotes the operators. The leaf node always denote the operands.

Expression tree for $3+((5+9)*2)$

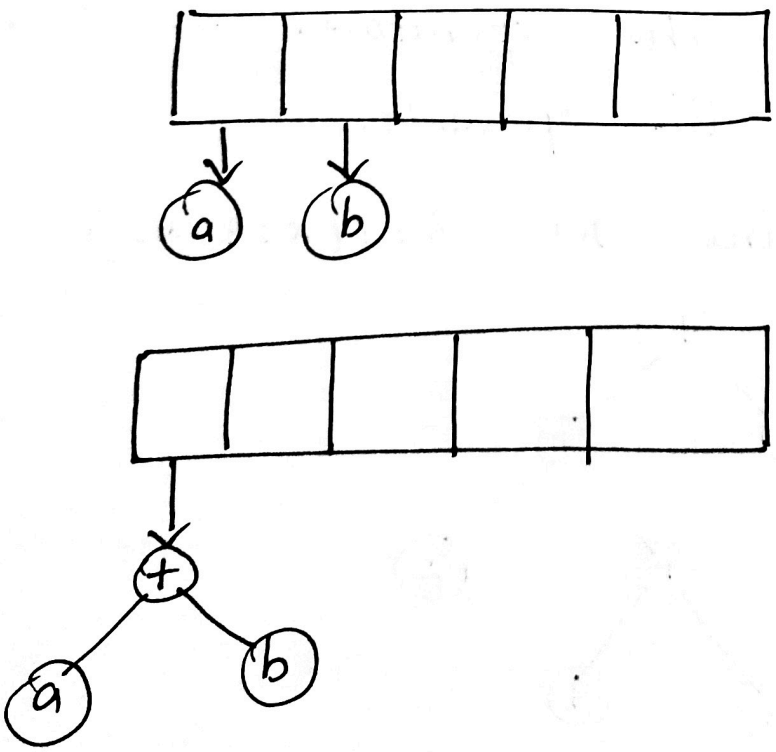


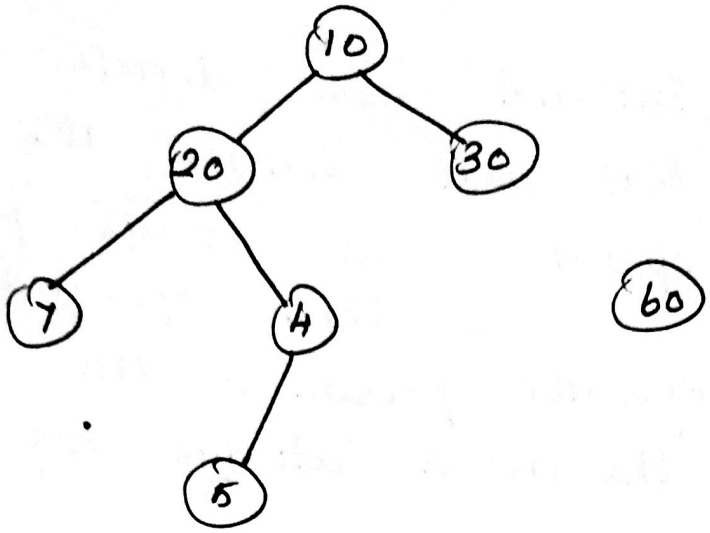
Properties:

⇒ The Internal node denotes the operators
⇒ The leaf node denotes the operands.
⇒ The operators are always performed on these operands.
⇒ The operator present in the depth of the tree is always the highest priority.
⇒ The operator which is not much at the depth in the tree is always at the lowest priority.
⇒ The operand will always present in depth of the tree.

Postfix expression construction:

ab+c *





Ans: $7 \rightarrow 5 \rightarrow 4 \rightarrow 20 \rightarrow 60 \rightarrow 30 \rightarrow 10$.

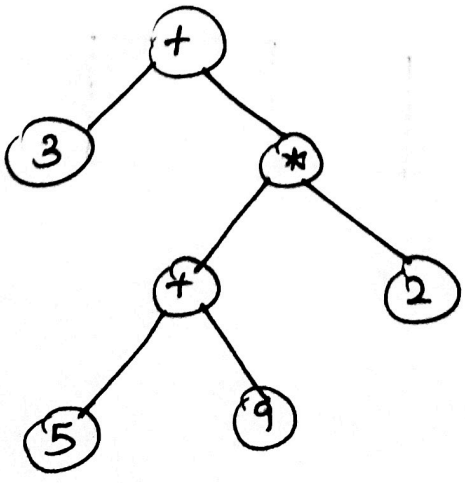
Expression Trees:

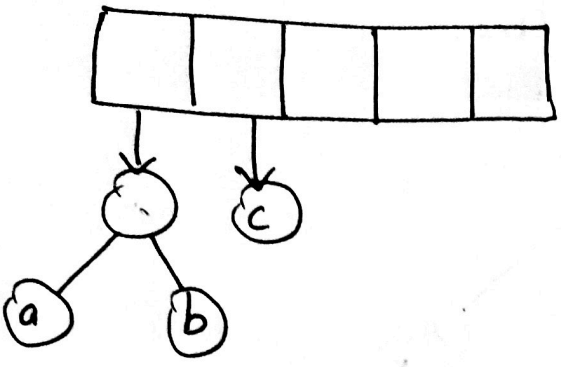
⇒ The expression tree is a tree used to represent the various expressions.

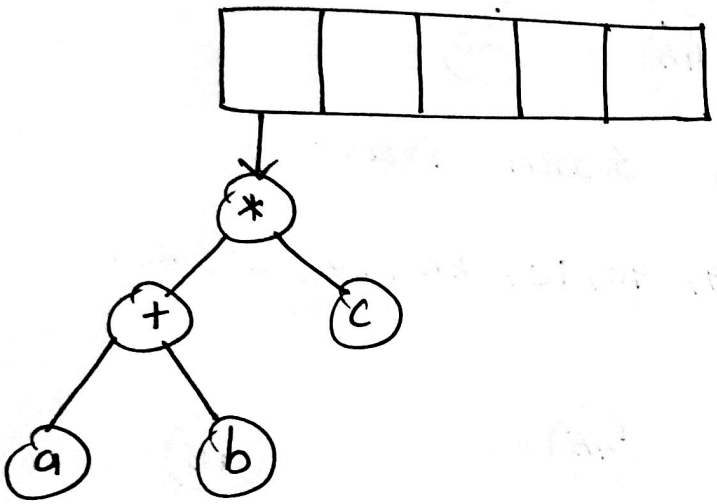
⇒ The tree data structure is used to represent the expressional statements.

⇒ In this tree, the internal node always denotes the operators. The leaf node always denote the operands.

Expression tree for $3+((5+9)*2)$





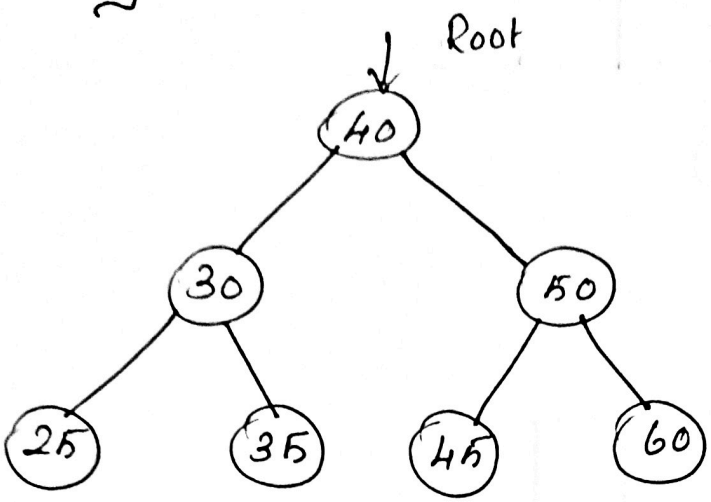


Binary Search Tree:

⇒ In Binary Search Tree, the value of the left node must be smaller than the parent node, and the value of right node must be greater than the parent node.

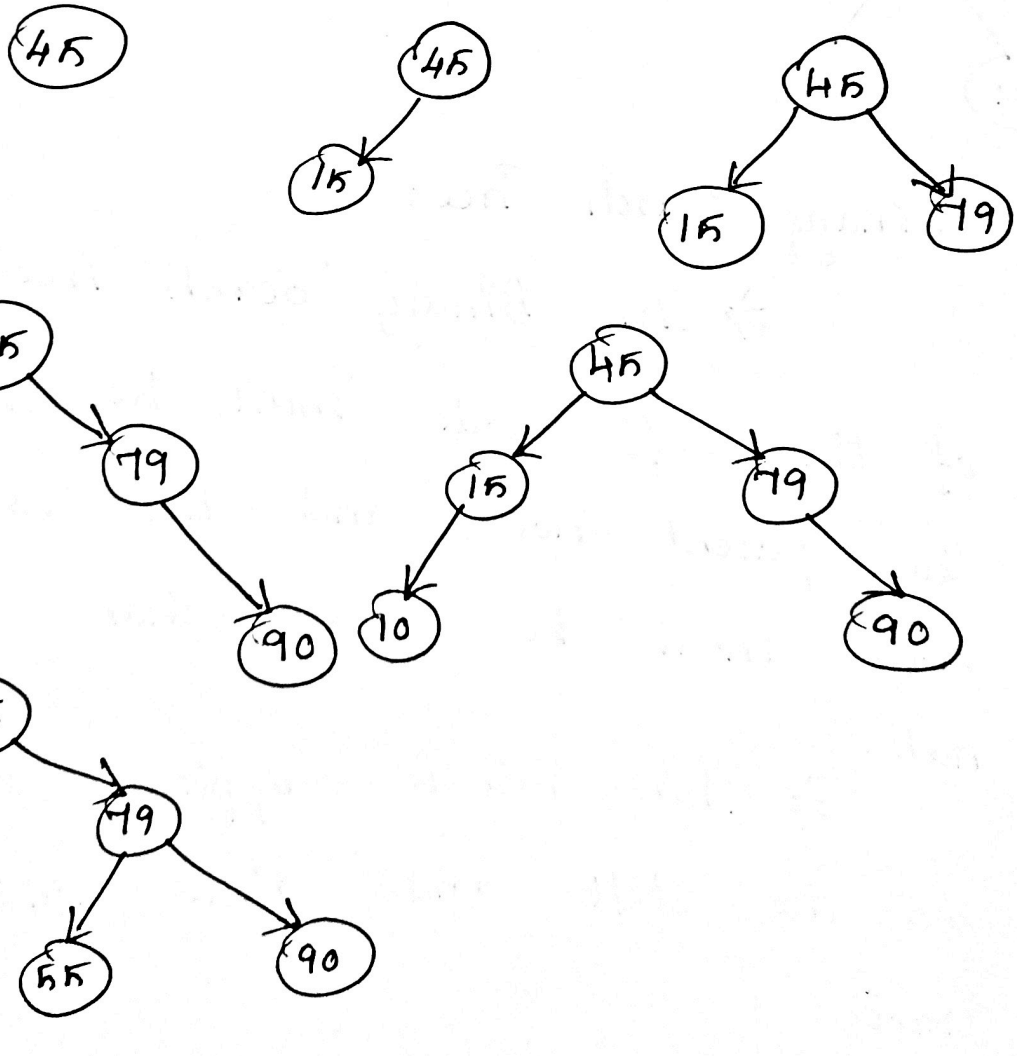
⇒ This rule is applied recursively to the left and right subtrees of the root.

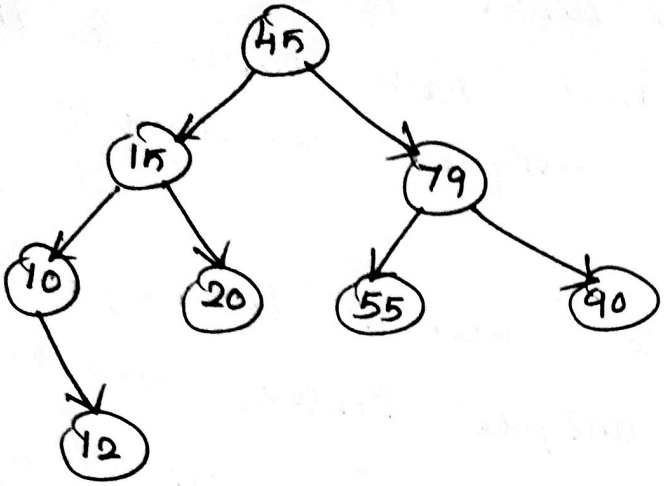
Binary Search Tree:

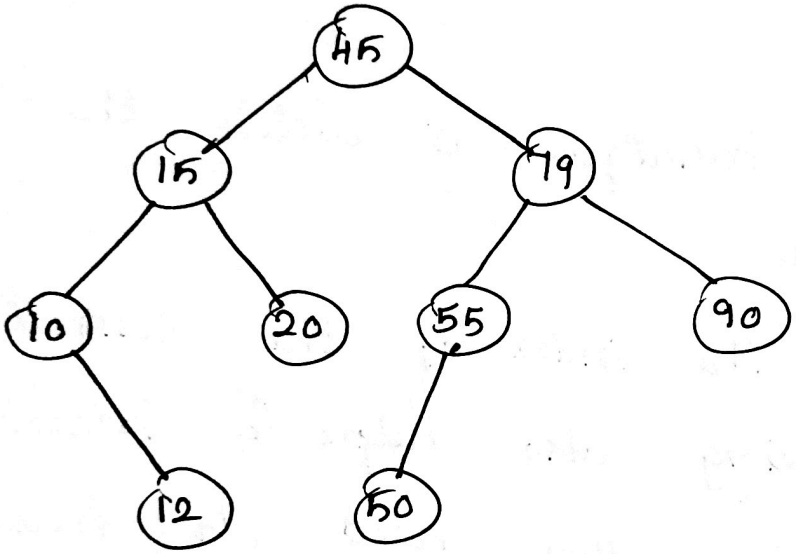


Creation of Binary Search Trees:

45, 15, 79, 90, 10, 55, 12, 20, 50.







Hashing :

⇒ Hashing is a technique of mapping a large chunk of data into small tables using a hashing function.

⇒ It is also known as the message digest function. It is a technique that uniquely identifies a specific item from a collection of similar items.

⇒ It uses hash tables to store the data in an array format. Each value in the array has been assigned a unique Index number.

⇒ Hash tables uses a technique to generate these unique Index numbers for each value stored in an array format

⇒ This technique is called the hash technique.

⇒ Find the index of the desired item.

⇒ Indexing also helps in inserting operations when you need to insert data at a specific location.

⇒ The hash table is basically the array of elements and the hash technique of search are performed on a part of the item. ie key.

⇒ Each key has been mapped to a numbers the range remains from 0 to table size 1.

⇒ Types of hashing in a data structure is a 2 step process

⇒ Hash function converts the item into a small integer or hash value. This integer is used as an index to store the original data.

⇒ It stores the data in a hash table.

⇒ Hash key is used to locate the data quickly.

Hash Function:

⇒ hash = hashfunc (key)

⇒ Index = hash % array-size.

⇒ A good hash function is easy to compute.

⇒ A good hash function never gets stuck in clustering and distributes keys evenly across the hash table.

⇒ A good hash function avoids collision when two elements or items get assigned to the same hash value.

⇒ collision free

⇒ property to be hidden.

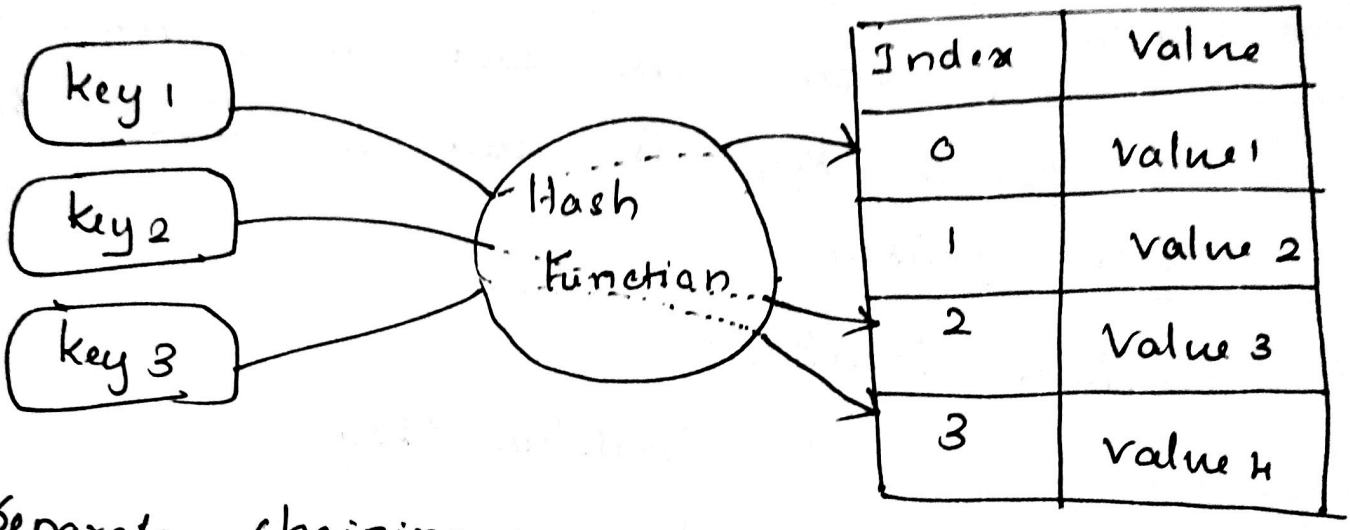
⇒ Puzzle friendly.

⇒

Hash table:

⇒ Hashing in datastructures uses hash table to store the key value pairs.

⇒ Hash table then uses hash function to generate an index.



Separate chaining:

⇒ Implemented using Linked List

When two or more elements are hash to the same location, these elements are represented into a singly linked list like a chain.

⇒ Since this method uses extra memory to resolve the collision, therefore it is also known as open hashing.

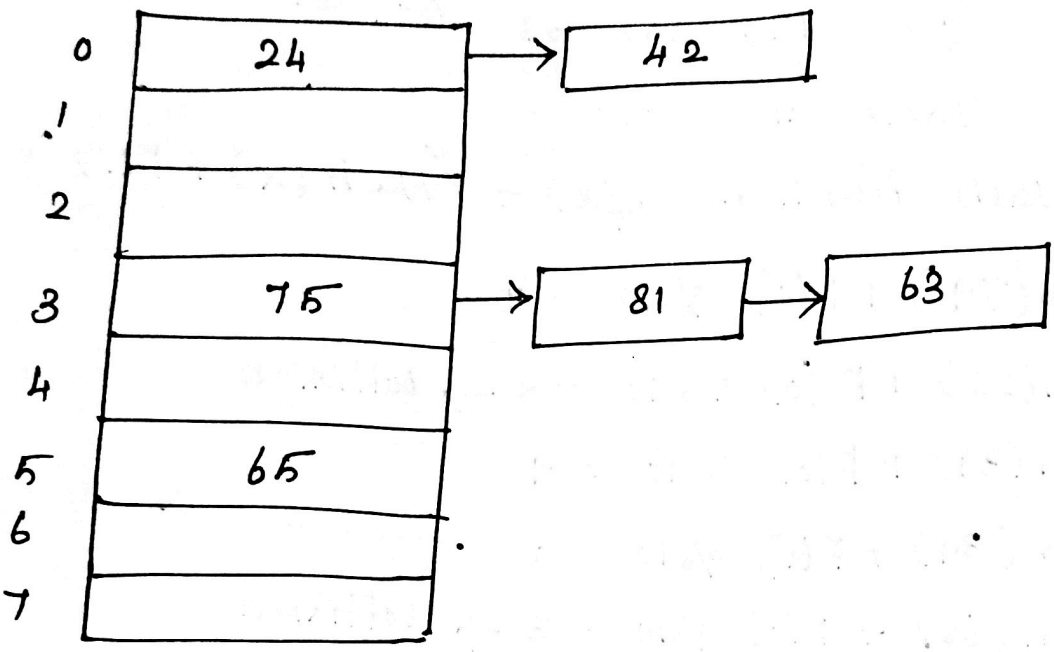
⇒ More than one element mapped to the same key, elements stored in the same linked list.

Hash Function = key % 6

Elements = 24, 75, 65, 81, 42 and 63.

24 % 6 = 0 75 % 6 = 3 65 % 6 = 5

81 % 6 = 3 42 % 6 = 0 63 % 6 = 3



Open Addressing:

⇒ another technique for collision resolution

⇒ It inserts data not into some other data structures.

⇒ It inserts the data into the hash table itself.

⇒ The size of the hash table should be larger than the number of keys.

⇒ Three open addressing techniques.

↳ Linear probing

↳ Quadratic probing

↳ Double hashing.

Linear probing:

⇒ Simple method, sequentially tries the new location until an empty location is found in the table.

(eg) 79, 28, 39, 68, 89.

Table size = 10.

Hash Function $h_i(x) = (Hash(x) + F(i)) \% \text{Table size}$.

Hash(79) + F(0) %10 = 9

Hash(28) + F(0) %10 = 8 → collision

Hash(39) + F(0) %10 = 9

Hash(39) + F(1) %10 = 0 ⟶ 0

Hash(68) + F(0) %10 = 8 → collision

Hash(68) + F(1) %10 = 9 → collision

Hash(68) + F(2) %10 = 0 → collision

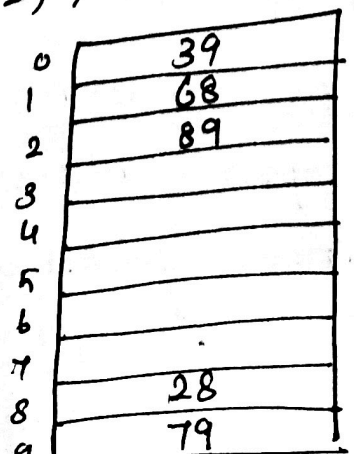
Hash(68) + F(3) %10 = 1 ⟶ 1

Hash(89) + F(0) %10 = 9 → collision

Hash(89) + F(1) %10 = 0 → collision

Hash(89) + F(1) %10 = 1 → collision.

Hash(89) + F(2) %10 = 2 ⟶ 2

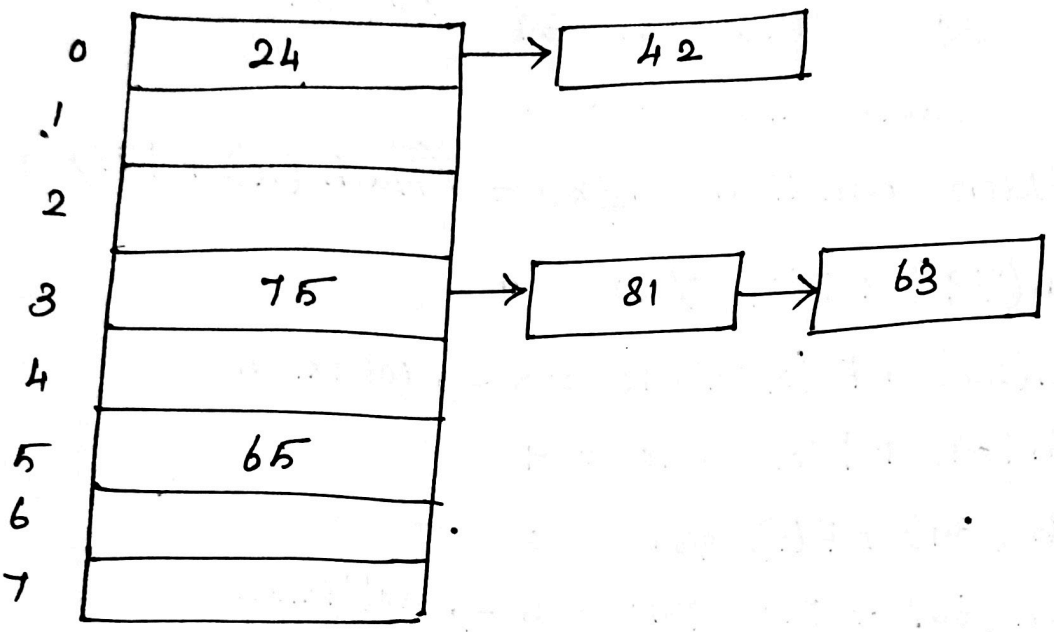


Hash function = key % 6

Elements = 24, 75, 65, 81, 42 and 63.

24 % 6 = 0 75 % 6 = 3 65 % 6 = 5

81 % 6 = 3 42 % 6 = 0 63 % 6 = 3



Open Addressing:

⇒ another technique for collision resolution

⇒ It inserts data not into some other data structures.

⇒ It inserts the data into the hash table itself.

⇒ The size of the hash table should be larger than the number of keys.

⇒ Three open addressing techniques.

↳ Linear probing

↳ Quadratic probing

↳ Double hashing.

Quadratic probing

⇒ open addressing method.

⇒ resolve collision in the hash table.

⇒ $H+1^2$, $H+2^2$, $H+3^2$...

$h_i(x) = (Hash(x) + F(i)^2$ % Tablesize.

$h_0(79) =$
$(Hash(79) + F(0)^2) \% 10$
$= 9$
$Hash(28) + F(1)^2 \% 10$
$= 8.$

0	39
1	
2	68
3	89
4	
5	
6	
7	
8	28
9	79

Double hashing.

⇒ uses 2 hash functions.

⇒ Therefore called double hashing.

⇒ 1st hash fn → initial location to locate the key.

2nd hash fn → size of Jumps in the probe sequence.

0	68
1	
2	39
3	89
4	
5	
6	
7	
8	28
9	79.

Rehashing:

⇒ process of recalculating the hashcode of already stored entries (key-value) pairs. to move them to another bigger size hashmap when the threshold is reached / crossed.

⇒ For each addition of a new enty to the map, check the current load factor.

⇒ If's greater than it predefined value then rehash.

⇒ Rehash make a new enty of double the previous size and make it the new bucket array.

element 1 → 0.2 → [0]

element 2 → 0.4 → [1]

element 3 → 0.6 → [2]

element 4 → 0.8 → [3]

element 2 → LF = 0.25 → [0]

element 1 → LF = 0.1 → [2]

element 3 → LF = 0.37 → [5]

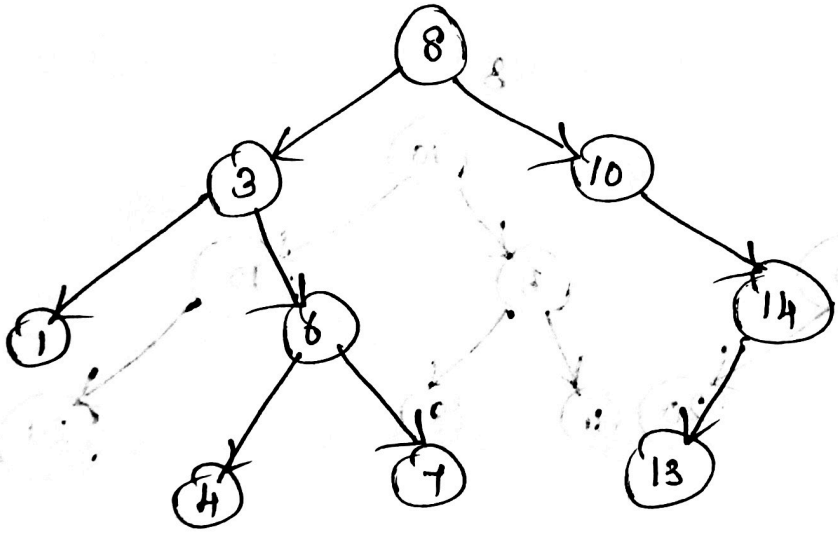
element 4 → LF = 0.5 → [7]

Binary Search Tree

⇒ Key values in the left sub tree is less than the key value of the root.

⇒ Key values in the right sub tree is greater than the key value of the root

⇒ When this relationship holds in the entire node in the tree, then the tree is called as binary search tree.



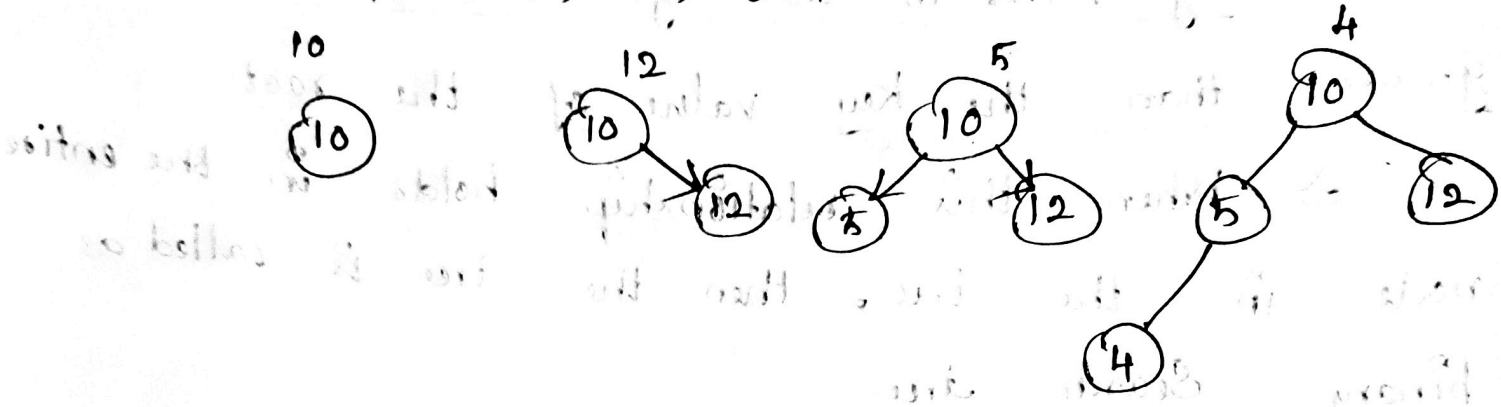
Operations on BST

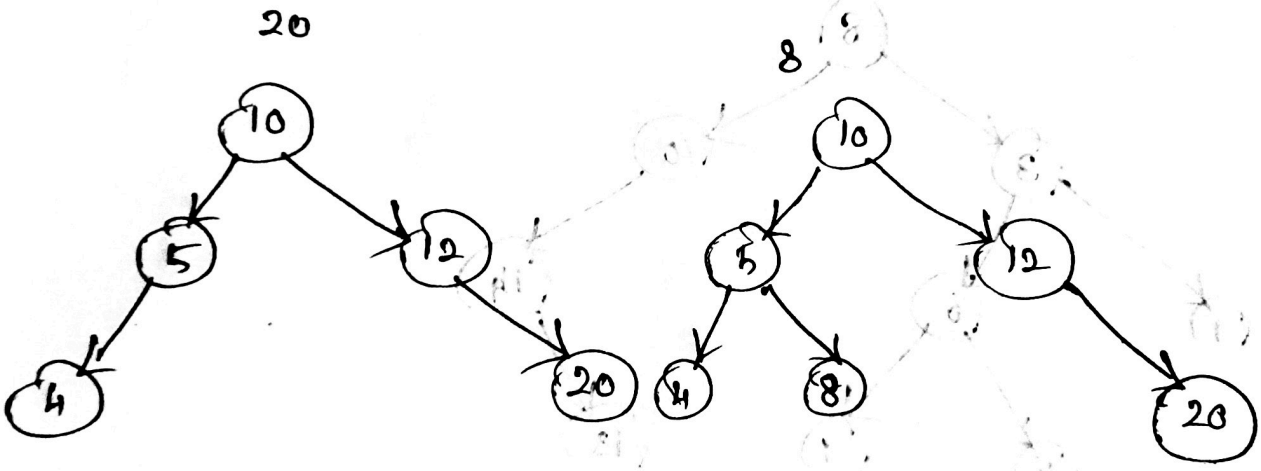
1. Insertion
2. Deletion
3. Find
4. Find min
5. Find max
6. Retrieve

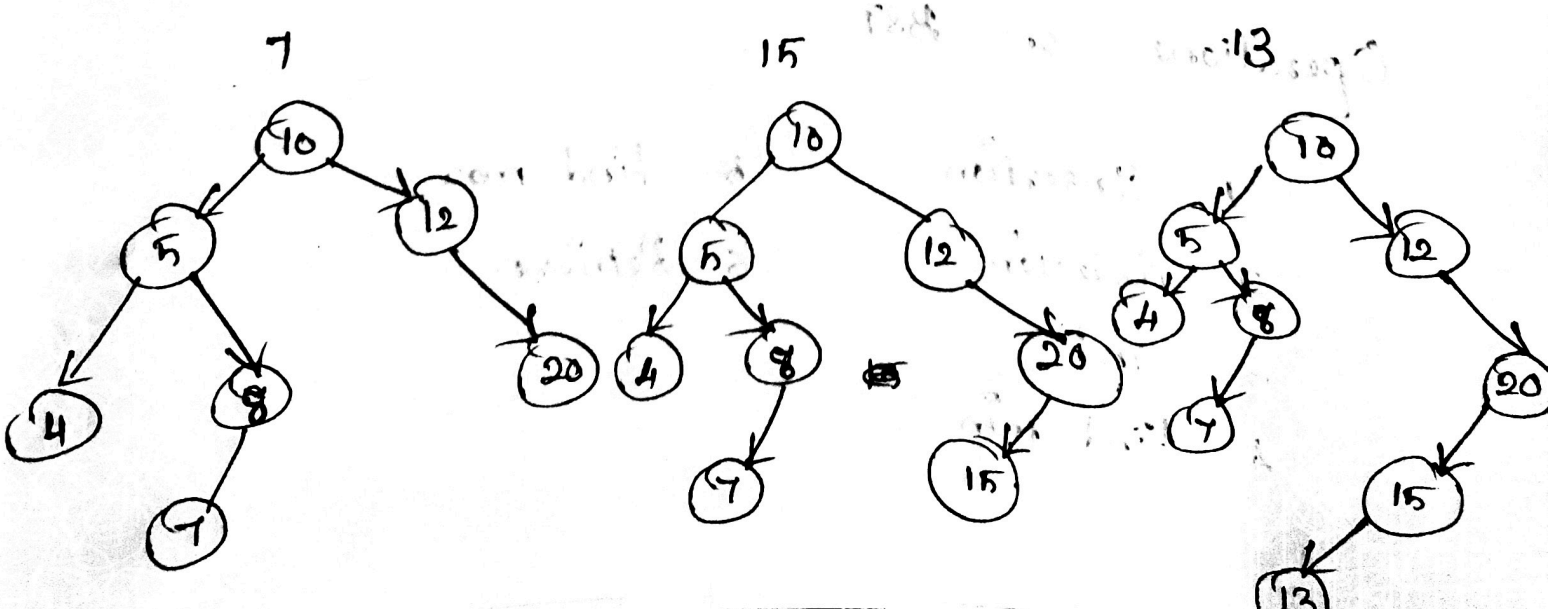
Binary Search Tree Construction

Insertion

10, 12, 5, 4, 20, 8, 7, 15, 18.







Search operation

```
if root == NULL
    return NULL;
if number == root → data
  return root → data;
if number < root → data
  return search(root → left)
if number > root → data
  return search(root → right)
```

Insertion

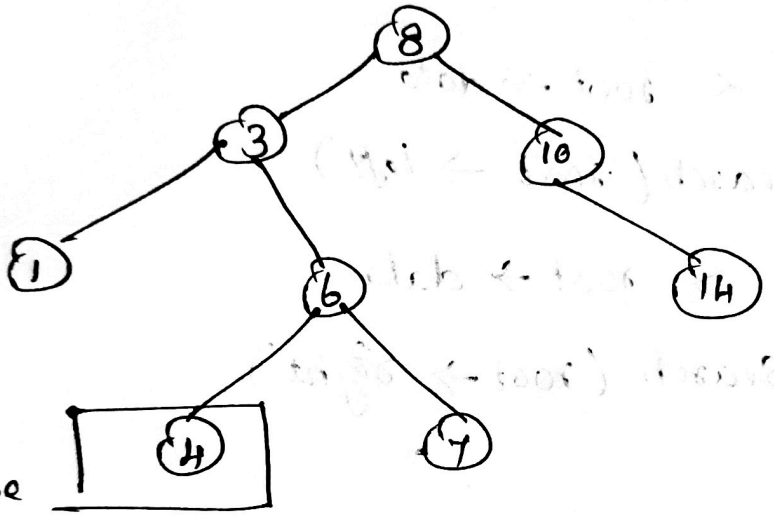
```
if node == NULL
  return create node(data)
if (data < node → data)
node → left = insert(node → left, data);
else if (data > node → data)
  node → right = insert(node → right, data);
return node;
```

Deletion Operation

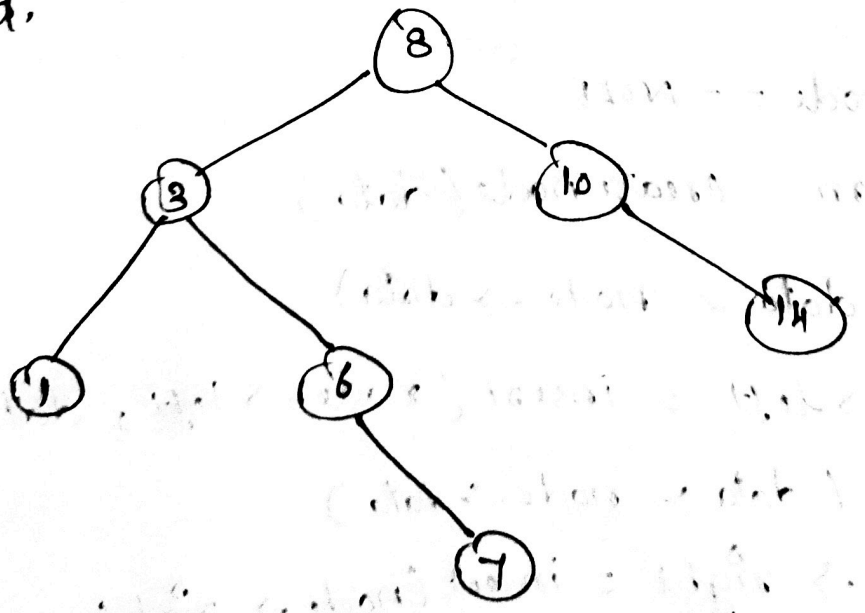
case 1

⇒ Node to be deleted is the leaf node.

⇒ Simply delete the node from the tree.



node to be deleted.

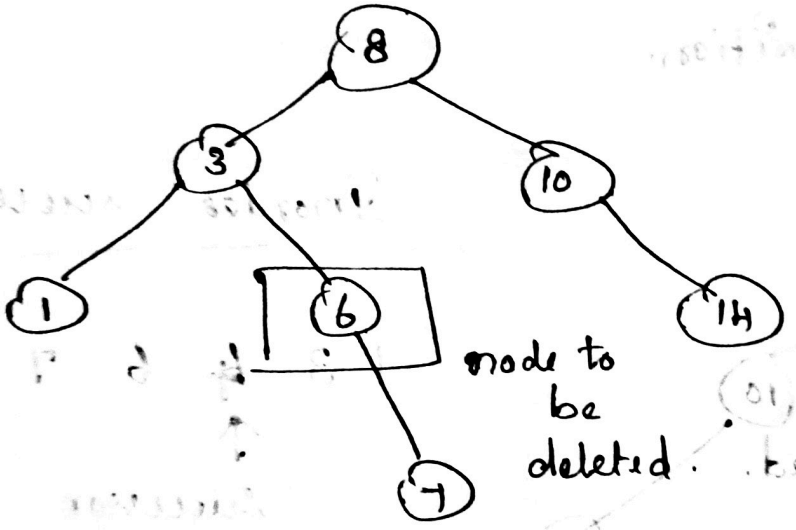


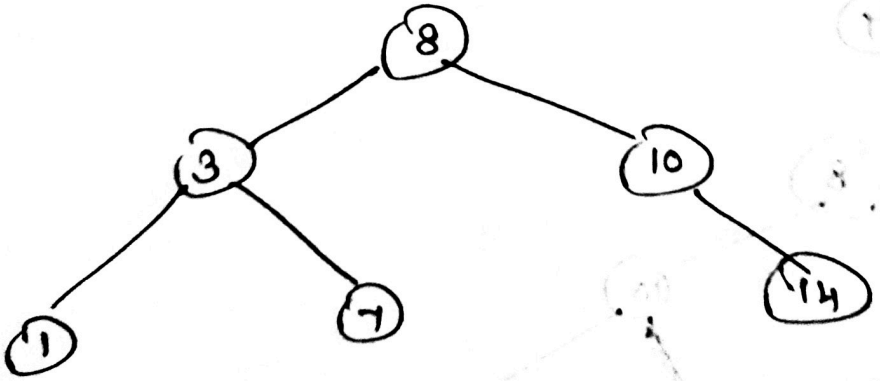
Case 2

→ node to be deleted has a single child node.

(1) Replace that node with its child node.

(2) Remove the child node from its original position.





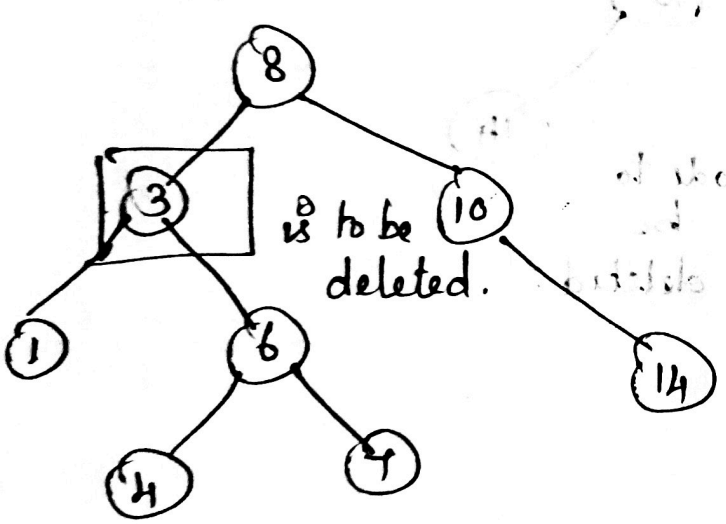
Case 3

⇒ node to be deleted has two children.

(1) Get the Inorder successor of that node

(2) Replace the node with the Inorder successor.

(3) Remove the Inorder successor from its original position.

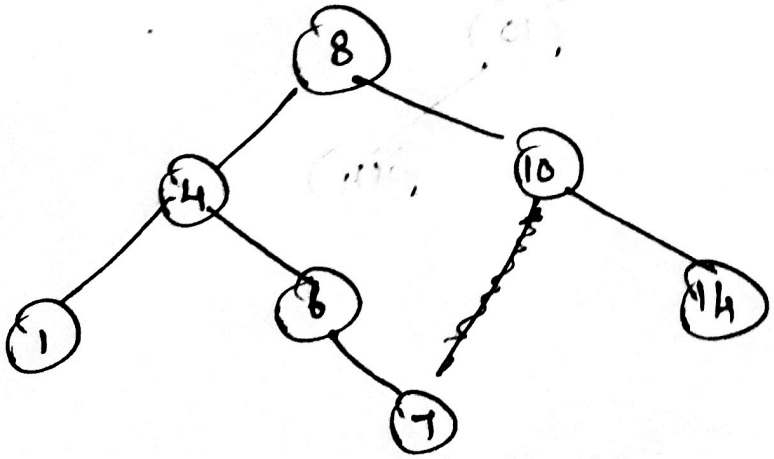


Inorder successor

1 3 4 6 7 8 10 14.

↑

successor.



Heap Sort

⇒ efficient sorting technique based on the heap data structure.

⇒ Heap ⇒ complete binary tree

↳ parent node → minimum or maximum

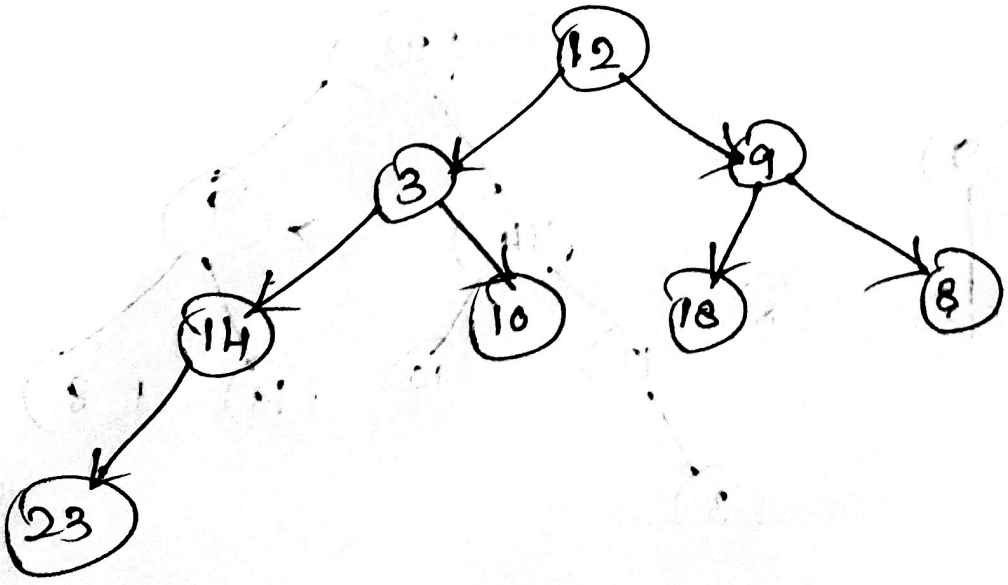
⇒ Heap with minimum root node ⇒ min heap

⇒ Heap with maximum root node ⇒ max heap.

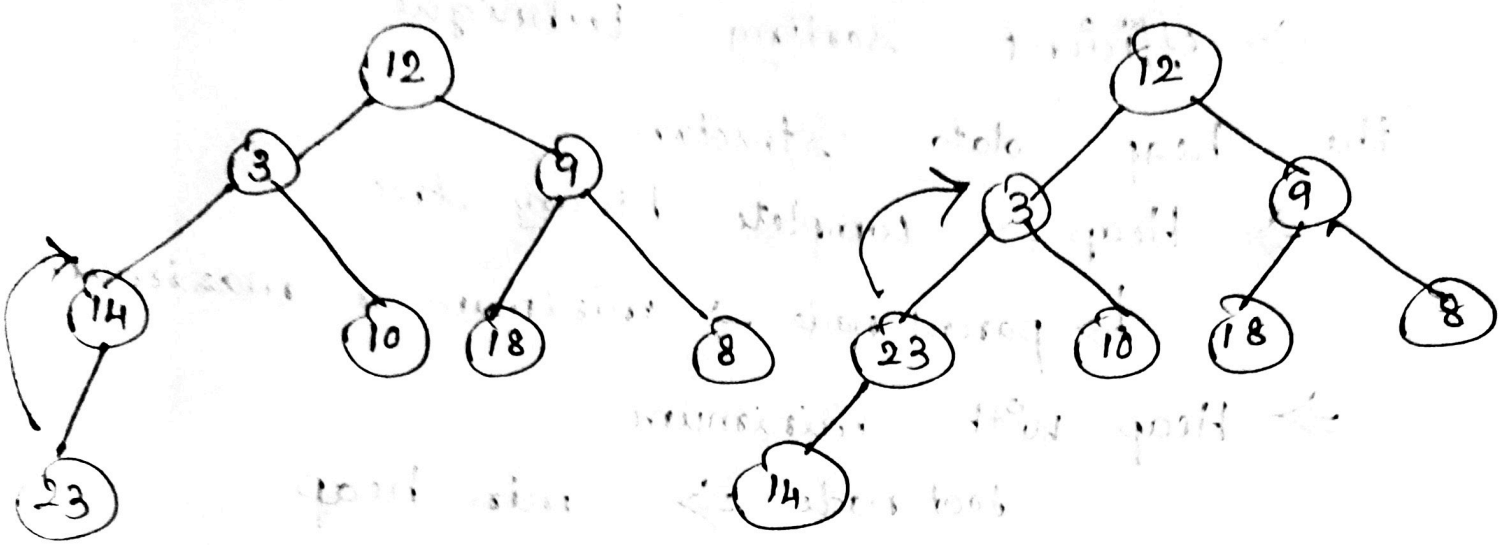
(Eg)

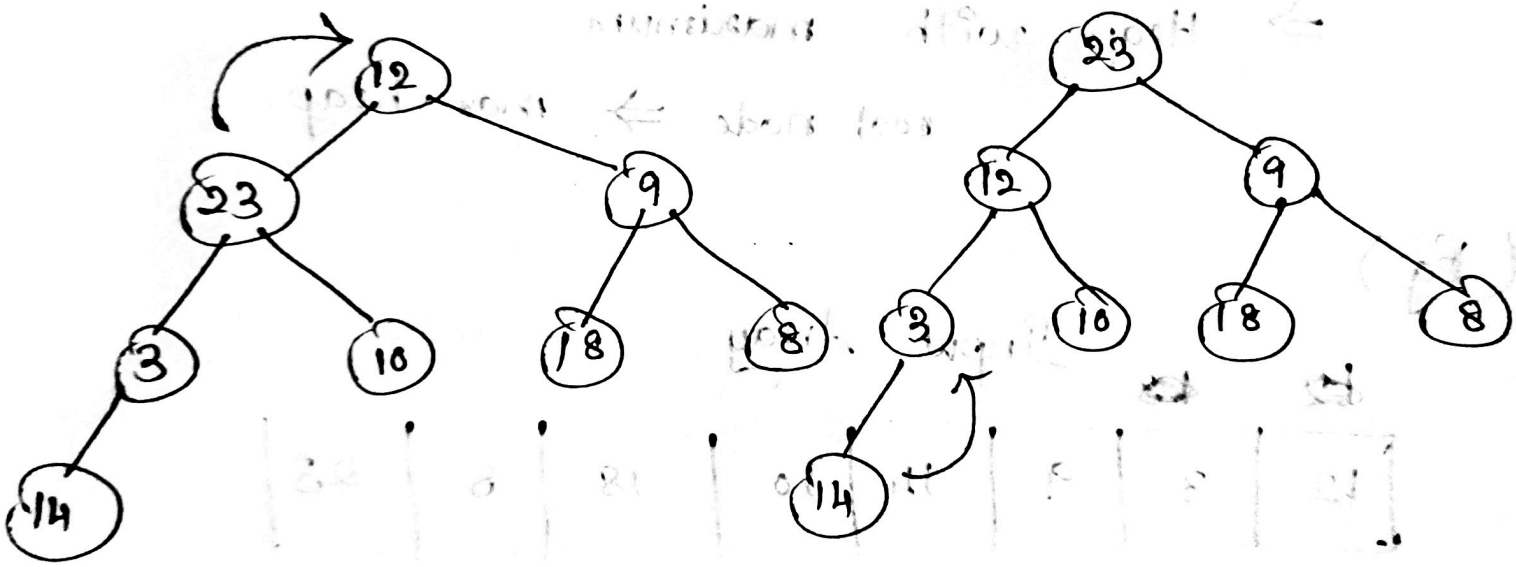
Input Array

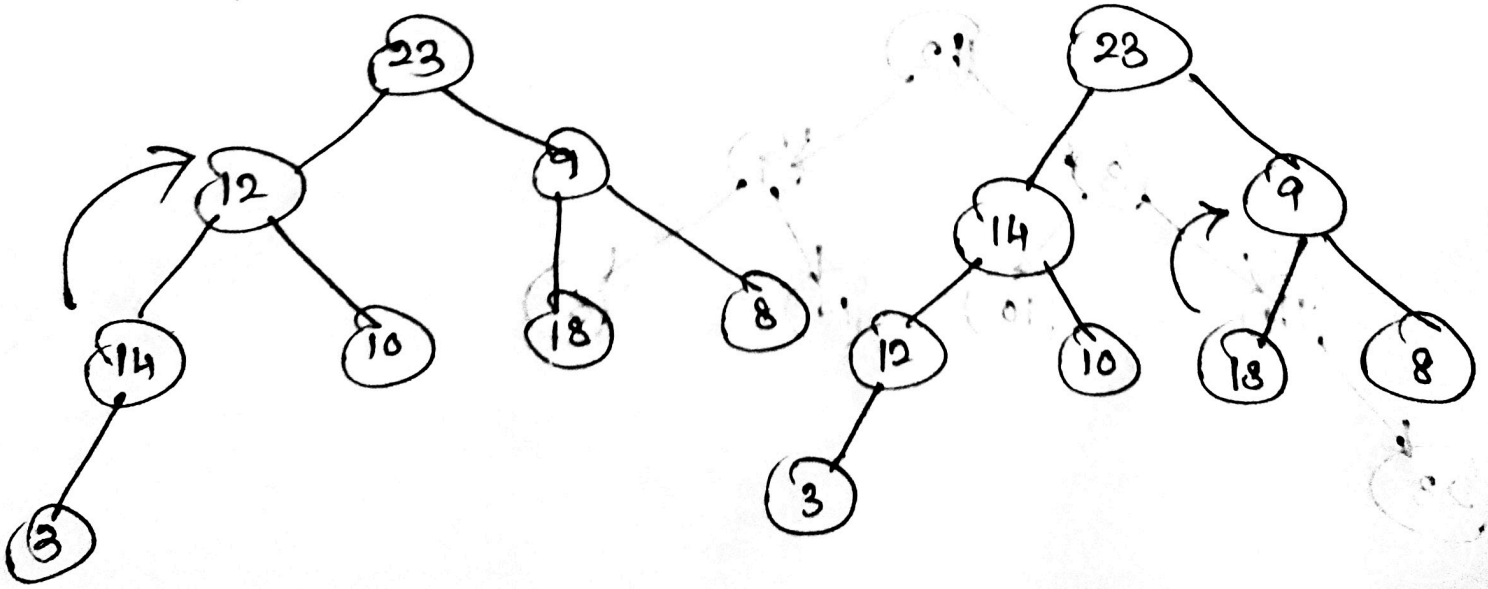
12	3	9	14	10	18	8	23

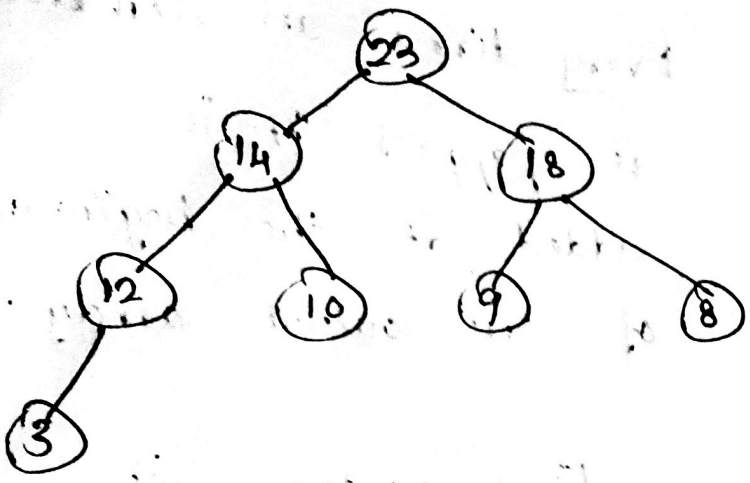


Build max heap









Heapify algorithm

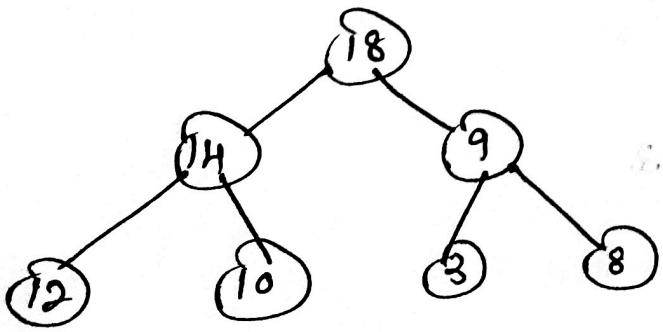
Remove the root node from the heap.

replace it with the next maximum valued child of the root

23 is root

23 is popped

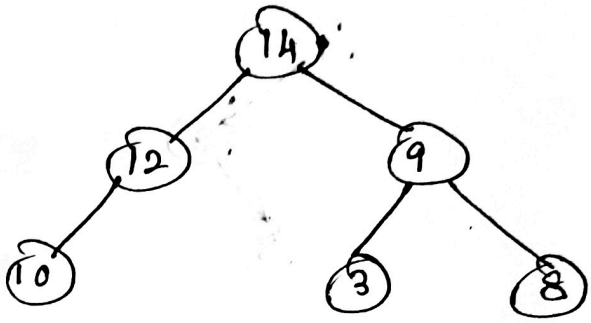
18 is root now



23

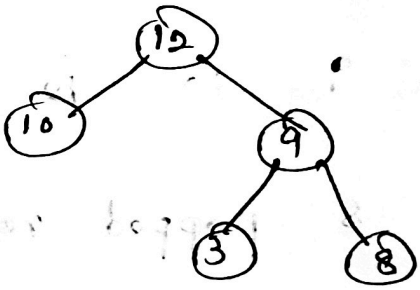
18 is popped

replace by 14



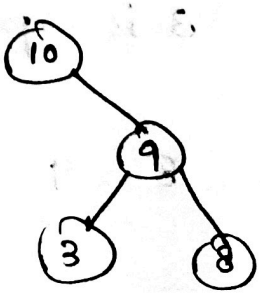
18 23

14 popped replaced by 12



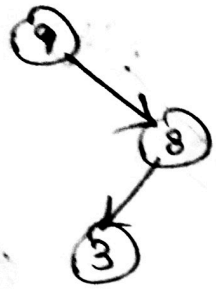
14 18 23

12 popped replaced by 10

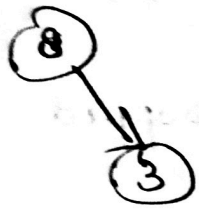


12 14 18 23

10 popped
replaced by 9:



10 12 14 18 23



9 10 12 14 18 23

8 popped replaced by 3

3

8 9 10 12 14 18 23

3 is popped.

3 8 9 10 12 14 18 23

Every time an element is popped it is added at the beginning of the output array.

Final sorted list

Output array.

3	8	9	10	12	14	18	23

Assignment-4

⇒ Minheap
⇒ Expression tree.
⇒

Hashing

⇒ process of mapping keys, values into the hash table by using a hash function.

⇒ process of converting a given key into another value with the help of a hash function.

Hash function

→ Mathematical algorithm which help generate a new value for a given input

→ result of the hash function is called a hash or hash value.

Good hash function

→ efficiently computable

→ should uniformly distribute the keys.

Hash table

→ is a data structure that implements an associative array abstract data type, a structure that can map keys to values.

→ Hash table uses a hash function to compute an index, also called a hash code into an array of buckets or slots

Collision

→ Two or more keys can generate, same hash value sometimes.

→ This is called a collision.

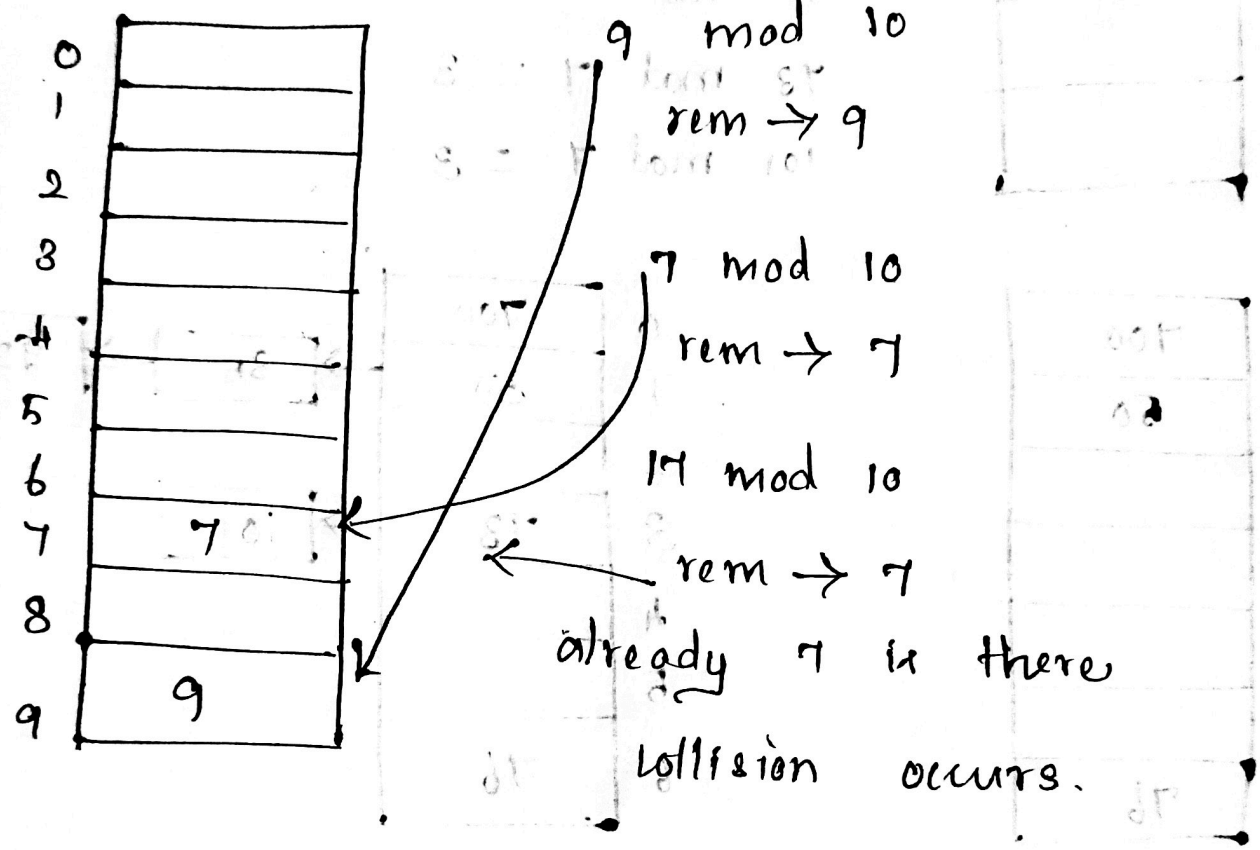
→ collision can be handled using various techniques.

1. Separate chaining
2. Open addressing
   - → Linear probing
   - → Quadratic probing
   - → Double hashing.

Collision situation

values - 9, 7, 17, 18, 19, 8

Hash fn $h(k') = h(k) \bmod m$.



Separate chaining

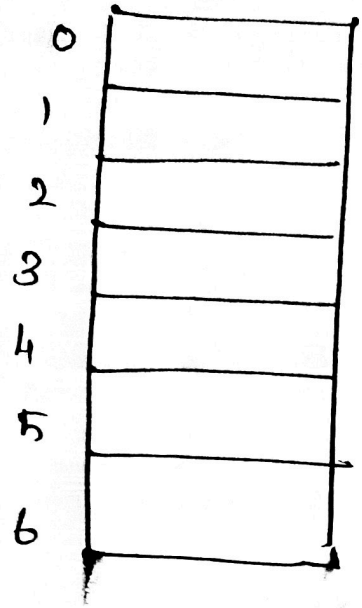
50, 700, 76, 85, 92, 73, 101

Empty hash table created

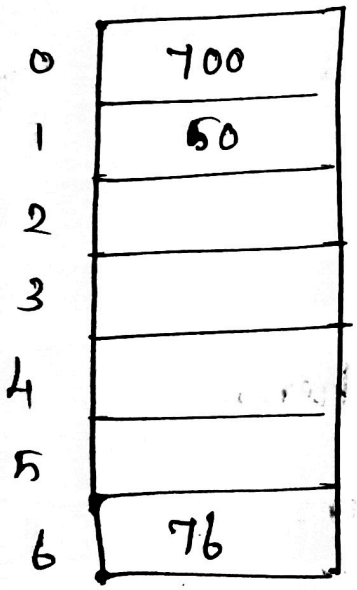
range [0, 6]

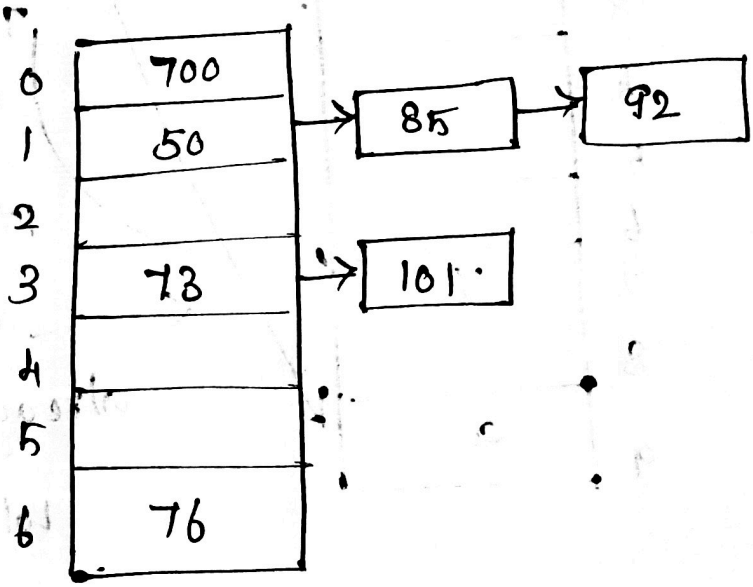
7 buckets

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



50 mod 7 = 1
700 mod 7 = 0
76 mod 7 = 6
85 mod 7 = 1
92 mod 7 = 1
73 mod 7 = 3
101 mod 7 = 3





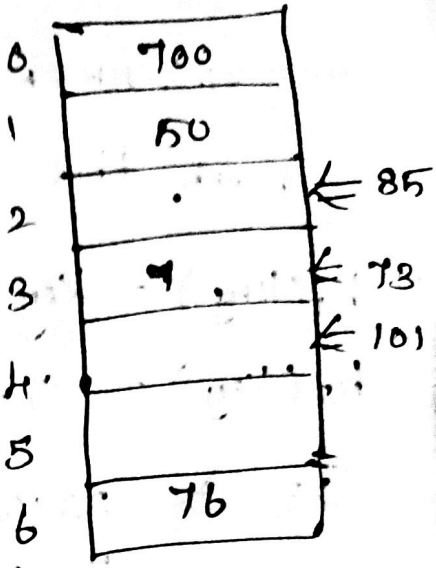
Open addressing

Linear probing

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

0	700
1	50
2	
3	
4	
5	
6	76

50 mod 7 = 1
700 mod 7 = 0
76 mod 7 = 6
85 mod 7 = 1
92 mod 7 = 1
73 mod 7 = 3
101 mod 7 = 3



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

Quadratic probing

73 + (0 * 0) mod 7

73 + 0 mod 7

73 mod 7 ⇒ 3

101 + (0)^2 mod 7

101 mod 7

101 + (1)^2 mod 7

102 mod 7 ⇒ 4

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

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

85 + 0^2 mod 7 ⇒ 1

85 mod

85 + (1*1) mod 7

86 mod 7 ⇒ 2

92 + (2*2) mod 7

96 mod 7 ⇒ 5

92 + 0^2 mod 7 = 1

92 + 1^2 mod 7 =

93 mod 7 = 2

92 + 2^2 mod 7 =

96 mod 7 = 5

Double hashing

23, 34, 45, 56, 67, 78, 89, 90

2 hash fns

$h_1(key) = key \% 10$

$h_2(key) = 7 - (key \% 7)$

$23 \Rightarrow 23 \% 10 \Rightarrow 3$

$7 - (23 \bmod 7) \Rightarrow 5$

$34 \Rightarrow 4, 4$

$45 \Rightarrow 5, 3$

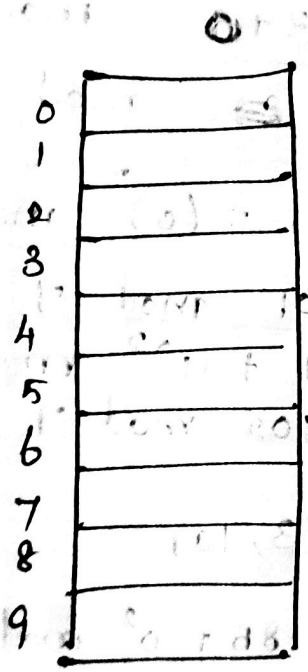
$56 \Rightarrow 6, 2$

$67 \Rightarrow 7, 1$

$78 \Rightarrow 8, 7$

$89 \Rightarrow 9, 6$

$90 \Rightarrow 0, 4$



Double hashing

67, 90, 55, 17, 49

67 %.10 = 7

90 %.10 = 0

55 %.10 = 5

17 %.10 = 7

= 7 - (17 %.7)

= 7 - 3

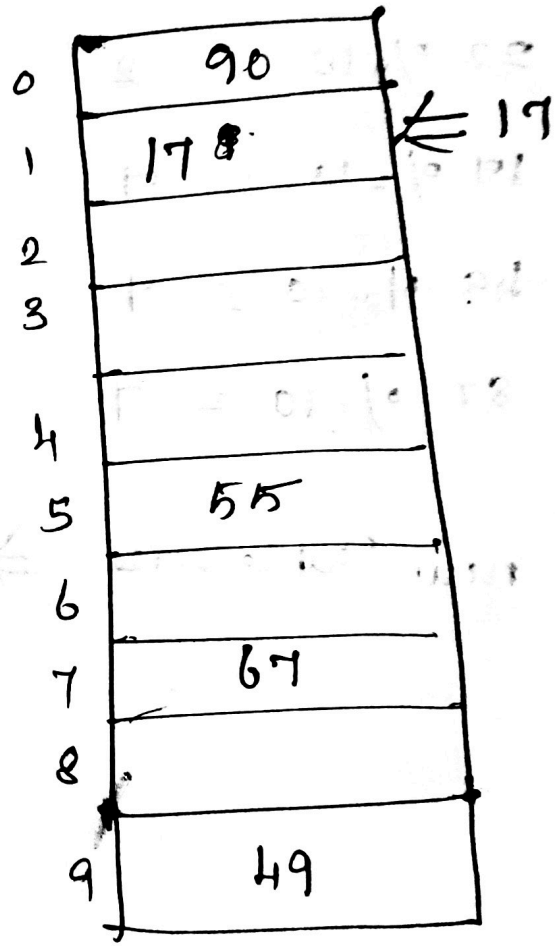
= 4

49 %.10 = 9

2 hash fns

① $h_1(key) = key \%.10$

② $h_2(key) = 7 - (key \%.7$



Rehashing

⇒ Table is resized (ie) size of the table is doubled by creating a new table.

Size of table ⇒ prime number is preferable.

(Eg)

37, 90, 55, 22, 17, 49, 87

37 %. 10 = 7

90 %. 10 = 0

55 %. 10 = 5

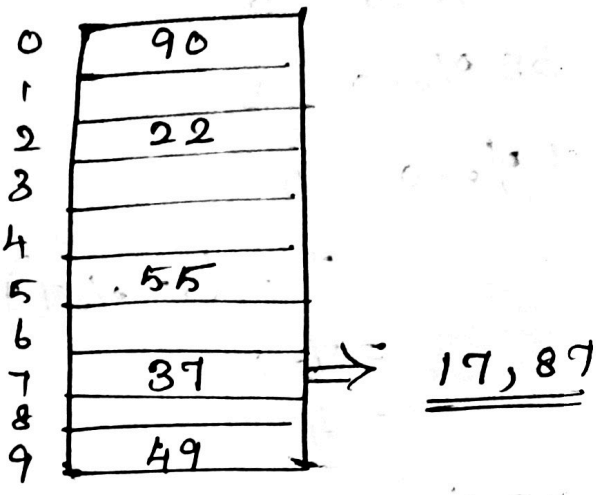
22 %. 10 = 2

17 %. 10 = 7

49 %. 10 = 9

87 %. 10 = 7

old size = 10



new Table size ⇒ resize the table by doubling 20 ⇒ but 20 is not prime ⇒ so 23 is the table's size

37 %. 23 = 14

90 %. 23 = 21

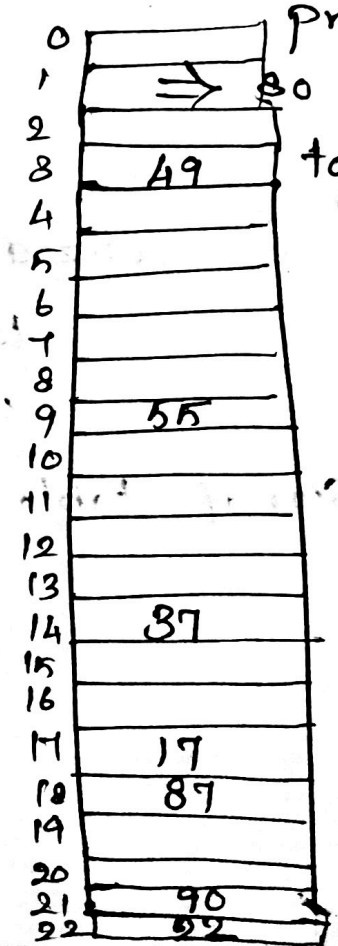
55 %. 23 = 9

22 %. 23 = 22

17 %. 23 = 17

49 %. 23 = 3

87 %. 23 = 18



Unit - 5

Sorting and Searching

Insertion Sort - Quick Sort - Heap Sort - Merge Sort - Linear Search - Binary Search.

Insertion Sort:

Sorting:

⇒ Sorting is the process of placing a list of elements from the collection of data in some order.

⇒ It is nothing but storage of data in sorted order.

⇒ Sorting can be done in ascending and descending order.

⇒ It arranges the data in a sequence which makes the searching easier.

Insertion Sort:

⇒ In this sorting technique, first elements are stored in an array.

⇒ The process of sorting starts with second element.

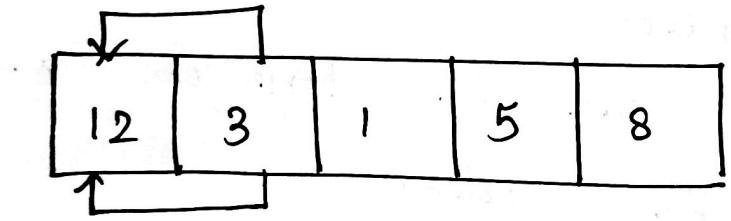
⇒ First, the second element is picked and is placed in specified order. Next third element is picked and is placed in specified order.

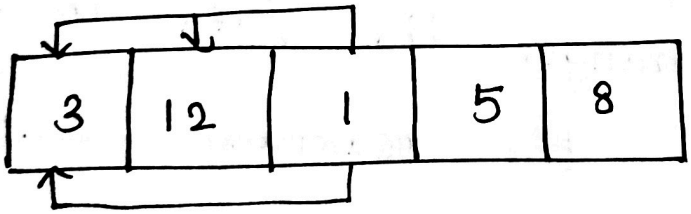
⇒ Similarly the fourth, fifth.. nth element is placed in specified order.

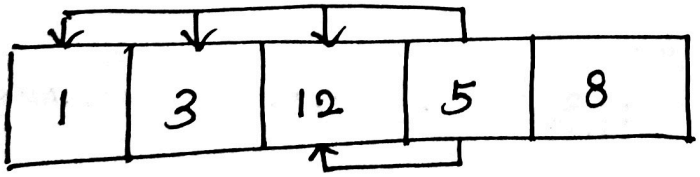
⇒ Finally we get the sorting elements

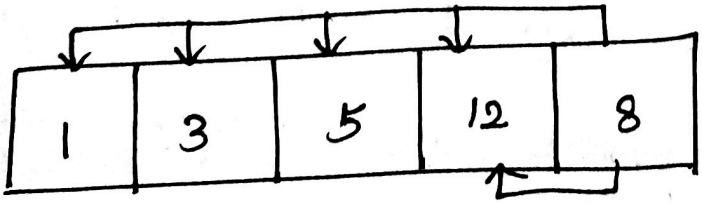
(Eg)

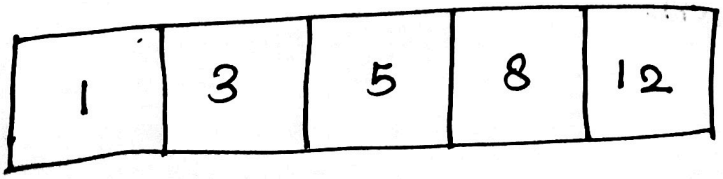
Let us consider the elements 12, 3, 1, 5, 8.











Program:

```
#include <stdio.h>
int main()
{
int n, a[20], key, i, j;
printf("Enter the total elements:\n");
scanf("%d", &n);
printf("Enter the elements:\n");
for(i=0; i<n; i++)
scanf("%d", &a[i]);
for(i=1; i<n; i++)
{
  j=i;
  while(j>0 && a[j]<a[j-1])
    {
      temp = a[j];
      a[j] = a[j-1];
      a[j-1] = temp;
      j--;
    }
}
  printf("After Sorting is:\n");
  for(i=0; i<n; i++)
  printf("%d", a[i]);
  return 0;
}
```

Quick Sort:

⇒ Quick sort is also one of the exchange sort.

⇒ In a Quick sort we take pivot element, then we place all the smaller elements are on one side of pivot and greater elements are on the other side of pivot.

⇒ After partitioning we have pivot in the final position. After repeadedly partitioning we get the sorted elements.

Example:

⇒ Let us consider the elements.

35, 50, 15, 25, 80, 20, 90, 45.

pivot = 35 = i , j = 45.

(35)	50	15	25	80	20	90	45
↑							↑
pivot / first / i							j

50 > i

20 < i taken as j

(35)	50	15	25	80	20	90	45
	↑ i				↑ j		

i < j swap i & j.

(35)	20	15	25	80	50	90	45

35 > < 80 = i 25 < 35 = j

35	20	15	25	80	50	90	45
			↑ j	↑ i			

i & j swap 35 & j

25	20	15	(35)	80	50	90	45
			↑ j				

repeat (a, first, j-1)

repeat (a, j+1, last).

Left side

25 20 15

↓ j i

pivot

Swap

15 20 25.

Left part is sorted.

80 50 90 45

↓ ↓ ↓

pivot i j

i < j swap i & j

(80) 50 45 90

> 80 is i and < 80 is j

i > j swap 80 and j

(80) 50 45 90

↓ ↓

j j

45 50 80 90. → sorted.

Joining left and the right part we get the sorted elements.

15 20 25 35 45 50 80 90.

Program:

```
#include <stdio.h>
void quicksort (int a[25], int first,
                int last)
{
```

```
int i, j, pivot, temp;
if (first < last)
 {
  pivot = first;
  i = first;
  j = last;
  while (i<j)
  {
    while (a[i]<=a[pivot] && i <= last)
     i++;
    while (a[j] > a[pivot]
    j--;
    if(i<j)
     {
      temp = a[i];
      a[i] = a[j];
      a[j] = temp;
     }
  }
  temp = a[pivot];
  a[pivot] = a[j];
  a[j] = temp;
  quicksort (a, first, j-1);
  quicksort (a, j+1, last);
 }
}
```

```
int main()
{
  int i, n, a[25];
  printf("Enter the total number of elements :\n");
  scanf("%d", &n);
  printf("Enter the elements:\n");
  for(i=0; i<n; i++)
  scanf("%d", &a[i]);

  quicksort(a, 0, n-1);
  printf("The sorted elements are:\n");
  for(i=0; i<n; i++)
  printf("%d", a[i]);

  return 0;
}
```

Heap sort:

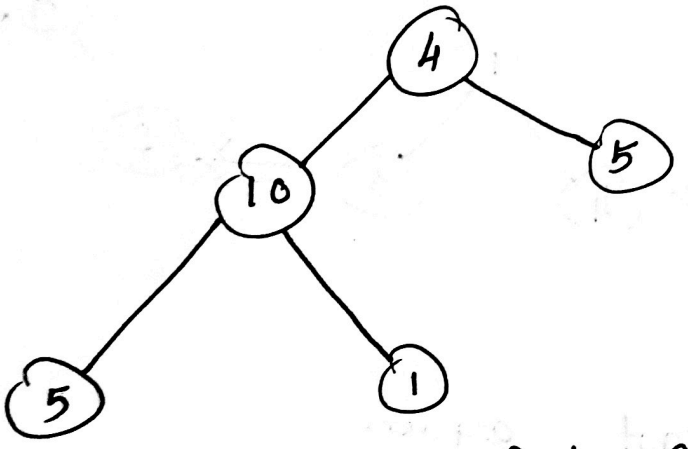
⇒ Heap sort is a comparison based sorting technique based on Binary heap data structure.

⇒ It is similar to the selection sort where we first find the minimum element at the beginning.

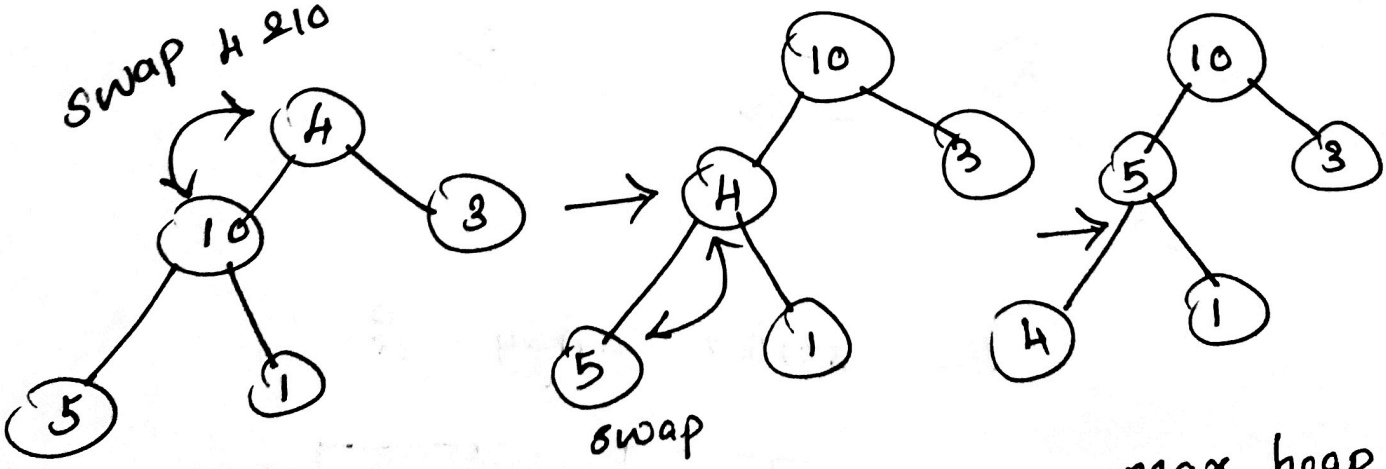
⇒ Repeat the same process for the remaining elements.

Consider the elements 4, 10, 3, 5, 1.

1. Build complete binary tree.

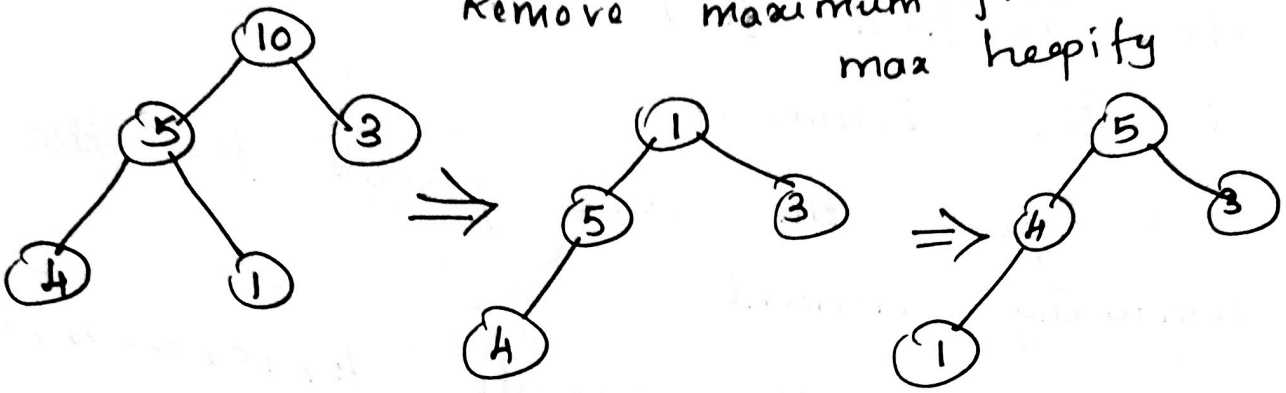


2. Transform into max heap.

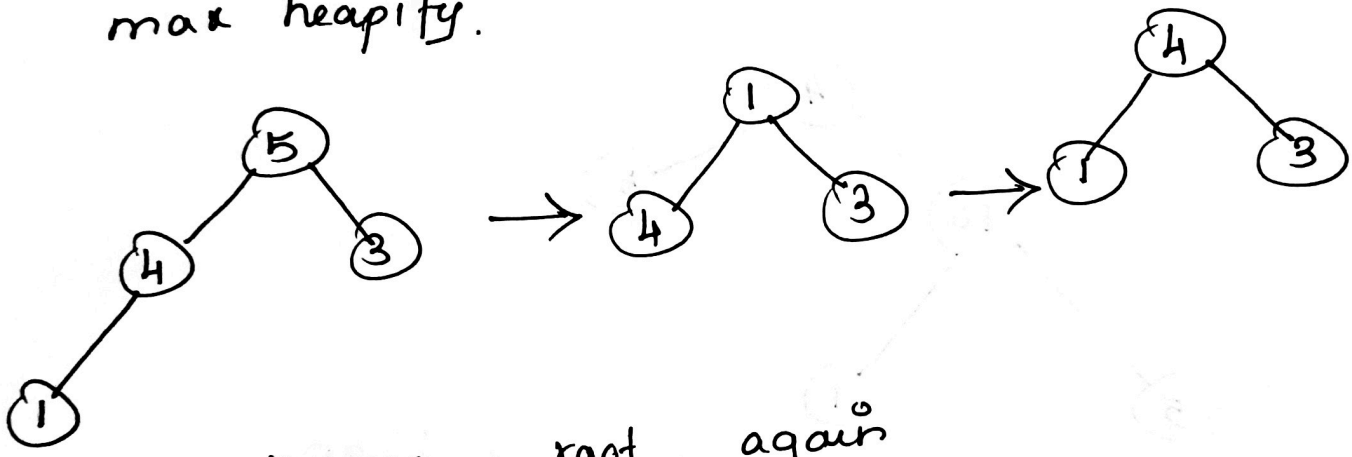


3. Perform heap sort.

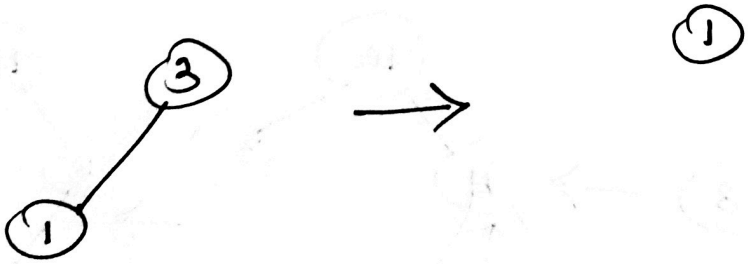
Remove maximum from root & max heapify



Remove next maximum from root & max heapify.



remove root again



The sorted array is

1	3	4	5	10

Program:

```
#include <stdio.h>
void heapify(int *, int, int);
void heapsort(int *, int);
void print_array(int *, int);
int main()
{
int arr[] = {10, 30, 5, 63, 22, 12, 56, 33};
int n = sizeof(arr);
printf("\nArray before Sorting: \n");
print_anay(arr, n);
heapsort(arr, n);
printf("\n\n Array after sorting: \n");
print_array(arr, n);
return 0;
}
void heapsort(int * arr, int n)
{
for(int i= n/2-1; i>=0; i--)
  {
  heapify(arr, n, i);
  }
```

```
for (int i = n-1; i>=0; i--)
{
  int temp = arr[i];
  arr[i] = arr[0];
  arr[0] = temp;
  heapify (arr, i, 0);
}
}

void heapify (int *arr, int n, int i)
{
  int largest = i;
  int left = 2*i+1;
  int right = 2*i+2;
if (left < n && arr[left] > arr[largest])
{
  largest = left;
}
if (right < n && arr[right] > arr [largest])
{
  largest = right;
}
if (largest != i)
{ int temp = arr[i];
```

```
    arr[i] = arr[largest];
    arr[largest] = temp; heapify(arr, n, largest);
  }
}
void print_array(int *arr, int n)
{
  for(int i=0; i<n; i++)
  {
    printf("%d", arr[i]);
  }
}
```

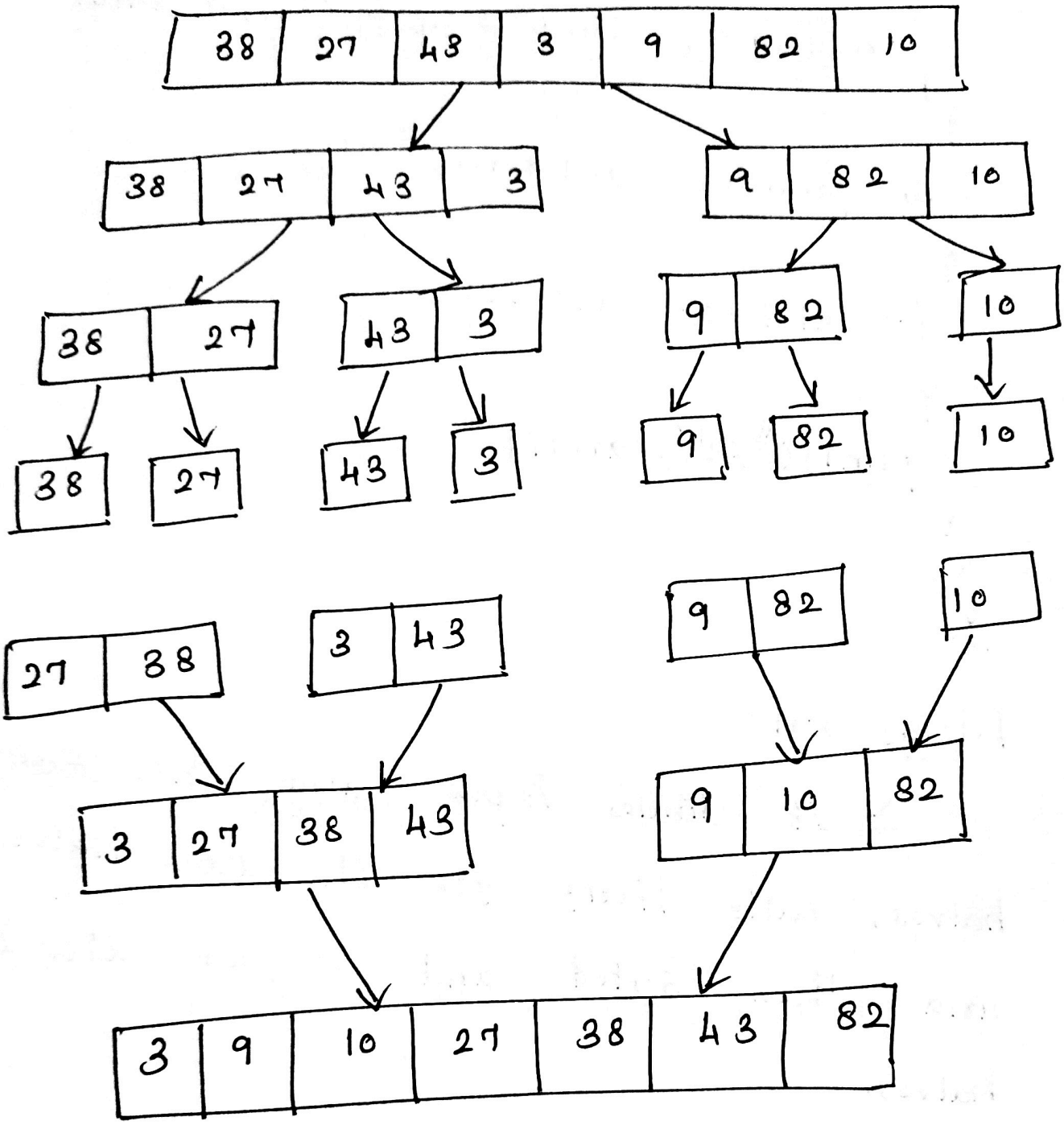
Merge Sort:

⇒ It divides input array into two halves, calls itself for the two halves and then sorted and merged the two halves.

(Eg)

38, 27, 43, 3, 9, 82, 10.

The array is recursively divided into two halves till the size becomes one.



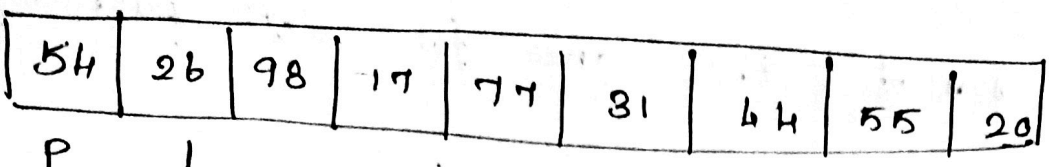
Program:

```
# include <stdio.h>
int n, a[30], i, j, k, temp[30];
void merge (int low, int mid, int high)
{
```

```
    i = low;
    j = mid+1
    k = low;
    while ((i<=mid) && (j<=high))
    {
      if(a[i] >= a[j])
      temp[k++] = a[j++];
      else
      temp[k++] = a[i++];
    }
    while (i<=mid)
    temp[k++] = a[i++];
    while (j<=high)
    temp[k++] = a[j++];
    for(i=low; i<=high; i++)
    a[i] = temp[i];
}
void mergesort (int low, int high)
{
    int mid;
    if (low != high)
    {
      mid = ((low+high)/2);
      mergesort(low, mid);
```

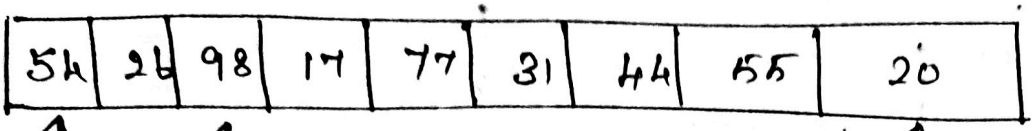
```
        mergesort ( mid+1, high) ;
        merge ( low, mid, high) ;
    }
}

int main()
{
printf (" Enter total elements :\n");
scanf ("%d", &n );
printf (" Enter elements :\n");
for (i=0; i<n; i++)
scanf ("%d", &a[i]);
mergesort (0, n-1);
printf (" After sorting is :\n");
for (i=0; i<n ; i++)
printf ("%d", a[i]) ;
return 0;
}
```



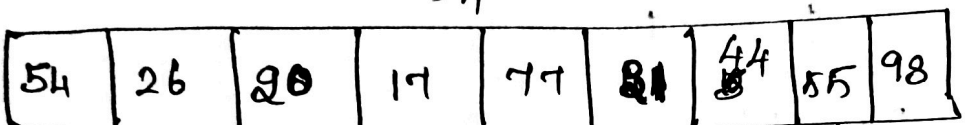
26 < 54 20 < 54

left and right markers will converge on split point.



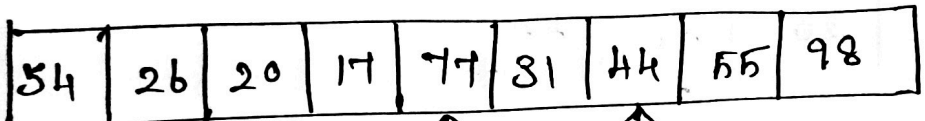
98 > 54
20 < 54

swap 20 & 98



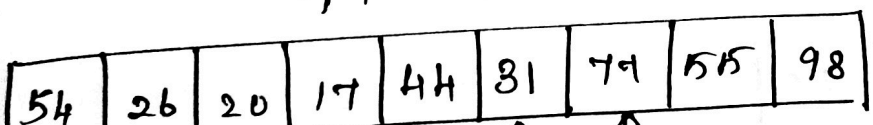
17 < 54

55 > 54 no change.



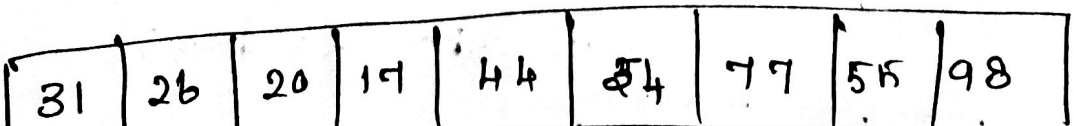
77 > 54

44 < 54 swap.



31 < 54 split point found.
77 > 54

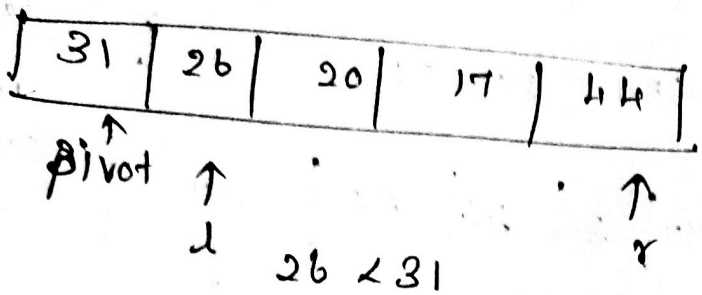
swap 31 & 54.



l < p p r > p.

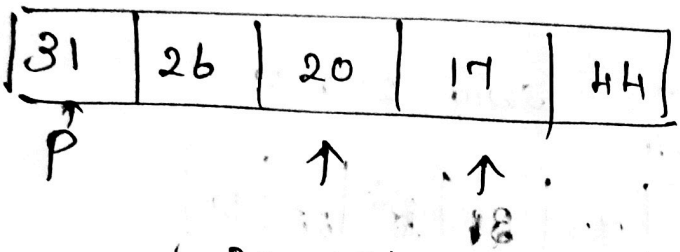
position found for pivot element

Now repeat the same for left partition



26 < 31

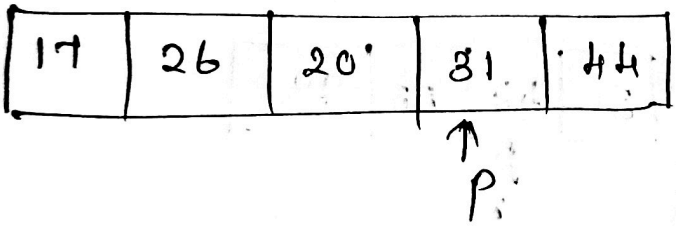
44 > 31



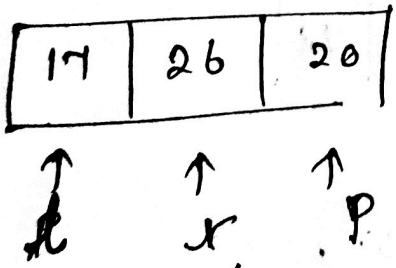
20 < 31

17 < 31

Swap 17 & 31.



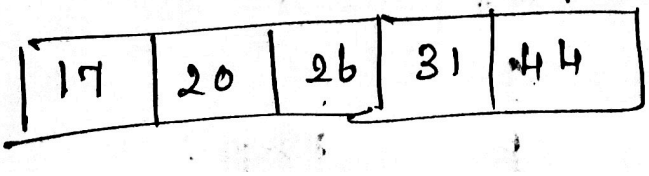
Again partition.



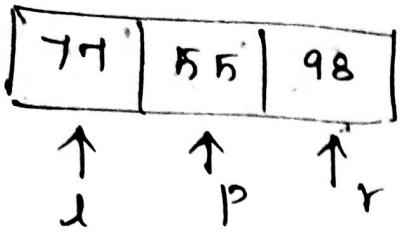
17 < p (20) i.e. 17 < 20, 26 > 20

26 > p, also

swap 20 & 26:

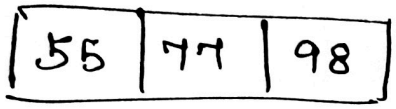


Again do for right partition.

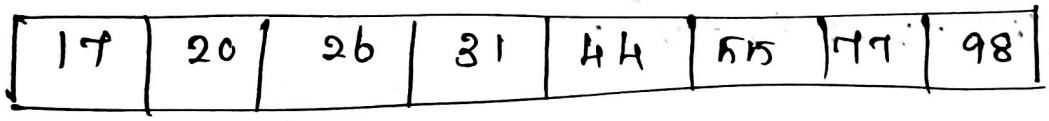


split point

77 > 55
98 > 55] swap 55 & 77.



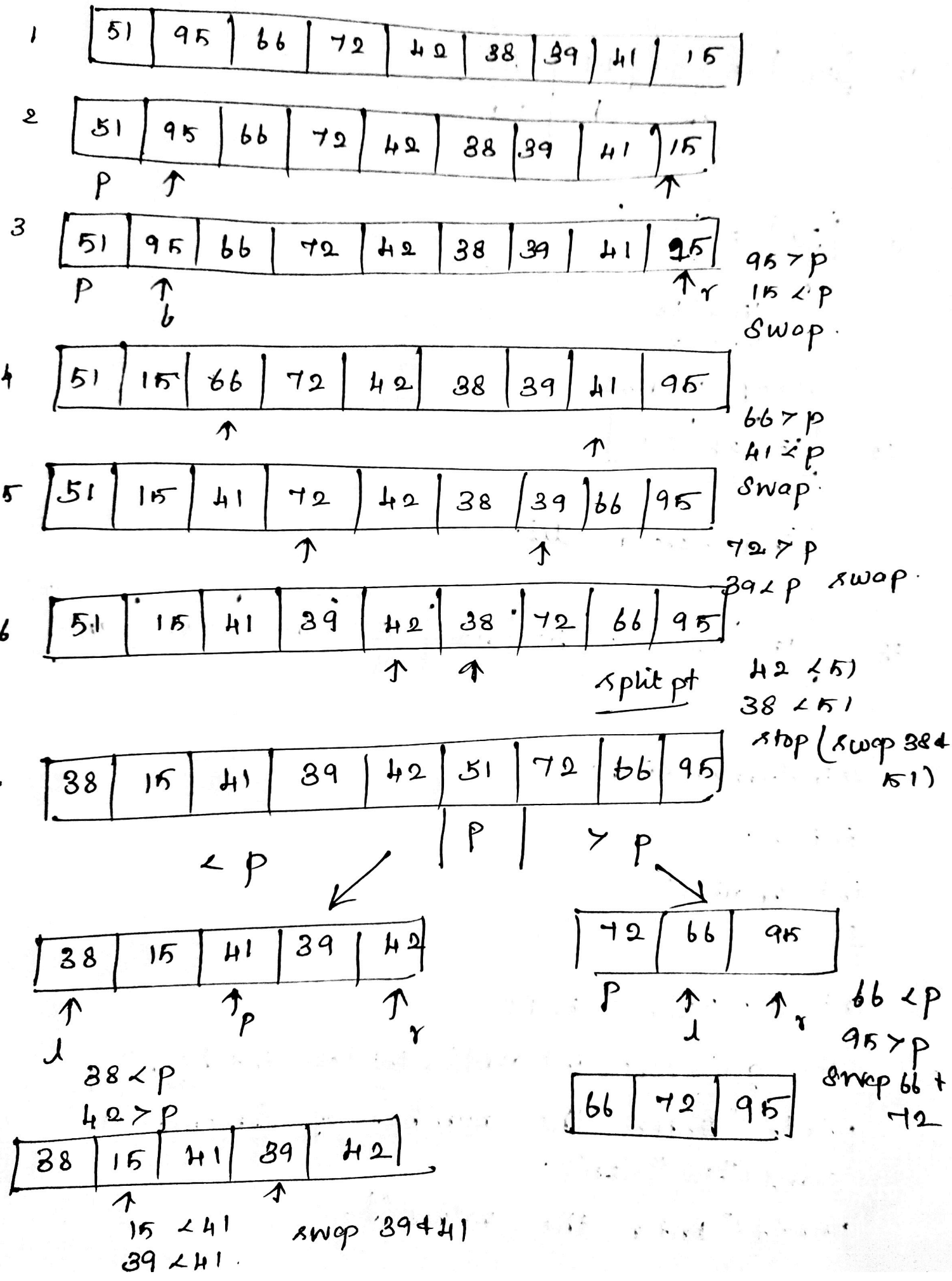
Now the entire list gets sorted.



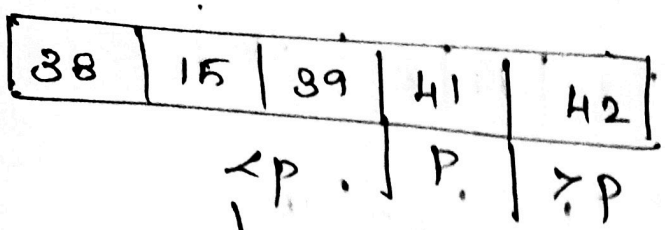

```
void quick_sort (int a[], int l, int u)
{
  int j;
  if (l<u)
  {
    j = partition(a, l, u);
    quicksort (a, l, j-1);
    quicksort (a, j+1, u);
  }
}
```

```
int partition (int a[], int l, int u)
{
  int v, i, j, temp;
  v = a[l];
  i = l;
  j = u+1;
  do
{
do
{
i++;
while (a[i] < v && i <= u);
do
j--;
while (v < a[j]);
 if (i < j)
{
temp = a[i];
 a[i] = a[j];
 a[j] = temp;
}} while (i < j);
a[l] = a[j];
a[j] = v;
return (j);
}.
```

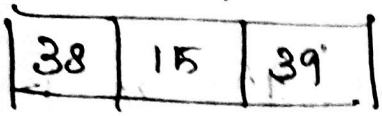
6	4	7	8	1


1
51 95 66 72 42 38 39 41 15
2
51 95 66 72 42 38 39 41 15
p
3
51 95 66 72 42 38 39 41 95
95 > p
15 < p
swap
4
51 15 66 72 42 38 39 41 95
66 > p
41 < p
swap
5
51 15 41 72 42 38 39 66 95
72 > p
39 < p swap
6
51 15 41 39 42 38 72 66 95
split pt
42 < 51
38 < 51
stop (swap 38 & 51)
7
38 15 41 39 42 51 72 66 95
< p
p
> p
8
38 15 41 39 42
38 < p
42 > p
72 66 95
66 < p
95 > p
swap 66 & 72
66 72 95
9
38 15 41 39 42
15 < 41
39 < 41
swap 39 & 41

10



11



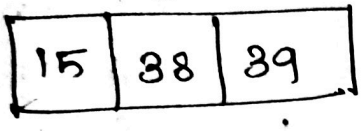
p

15 < 38

39 > 38

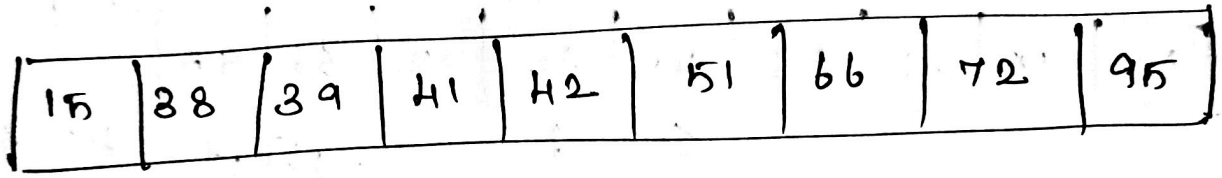
Swap 15 & 38

12



Final sorted list.

13.




```
#include<stdio.h>
int n;
int main()
{
int arr[30], j, x, i;
void quicksort(int arr[], int low, int high);
printf("Enter the number of elements:");
scanf("%d", &n);
printf("Enter the values");
```

Quick Sort

⇒ Based on Divide and conquer algorithm.

⇒ key process ..

⇒ Any element is chosen as a pivot element.

⇒ put all the smaller elements to the left of the pivot.

⇒ All the greater elements to the right of the pivot.

⇒ partition is done recursively on each side of the pivot after the pivot is placed in its correct position and this will finally sort the array.

⇒ There are many different choices for picking pivots.

(i) First element as a pivot
(ii) last element as a pivot
(iii) random element as a pivot
(iv) middle element as a pivot

Split point

→ actual position where the pivot value belongs in the final sorted list.

→ Two markers are located left mark and right mark.

→ goal

more the items that are on the wrong side with respect to the pivo value.

Expression Tree Construction

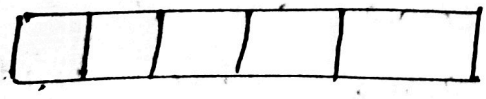
(1) Stack is used to construct the expression tree.

Do the following for every character.

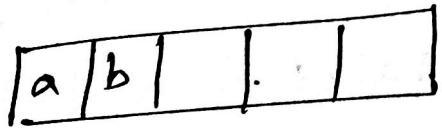
(i) if a character is an operand push that into the stack.

(2) if a character is an operator pop two values from the stack make them its child and push the current node again.

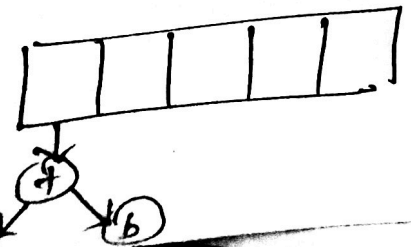
a b + c *



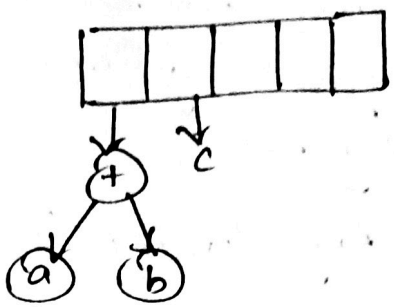
read a, b



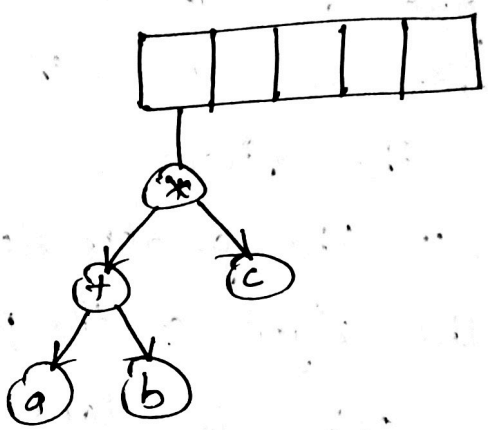
read +



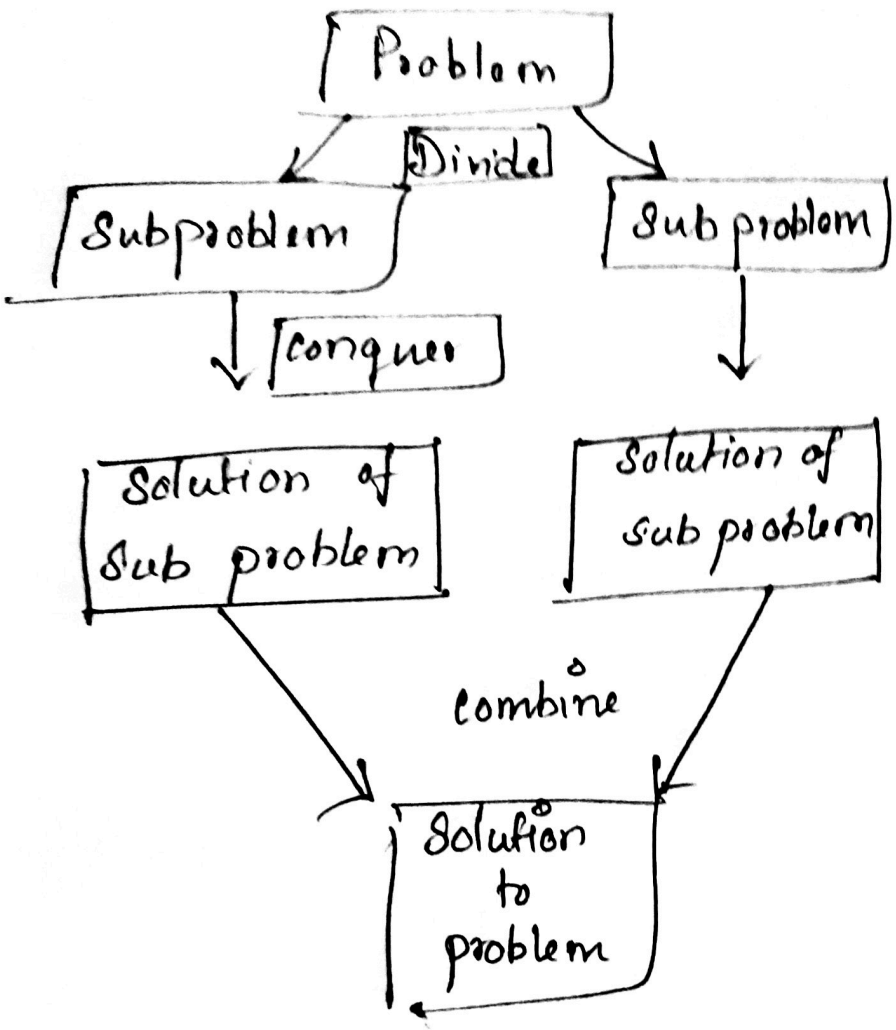
read c



read *



⇒ Divide and conquer



Insertion Sort

(Eg) 12, 11, 13, 5, 6

1st pass

(i) Compare the first two elements of the array

12 11 13 5 6

12 > 11

So swap 11 & 12.

11 12 13 5 6

2nd pass

11 12 13 5 6

12 & 13 are in ascending order

3rd pass

11 12 13 5 6

Swap 5 & 13.

11 12 5 13 6

Swap 5, 12

11 5 12 13 6

Swap 11 & 5

5 11 12 13 6

4th pass

5 11 12 13 6

5 11 12 6 13

5 11 6 12 13

5 6 11 12 13

Insertion Sort Algorithm.

1. Iterate from arr[1] to arr[N] over the array.
2. Compare the current element to its predecessor, compare it to the elements before.
3. Move the greater elements one position up to make space for the swapped element.

```c
int main (void)
{
 int n, i, j, temp;
 int arr[64];
 printf(" Enter number of elements");
 scanf("%d", &n);
printf("Enter %d integers \n",n);
 for (i=0 ; i<n ; i++)
 {
  scanf("%d", &arr[i]);
 }
 for (i=1 ; i<n ; i++)
 {
  j = i ;
while(j>0 && arr[j-1] >arr[j])
 {
   temp = arr[j];
   arr[j] = arr[j-1];
   arr[j-1] = temp;
   j--;
 }
}
 printf ("Sorted list");
 for(i=0; i<n; i++)
 {
  printf("%d\n", arr[i]);
 }
 return 0;
}
```

Enter number of elements
5
Enter 5 integers
12 . 0
11 1
18 2
5 3
6 4

0	1	2	3	4
12	11	18	5	6

i=1, j=1
j>0, arr[0] > arr[1]
12>11, temp = 11
arr[j] = 12
arr[j-1] = 11
j = 0. -j--
swap 11 & 12

0	1	2	3	4
11	12	18	5	6

1st pass

[j=0] i=2 } Now
j=2
j>0, arr[1] > arr[2]
12>18 no.
So no change.

2nd pass.

i=3 } Now
j=3
j>0, arr[2] > arr[3]

0	1	2	3	4
11	12	18	5	6

18 > 5
swap 11 12 5 18 6
0 1 2 3 4.
j=2, decremented.
j>0, arr[1] > arr[2]
12 > 5
swap

0	1	2	3	4
11	5	12	13	6

j=1 decremented
j>0, arr[0] > arr[1]
11 > 5 swap.
5 11 12 13 6.

3rd pass.

0	1	2	3	4
5	11	12	13	6

$i = 4$
$j = 4$ } Now

$j > 0$, $arr[3] > arr[4]$

$13 > 6$

Swap $13 \leftrightarrow 6$

5	11	12	6	13
0	1	2	3	4

decrement $j \Rightarrow j = 3$

$j > 0$ $arr[2] > arr[3]$

$12 > 6$

Swap $12 \leftrightarrow 6$

5	11	6	12	13
0	1	2	3	4

decrement $j \Rightarrow j = 2$

$j > 0$ $arr[1] > arr[2]$

Swap $11 \leftrightarrow 6$

5	6	11	12	13
0	1	2	3	4

dec $j \Rightarrow j = 1$

$j > 0$ $arr[0] > arr[1]$

$5 > 6$ no.

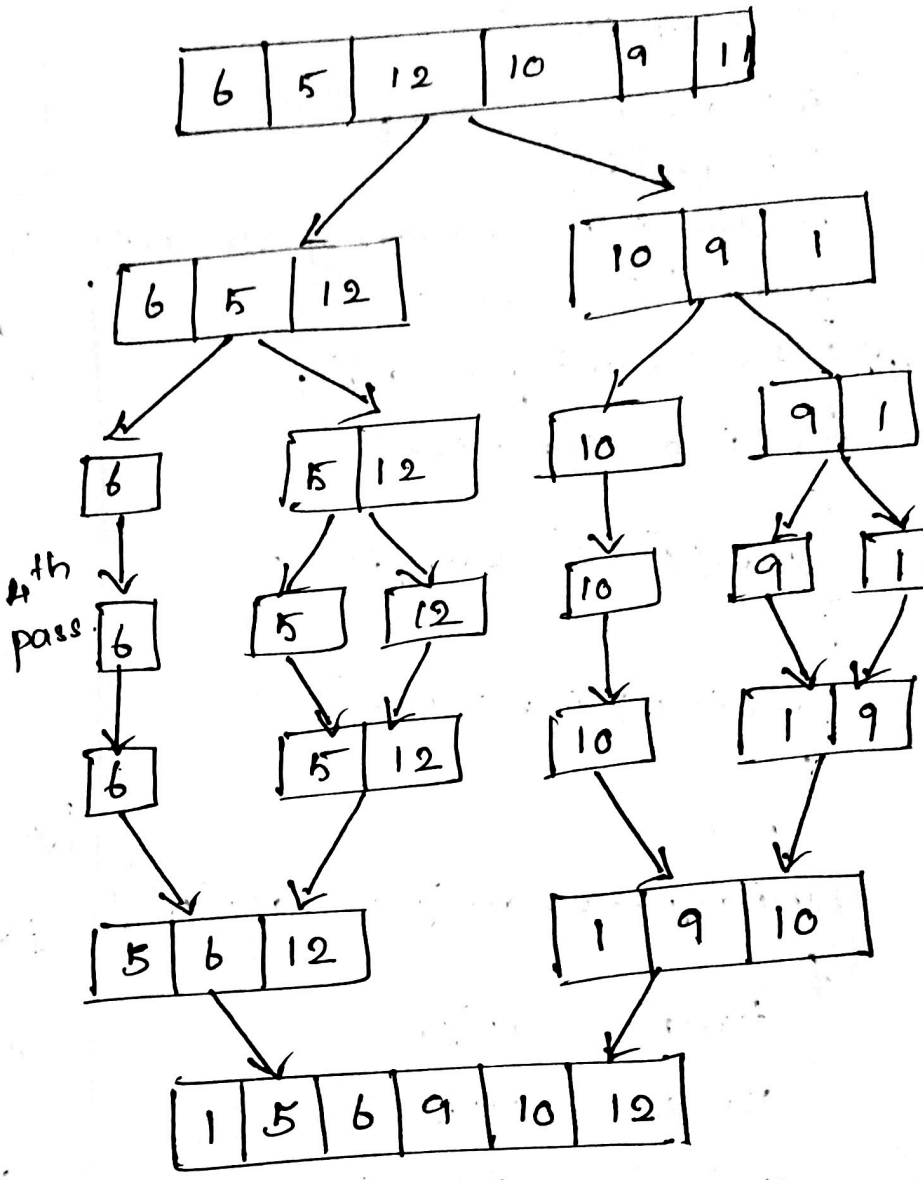
Stop.

Merge Sort

Given

6, 5, 12, 10, 9, 1

unsorted array of elements.



Algorithm

1. Divide the array until the size of each sub array becomes 1.
2. Merge function is called to merge sub arrays into bigger sorted subarrays.
3. Finally fully sorted array is received.

Linear Search:

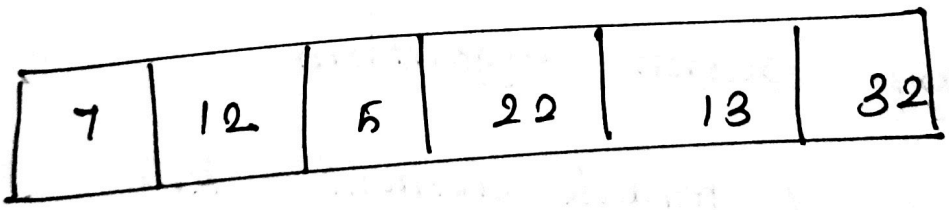
⇒ Searching is an operation or a technique that helps finds the place of a given element or value in the list.

Linear search:

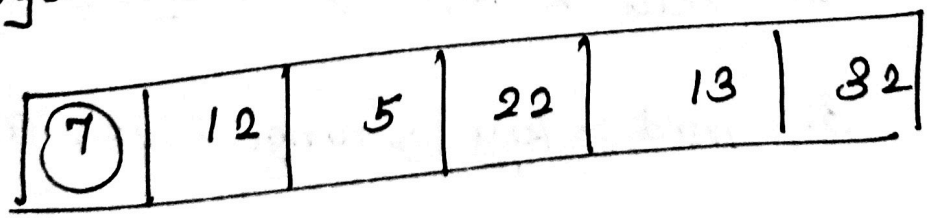
⇒ Basic and simple search algorithm.

⇒ Search an element or value in a given array by traversing the array from the starting.

⇒ Look at each item starting from the first to search the given element



target = 13..



13

7 (12) 5 22 13 32

(13)

7 12 (5) 22 13 32

(13)

7 12 5 (22) 13 32

(13)

7 12 5 22 (13) 32

target = (13)

Binary Search:

⇒ List must be sorted before using Binary Search algorithm.

1. middle element = key
2. mid < key → range [mid+1, high]
3. mid > key → range [low, mid-1]

Eg.

2	3	7	7	9	15	16	18	20	21

target = 9 comparison = 1

Iteration	low	mid	high
1	0	4	9

target = 7 comparison = 3

Iteration	low	mid	high
1	0	4	9
2	0	1	3
3	2	2	3.

Linear Search

⇒ Sequential search algm that starts at one end and goes through each element of the list until the desired element is found.

10	50	30	70	80	60	20	90	40

Key = 70.

10 70
50 70
30 70
70 70 equal found.

```
int search (int arr[], int N, int x)
{
for (int i=0; i<N; i++)
{   if (arr[i] == x)
        return i;
}
    int main()
{ int arr[] = { 2, 3, 4, 10, 40};
    int x = 10;
    int N = sizeof (arr);
```

```
int result = search(arr, N, x);
(result == -1) ? printf("Element is not present
                         in array")
               : printf("Element is present");
return 0;
}
```

Drawback:

⇒ not suited for large arrays.

Uses:

→ For dataset stored in contiguous memory.

→ For small dataset.

Binary Search

⇒ calculate mid element and compare it with the key.

⇒ key < mid

Search space = left

⇒ key > mid

Search space = right.

⇒ Search space

(23) → right

23	88	56	72	91

↑ mid.

→ left.

low + (high - low) / 2.

(23) → found.

0	1	2	3	4	5	6	7	8	9
2	5	8	12	16	23	38	56	72	91.

low ↑ mid. high

$\frac{5+(9-5)}{2}$

$5+(9-5)/2$

$5+\frac{4}{2}$

7.