1 C LANGUAGE

1) C-PROGRAMMING

1.1) Basics of c-programming

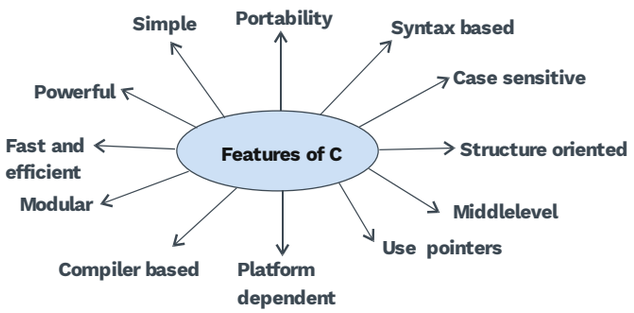


Fig. 1.1 Features of C

Introduction:

- C is a programming language designed and written by Dennis Ritchie.
- It was designed and developed at AT & T Bell Laboratories, USA, in 1972.
- In the late 70's, C began to replace more familiar languages such as ALGOL, PL/I etc. It gradually gained popularity.
- Major portions of operating systems such as Windows, Linux are written in C mainly due to its speed of execution.
- Device drivers and embedded systems are majority written in C.
- Like any other language, C-programming language is composed of alphabets, digits and special symbols which are used to form constants, variables, keywords to write instructions further. These combinations of instructions form a C-program.

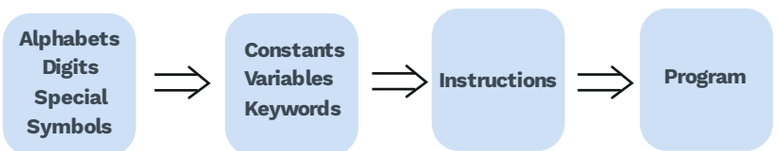


Fig. 1.2

C-character set:

- A character set denotes any alphabets, digits or special symbol used to represent information.
- The following table shows the C-character set:

Alphabets	A – Z, a – z
Digits	0 – 9
Special symbol	~ ' ! @ # % ^ & * () _ - + = | \ { } [ ] : ; " ' < > , . ? /

Token:
Smallest individual unit (Keywords, identifiers, constants, strings, special symbols, operators).

Note

- These are predefined words, each having a specific meaning.
- They are also known as reserved words.
- These words cannot be used as constansts, variables or other identifier names.
- There are total 32 keywords in C.

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

Table 1.1 Keywords in C

Identifiers:

- It is a name used to identify a variable, constant or a function.
- An identifier is a combination of letters, numbers and special symbol (only _ underscore)

Example:

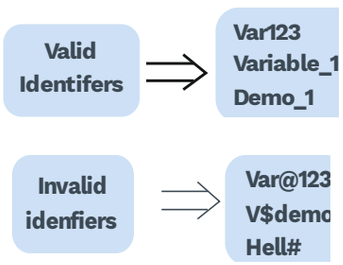


Fig. 1.3

- C does not allow punctuation characters and special symbols such as ?,-, @, #, $, % within identifiers.

Note

C is a case-sensitive language i.e. the identifiers 'Demo' and 'demo' both are two different identifiers in C language.

Whitespace:

- Whitespace in C programming language are blank lines, tabs or new lines, and blank spaces.
- These whitespace helps the compiler in understanding, and identifying when one element in the statement ends, and next element starts.

Example: int age;
A single whitespace or blankspace between int, and age enables the compiler to identify int datatypes, and age as the variable of int datatype.
whereas: int age = age1 + age2;
the space between age, = age1, + and age2 is not necessary, but for readability purposes, it is always preferred to add them.

Comments:

- They are ignored by the compiler.
- Comments are like helping texts in a C-program.
- Mostly added to increase the readability of a program.

Single line comments: //
Multi-line comments: /* _____
_____
_____ */

Instruction:

- It is a combination of keyword, variable, and constants.

Example: int a = 10;

Program

- It is a collection of instructions.

Example:

```
#include <stdio.h>          → header file
void main()                 → function
{  int a = 1; int b = 2;    → variable
   int c = a + b;           → operation
   printf("sum%d", c);
}                           → print statement
```

C-program structure:

- Preprocessor commands
- Functions
- Variables
- Statements and expressions
- Comments

Semicolon:

- A semicolon ";" is a statement terminator in C.
- Every statement must be ended with a semicolon.
- It indicates the end of a logical entry.

Constants and variables:

- Alphabets, numbers and special symbols, when properly combined form constants & variables.
- **Constant** is an entity that does not change during the execution of the program.
- **Variable** is an entity that may change during the execution of the program.

Rules to name variables and constants:

1) The variables or constant name is a combination of 1 to 31 characters, i.e. length can be maximum 30.
2) These characters can be alphabets, digits, underscores.
3) The first character in the variable or constant name must be an alphabet or underscore.
4) No comma, whitespaces are allowed within the variable or constant name.
5) No special symbol other than underscore and no keyword can be used as constant or variable name.

Example:

Valid	Invalid
ab4cd	4abcd
Var_Name	Var@Name
principal_Value1	principal#Val1

Table 1.2

Note

- Variables are the name given to a storage location which can be manipulated by the program written.
- **ivalue**
  An expression which is an Ivalue/ivalue may appear on L.H.S. or R.H.S. of the assignment.
- **rvalue**
  An expression which is a Rvalue/rvalue may appear only on the R.H.S. of the assignment.

Constants:

- Value that cannot be changed during the execution of the program

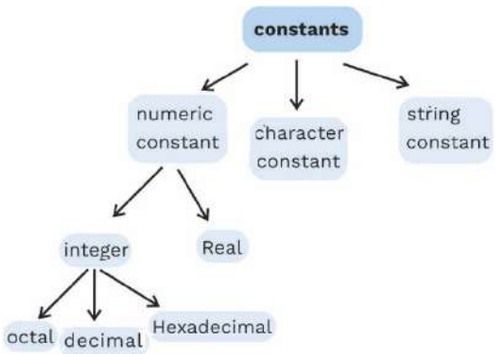


Fig. 1.4 Types of Constants

1) Numeric constant:

- numeric digits (may or may not have decimal point).
- should have at least one digit, no commas or space, either positive or negative sign.
- by default sign is positive.

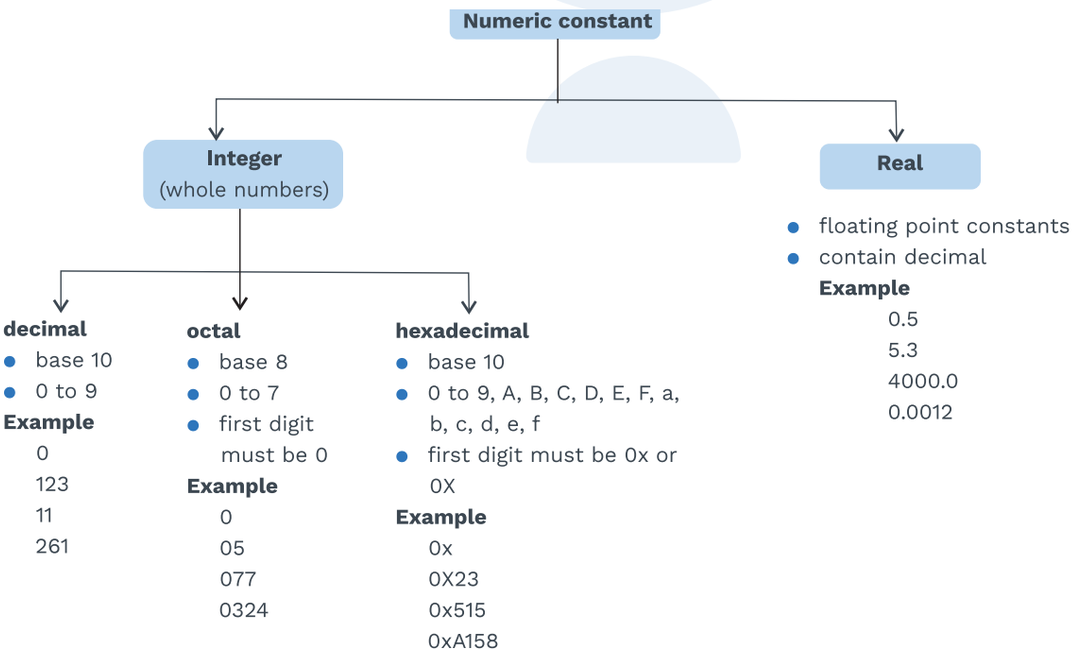


Fig. 1.5 Types of Numeric Constants

2) Character constants:

- Single character enclosed within single quotes.
- A character constant is a single alphabet, a single digit or a single special symbol enclosed within single quotes.

Example: 'a', 'B', '8', '='

Example:

Valid	Invalid
'9'	'four'
'D'	"d"
's'	y
' '	''
'#'	

Table 1.3

Note:

Character	ASCII value
A-Z	65-90
a-z	97-122
0-9	48-57
;	59

Table 1.4

3) String constants:

- It has zero, one or more than one character.
- enclosed within double quotes ""
- at the end of string \0 is automatically placed by compiler.
- \0 refers to as NULL string.

Example:

"Hello"
"8"
"593"
" "
"A"

Note

"A" represents two character: A, \0
'A' represents one character constant with ASCII value 65.

4) Symbolic constants:

- using the keyword: define
- when one constant is to be used several times.
- A symbolic constant is a tag used to replace a number, e.g. pi can be used to replace 3.14.

- These sequences of characters may be numeric constant, character constant or string constant.
- Generally defined at the beginning of the program with header files.

Syntax: #define name value

Example: Defining a constant pi whose value is 3.14159625
#define pi 3.14159625

Some more examples:
#define max 100
#define CH 'a'
#define Name "Somesh"

Note

The names are replaced by their respective values at the time of compilation in the program.

1.2 TYPES OF VARIABLES

Datatypes in C:

- An attribute given to every data used in the program is known as data type for the data. A variable is declared with a data type so that it holds a value of that type.
- The type of variable specifies how much space it requires in storage and how it is stored in memory.

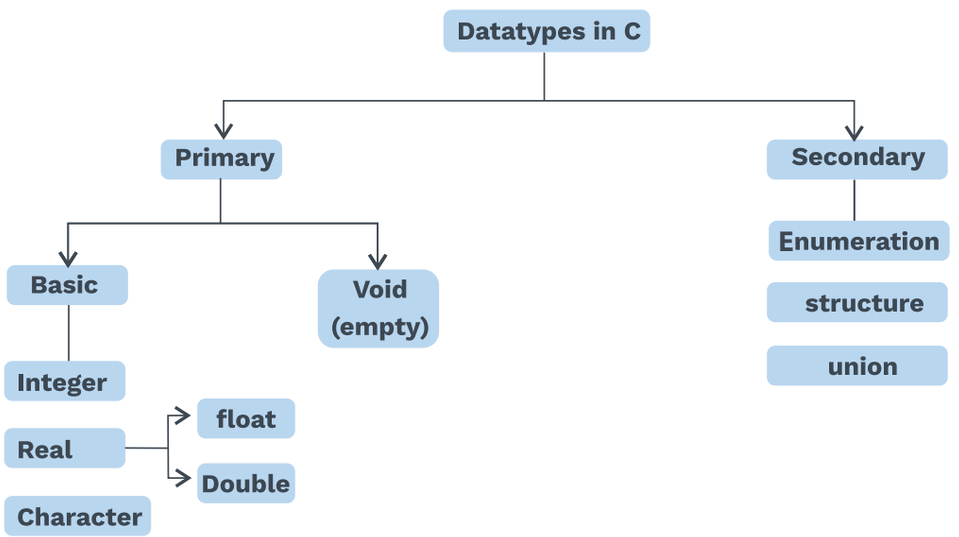


Fig. 1.6 Data-Types in C

1) Primary datatypes:

Integer	Real	Character
Signed • short int • int • long int	float double long double	signed char unsigned char
Unsigned • unsigned short int • unsigned int • unsigned long int		

Table 1.5

Note

- By default, datatypes are signed, we need to specify unsigned to use unsigned datatypes.
- The size of datatypes depends on the machine (compiler and architecture) whether they are 16 bit, 32 bit etc.
- printf("%u", sizeof(int)); //This statement prints the size of datatypes used by the system.

S.No.	Datatype	Size (n)	Range
1)	char	8 bit	– 128 to + 127
2)	unsigned char	8 bit	0 to 255
3)	short int	16 bit	– 32768 to +32767
4)	int	16 bit	–32768 to +32767
5)	unsigned int	16 bit	0 to 65535
6)	long int	32 bit	– (2^{31}) to + $(2^{31} – 1)$
7)	unsigned long int	32 bit	0 to +$(2^{32} – 1)$
8)	float	32 bit	– 3.4e38 to +3.4e38
9)	double	64 bit	– 1.7e308 to +1.7e308
10)	long double	80 bit	– 1.7e4932 to +1.7e4932
11)	unsigned short int	16 bit	0 to 65535

Table 1.6

Note

The sizes and ranges of int, short and long are compiler dependent. Sizes in this figure are for 16-bit compiler.

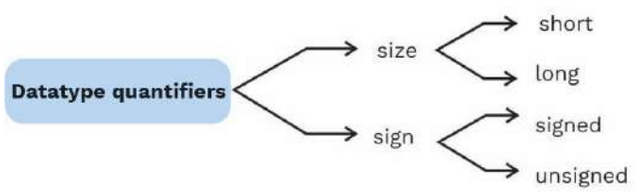


Fig. 1.7

The variable of integer data type in C Programming can be signed or unsigned number. Three different declarations of type integer are short, int, and long. All of them in size, e.g. "short" has a lower range than "int" and "long" has a larger range than "int". Though the actual size in bytes depends on the architecture on which program is being written.

Note

Range for n-bit data:
signed : -2^{n-1} to $2^{n-1} - 1$
unsigned : 0 to $2^{n} - 1$

2) Secondary datatype:

- These data types are divided or defined using primitive datatypes

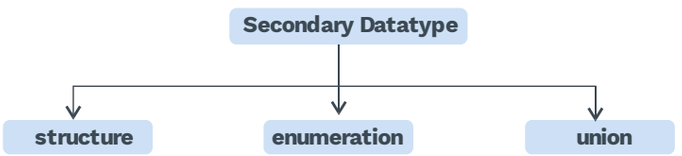


Fig. 1.8 Types of Secondary Datatypes

1.2.2 Scope and lifetime variables:

Storage classes:

- In addition to datatypes, each variable has one or more attribute known as the storage class.

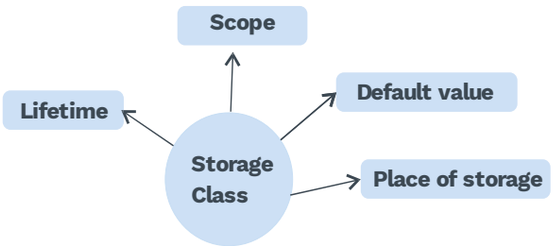


Fig. 1.9 Features of Storage Classes

- The use of storage class makes the programs more efficient and swift.
- The syntax of declaring the storage class of a variable is:

Syntax: Storage_class_name datatype variable_name;

Example: Storage_class_name int a;

- There are four types of storage classes, namely automatic, external, static and register.

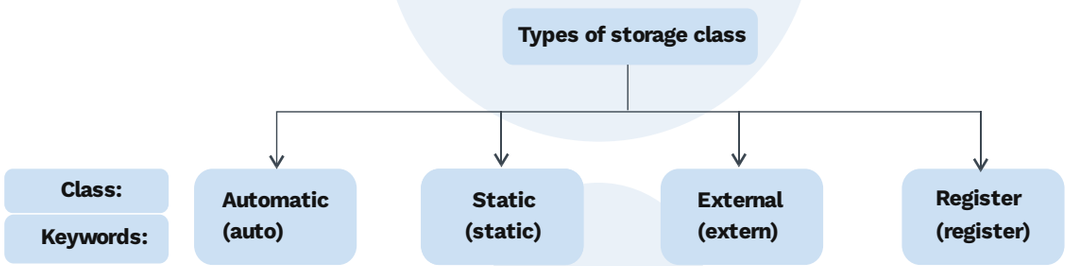


Fig. 1.10

A storage class decides about the following aspects of a variable:

1) **Lifetime:** It is the time between creation & destruction of a variable.
2) **Scope:** The progress range/location where the variable is available for use.
3) **Default Value:** The default value which is taken by uninitialized variable.
4) **Place of storage:** The place in memory which is allocated to the variable.

Note

- When a storage class is not specified, then the compiler assumes a default storage class based on the place of declaration.
- Static scoping is sometimes referred to as lexical scoping.

Types of storage class:

Storage Class Name ⟶	Automatic	External	Static (Global/Local)	Register
Keyword	auto	extern	static	register
Lifetime	function block	whole program	whole program	function block
Scope	Local	Global	Global/local	Local
Initial value	Garbage	Zero	Zero	Garbage
Storage	Stack	Data segment	Data segment	CPU register
Declaration	Inside function block	Outside/Inside function block	Outside/Inside function block	Inside function block

Table 1.7 All about Storage Classes

- To make global static variables, declare the static variable outside of all functions.
- Static variables can only be initialized by constants or constant expressions.
- An static variable is initialized once, and it retains its value during the recursive function calls.

Example 1:

Consider the given C-program statements:

```
int x = 12
static int y = 15;
func()
{    static int x;
     x = x + 2;
     printf("inside func(): x = %d, y = %d\n", x, y);
}
main()
{    int x = 13;
     func();
     func();
     printf("inside main ():x = %d, y = %d\n", x, y);
}
```

Output: Inside func():x = 2, y = 15
Inside func():x = 4, y = 15
Inside main():x = 13, y = 15

In func(), x is initialized to 0 and x = 0 is used, since it is a static variable, hence its value is retained between function calls. Since y has been initialized outside main and func(), it is accessible to both the functions.

Example 2:
Consider the given C-program statements:

```
func1()
{    extern int x;
     x++;
     printf("func1:%d\n", x);
}
     int x = 189;
     func2()
{    x++;
     printf("func2:%d\n", x);
}
main()
{    func1();
     func2();
}
```

Output: func1:190
func2:191

Previous Years' Questions

The value of j at the end of the execution of the following C-program is _____

```
int incr(int i) {
static int count = 0;
count = count + i;
return count;
}
main(){
int i, j;
for (i = 0;i = 4; i++)
j = incr(i);
}
```

a) 10 **b)** 4 **c)** 6 **d)** 7

Sol: Option a) **(GATE-2000)**

Previous Years' Question

Consider the following C-function:

```
int f(int n)
    {  static int i = 1;
       if(n >= 5) return n;
       n = n + i;
       i++;
       return f(n);
     }
```

The value returned by f(1) is:

a) 5 **b)** 6 **c)** 7 **d)** 8

Sol: Option c) **(GATE-2004)**

Previous Years' Question

Early binding refers to a binding performed at compile time, and late binding refers to a binding performed at execution time. Consider the following statements:

i) Static scope facilitates **W1** bindings.
ii) Dynamic scope requires **W2** bindings.
iii) Early bindings **W3** execution efficiency.
iv) Late bindings **W4** execution efficiency.

The right choices of W1, W2, W3 and W4 (in that order) are:

a) Early, late, decrease, increase **b)** Late, early, increase, decrease
c) Late, early, decrease, increase **d)** Early, late, increase, decrease

Sol: Option d) **(GATE-2007)**

Previous Years' Question

Consider the program given below, in a block-structured pseudo-language with lexical scoping, and nesting of procedures permitted.

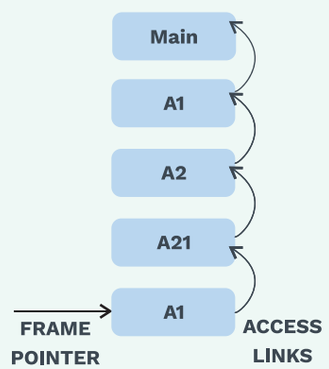
```
Program main;
    Var ...
        Procedure A1;
            Var ……..
            Call A2;
            End A1
        Procedure A2;
            Var ...
            Procedure A21;
                Var ...
                Call A1;
                End A21
        Call A21;
    End A21
        Call A1 ;
    End main.
```

Consider the calling chain:

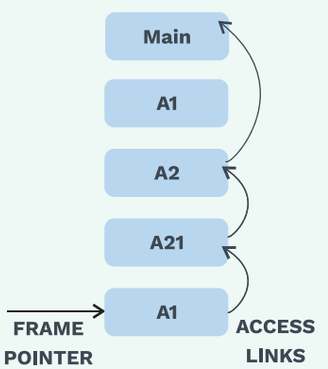
Main → A1 → A2 → A21 → A1

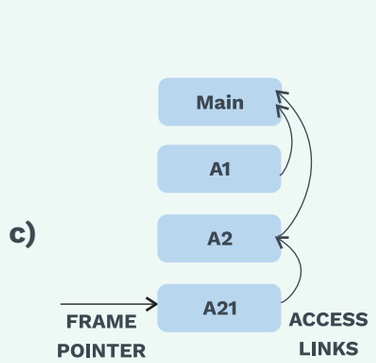
The correct set of activation records along with their access links is given by:

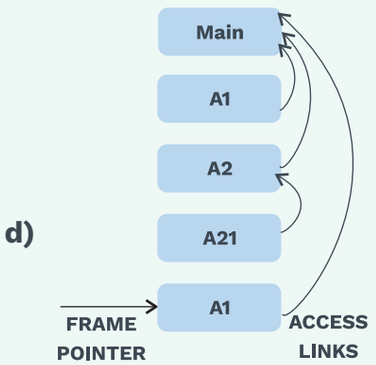
a)



b)







Sol: Option d) **(GATE-2012)**

1.3 OPERATORS IN C

- An operator specifies an operation to be performed that yields a value.
- An operand is a data item on which an operator acts upon.

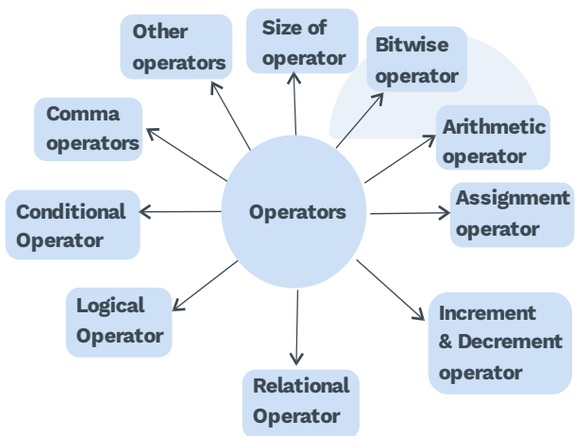


Fig. 1.11 Types of Operators in C

- Operators can be of two types primarily:
  1) unary operator **2)** binary operators

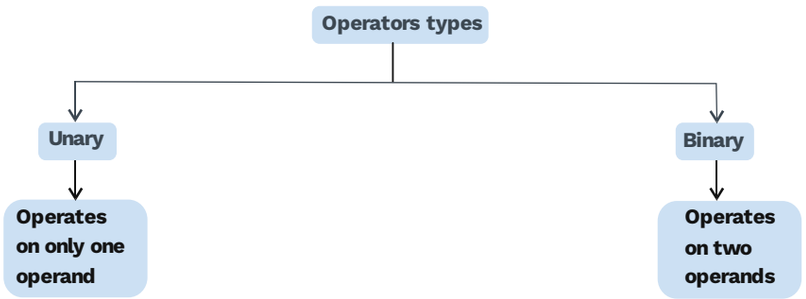


Fig. 1.12

Bitwise operator:

- Manipulation at bit level can be performed using bitwise operators.
- These operators perform operation on individual bits.

Operator	Meaning
&	bitwise AND
|	bitwise inclusive OR
^	bitwise XOR
~	One's compliment (unary)
<<	Bitwise left shift
>>	Bitwise right shift

Table 1.8 Bitwise Operators

Note

- Bitwise operators operates on integral operands only.
- All bitwise operators are binary except compliment operator, which is unary.
- All bitwise operators except compliment can be obtained with the assignment operators.

&=,|=, <<=, >>=, ^ =
Compound Assignment Operators

Bitwise AND (&):

- Binary operator represented as &

Bit of operand 1	Bit of operand 2	Resulting bit
0	0	0
0	1	0
1	0	0
1	1	1

Table 1.9

Syntax: operand 1 & operand 2

Bitwise inclusive OR (|):

- Binary operator represented by |.

Bit of operand 1	Bit of operand 2	Resulting bit
0	0	0
0	1	1
1	0	1
1	1	1

Table 1.10

Syntax: operand 1 | operand 2

Bitwise XOR (^):

- produces output 1 for only those input combinations that have odd number of 1's.
- Binary operator represented by ^.

Bit of operand 1	Bit of operand 2	Resulting bit
0	0	0
0	1	1
1	0	1
1	1	0

Table 1.11

Syntax: operand 1 ^ operand 2

Compliment (~):

- One's compliment operator represented by ~.
- unary operator

Bit of operand	Resulting bit
0	1
1	0

Table 1.12

Syntax: ~ operand.

Bitwise left shift (<<):

- Binary operator represented by <<.

Syntax:

Operand 1 ≪ Operand 2

↓ ↓

Operand whose bits are to be shifted left — No. of bits to be shifted

- Shifting bits results in equal no. of bits being vacated on the other side, these vacated spaces are filled with 0 bits.

Example:

Let x = 0001 0011 0000 0100, then x<<4 means left shift x by 4 bits, then:

0001 0011 0000 0100 0001

Lost bits **Filled bits**

∴ **After left shifting:** 0011 0000 0100 0000

Bitwise right shift (>>):

- Binary operator represented by >>.
- Similar to left shift except it shifts bit in the opposite direction as of left shift i.e. shift bit to the right side.
- Similar to left shift, here the bits are shifted to right & bits are vacated in the left and right shifting on unsigned quantity always fill with zero in the vacated positions. In right shifting on signed quantity, then will fill with sign bits ("arithmetic shift") on the vacated positions.

Example:

Let unsigned int x= 0001 0011 0000 0100, then x>>4 means right shift x by 4 bits, then:

0000 0001 0011 0000 0100

Filled bits **Lost bits**

After right shifting: 0000 0001 0011 0000

Note

In right shift if the first operand is signed integer, then the result is compiler dependent.

Arithmetic operator:

- On the basis of no. of operands:

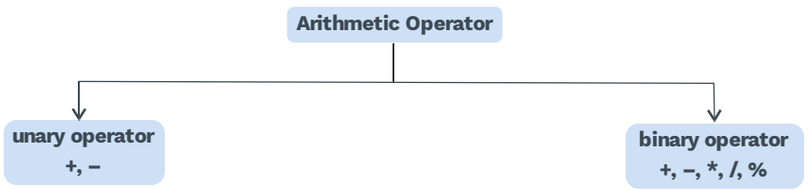


Fig. 1.13

- On the basis of value of operands:

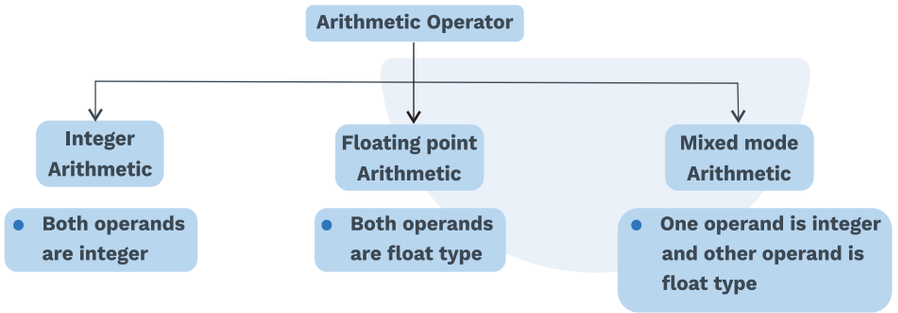


Fig. 1.14

Binary arithmetic operator:

Operator	Meaning
+	Addition
–	Subtraction
*	Multiplication
/	Division
%	Modulus (gives remainder in integer division)

Table 1.13

Note

% (Modulus operator cannot be applied to floating point operands). There are no exponent operator in C, whereas the library function pow () is used as exponentiation operation.

Relational operator:

- These operators are used to compare values of two expressions depending on their relations.
- Relational expression is an expression with relational operators.

Operator	Meaning
<	less than
<=	less than or equal to
==	equal to
!=	not equal to
>	greater than
>=	greater than or equal to

Table 1.14

Note

Value of relational expression = 1, if relation is True.
Value of relational expression = 0; if relation is False.

'=' and "= =" have entirely different meaning.

↓ → checks

assignment equality

operator

Logical operators:

- They are also known as Boolean operators.
- Used for combining expressions.
- They return 0 for false and 1 for true.

Operator	Meaning
&&	AND
||	OR
!	NOT

Table 1.15

Note

Logical NOT is a unary operator whereas && and || are binary operators.
In C, any non-zero value is regarded as true and 0 is regarded as false.

Non – zero value → True
Zero value → False

AND operator (&&):

- Binary operator represented as &&.

Condition 1	Condition 2	Result
False	False	False
False	True	False
True	False	False
True	True	True

Table 1.16 Truth table for AND

Syntax: (condition 1) && (condition 2)

OR operator (||):

- Binary operator represented as ||.

Condition 1	Condition 2	Result
False	False	False
False	True	True
True	False	True
True	True	True

Table 1.17 Truth table for OR

Syntax: (condition 1) || (condition 2)

NOT operator (!):

- unary operator represented by !.
- negates the value of the condition.

Condition	Result
False	True
True	False

Table 1.18 Truth table for NOT

Syntax: ! (Condition)

Assignment operator:

- used to assign values to variables, denoted by symbol '='.

Syntax:

Operand 1 = **Operand 2**

↓ Should be a variable

↓ Can be a variable, expression or a constant

Compound assignment operator:

- Combination of arithmetic operator with assignment operator.

Compound assignment operator	Alternate
x + = 5	x = x + 5
x – = 5	x = x – 5
x/=5	x = x/5
x% = 5	x = x%5
x*=5	x = x*5

Table 1.19

Increment and decrement operator:

- Unary operator
- Increment operator represented by ++.
- Decrement operator represented by ––.

Operator	Meaning	Equivalent
++x	increments the value of variable by 1	x = x + 1
––x	decrement the value by variable by 1	x = x – 1

Table 1.20

- ++ increment by 1
- -- decrement by 1.
- The increment and decrement operator can be either: (1) prefix; (2) postfix;

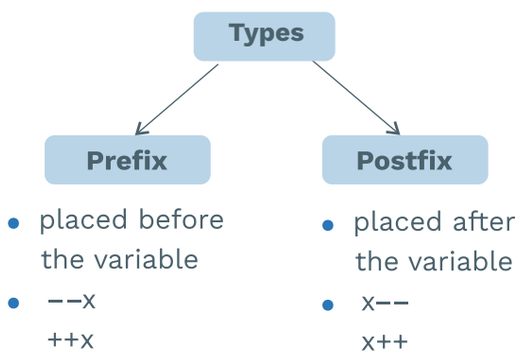


Fig. 1.15

Prefix increment/decrement:

- The value of the variable is incremented or decremented, then the value is used in the operation.

Syntax: ++x or Operator variable;
––x

Example:
Let x =5, then y=++x means increment the value of x by 1 and then use the value to be assigned to y. ∴ y = 6

Similarly, y=––x means decrement the value of y by 1 and then use the value to be assigned to y. ∴ y = 1

Postfix increment/decrement:

- First the value of variable is used, then it is incremented or decremented.

Syntax: x++ or Variable operator;
x––

Example:
Let x =5, then y = x++ means assign y = 5, then increment the value of x by 1 i.e. after execution y = 5 & x = 6.
Similarly, y = x–– means assign y = 5, then decrement the value of x by 1 i.e. after execution y = 5 & x = 4.

Conditional operator:

- It is a ternary operator which requires three expressions as operands.
- Represented by ? and : space
- **Syntax:** Text expression ? expression 1: expression 2

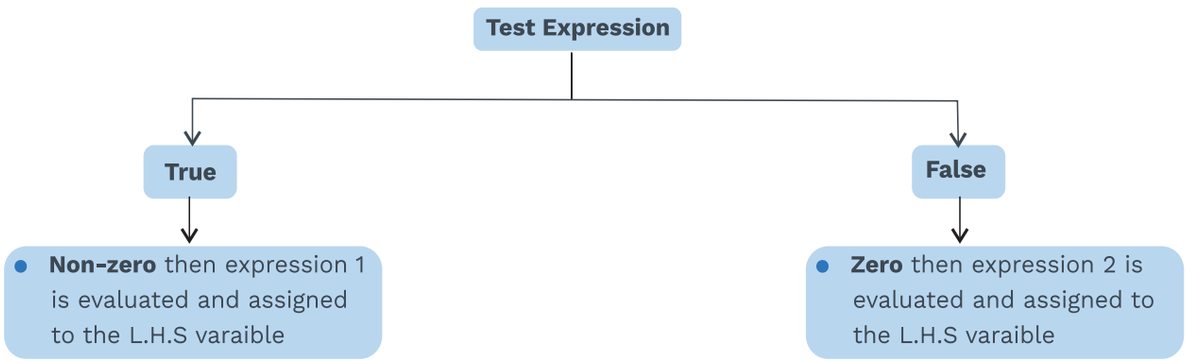


Fig. 1.16

Example:
max = a > b ? a: b

This translates to that if (a > b) ⟶ True, then max = a
⟶ False, then max = b

- printf statements can also be used.

Example: a > b ? printf (“a is larger”): printf (“b is larger”);

Rack Your Brain

Which of the following is not a unary operator?
1) ++ **2)** – **3)** sizeof **4)** ?:

Comma operator:
- Represented by ‘,’
- It is used to allow different expressions to appear in places where exactly one expression would be used.
- Also commonly used as a separator.

Example:

Without comma operator	With comma operator
a = 8	a = 8, b = 7, c = a + b;
b = 7;	
c = a + b;	

Table 1.21

Example:
Sum = (a = 10, b = 7, c = 3, a + b + c);
Here sum would be a + b + c i.e. sum = 20.

Sizeof operator:
- unary operator
- returns the size of operand in bytes.
- The parameter passed can be a variable, constant or any datatype (int, float, character).

Syntax: sizeof(parameter);
↓
Variable\ constant\ Datatype

Example:
sizeof(int); would return the bytes occupied by integer datatype.

Note

Another operator called the typecast operator is used for carrying out type conversion.

Rack Your Brain

Consider the given C-program:

```
#include <stdio.h>
int main()
{printf("%u%u%u%u",   sizeof(schar),   sizeof(int),   sizeof(float)
sizeof(double));
return 0;
}
```

1) 1 4 4 8 **2)** 1 2 4 8 **3)** 1 4 8 16 **4)** Machine dependent

Previous Years' Question

Consider the following C program:

```
#include<stdio.h>
int main()
{
        int m = 10;
        int n,n1;
        n =++m;
        n1 = m++;
        n--;
        --n1 ;
        n-=n1;
        printf ("%d", n);
        return 0;
}
```

The output of the program is _____.

Sol: 0) **(GATE-2017 (Set-2))**

Previous Years' Question

Consider the following C-program:

```
int a, b, c = 0;
Void prtFun(void);
main()
{    static int a = 1; /*Line 1*/
     prtFun();
     a+=1;
     prtFun();
     printf ("\n%d %d", a, b);}
void prtFun(void)
{    static int a = 2; /*Line 2*/
     int b = 1;
     a+ = ++b;
     printf ("\n%d%d",a,b);
}
```

What output will be generated by the given code segment?

a)
3 1
4 1
4 2

b)
4 2
6 1
6 1

c)
4 2
6 2
2 0

d)
3 1
5 2
5 2

Sol: c)

What output will be generated by the given code segment if:
Line 1 is replaced by auto int a=1;
Line 2 is replaced by register int a=2;

a)
3 1
4 1
4 2

b)
4 2
6 1
6 1

c)
4 2
6 2
2 0

d)
4 2
4 2
2 0

Sol: d)

(GATE-2012 (Common Data Question))

1.4 TYPE CONVERSION

- It includes converting data type of one operand into data type of another operand to perform operations.

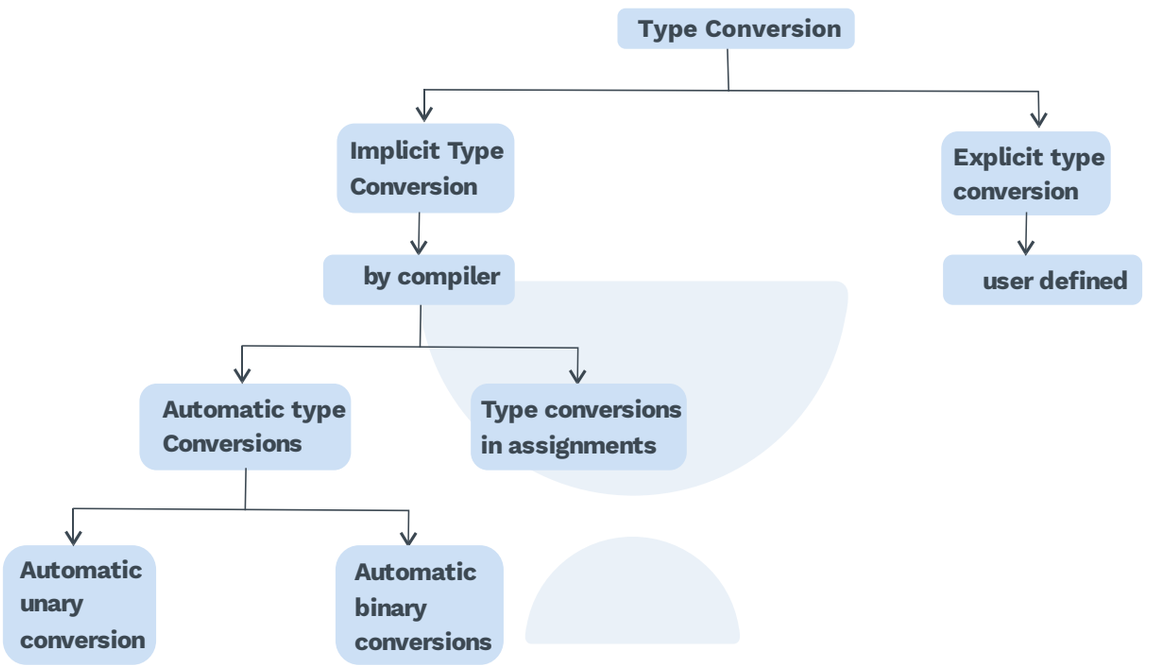


Fig. 1.17

Implicit type conversion:

- Done by the C compiler based on some predefined C-language rules.

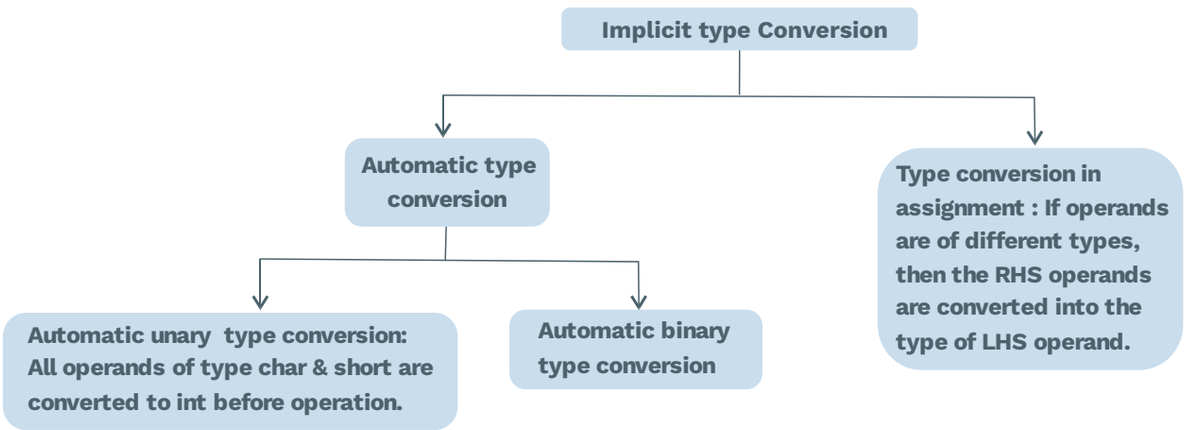


Fig. 1.18

Note

- For automatic unary type conversion, some compilers convert all float operands to double before operation.
- There are some rules and rank hierarchy followed for automatic binary type conversion.
- In type conversion during assignment, the RHS operand are demoted or promoted based on the rank of LHS operand.

Rules for automatic binary type conversion:

1) With a binary operator, if both the operands have dissimilar data types then the operand with lower rank gets converted into the data type of higher rank operand. It is known as data type promotion.

2) **In case of unsigned datatypes:**

i) One operand is unsigned long int, others are converted to unsigned long int as well as the result is stored as unsigned long int.

ii) One operand is unsigned int and other is long int, then:

Case I If long int can represent all values of unsigned int, then unsigned is converted to long int and the result is also stored in long int etc.

Case II Both are converted to unsigned long int along with the result.

iii) One operand is unsigned int, then others along with the result are converted and stored as unsigned int.

Consequences of these promotions and demotions

1) Few higher order bits get dropped when higher order rank data type is converted to a lower order rank data type. E.g. when int is converted to short.

2) During the conversion of the float data type into int, the fractional part gets removed.

3) Digits are rounded off while conversion of double type to float type.

4) The sign may be dropped in case of conversion of signed type to unsigned types.

5) There is no increased accuracy or precision when an int is converted to float or float to double.

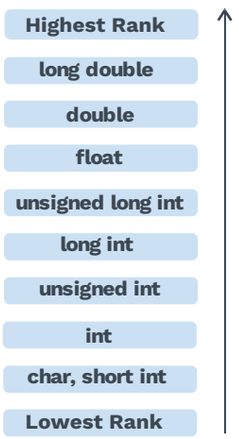


Fig. 1.19 Rank of Datatypes in C

Explicit type conversion (Type casting):

Some cases such as:

```
float z;
int x = 20, y = 3;        ...(i)
z = x / y;
```

The value of z would be 6.0 and not 6.666666

- These cases require the implementation of explicit type conversion.
- This allows user to specify our own/user defined conversion.
- Also known as Type casting or Coercions.
- Type casting is implemented using the cast operator.
- The cast operator is a unary operator which converts an expression to a particular datatype temporarily.

Syntax : (Datatype) expression;
Cast Operator

- Using the cast operator (float) in (i).

z = (float) x/y;

then the value of z would be 6.666667

Fig. 1.20

Precedence and associativity of operators:

- For an expression having more than one operator, there exists certain precedence and associativity rules for its evaluation.

Example:

Given an expression: 2 + 3 * 5

It contain 2 operations: + and *, if + performed before *, then result would be 25 and if * performed before +, then result would be 17.

∴ To remove this ambiguity of which operation to perform first the C-languages specifies the order of precedence & associativity of operators.

- **Operator precedence:** Determines which operator is performed first in an expression.
- **Operator associativity:** It is used when two operators of same precedence appear in an expression.

Associativity can be: Left to right

or

Right to left

Operator	Description	Precedence	Associativity
( )	Parentheses or function call	1	Left to right
[ ]	Brackets or array subscript		
●	Dot or member selection operator		
→	Arrow operator		
+	Unary plus	2	Right to left
–	Unary minus		
++	Increment		
––	Decrement		
!	Logical NOT		
~	One's complement		
*	Indirection or dereference operator		
&	Address		
(Datatype)	Type cast		
sizeof	Size in bytes		
*	Multiplication	3	Left to right
/	Division		
%	Modulus		

+	Addition	4	Left to right
–	Subtraction		
<<	Left shift	5	Left to right
>>	Right Shift		
<	Less than	6	Left to right
<=	Less than or equal to		
>	Greater than		
>=	Greater than or equal to		
==	equal to	7	Left to right
!=	not equal to		
&	Bitwise AND	8	Left to right
^	Bitwise XOR	9	Left to right
!	Bitwise OR	10	Left to right
&&	Logical AND	11	Left to right
‖	Logical OR	12	Left to right
?:	Conditional operator	13	Right to left
=	Assignment Operator	14	Right to left
*=,/=, %=			
+=, –=			
&=,^=, /=			
<<=, >>=			
,	Comma Operator	15	Left to right

Table 1.22 Operator Precedence and Associativity Table

Example: 5 + 16 / 2 * 4

- Since / and * have higher precedence than +

∴ they are evaluated first.

- / and * have same precedence, so which would be evaluated first amongst the two is determined by the associativity rules. Since, /and * are left to right associative.

∴ / is performed before *.

The given expression can be considered as:

5 + (16/2) * 4

Solving: 5 + (8 * 4)

⇒ 5 + 32

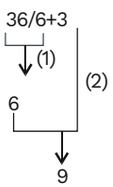
⇒ 37

Role of parentheses:

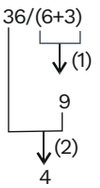
- Parenthesis are used to change the order of precedence of any operation.
- All the operations enclosed in parenthesis are performed first.

Example:

I. Without parenthesis



II. With parenthesis



- Sometimes to increase the readability of the code and expression parenthesis are used.

Example: x = a!=b & c * d >= m%n
Better: x = (a!= b) & ((c*d)>=(m%n))

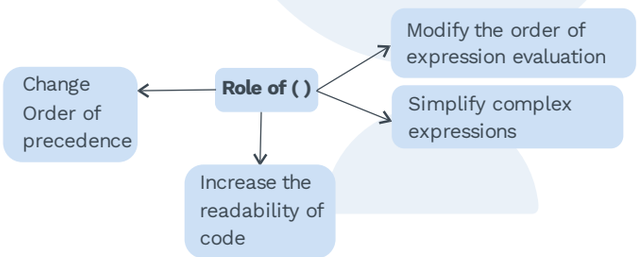


Fig. 1.21

1.5 FLOW CONTROL IN C

- Control statements enable us to specify the order in which the various instructions in the program are to be executed.
- It determines the flow of control in C.

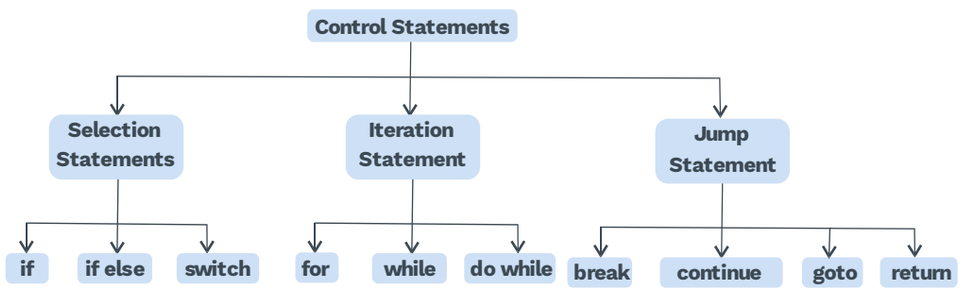


Fig. 1.22 Types of Control Statements

- Compound statements or a block are a group of statements which are enclosed within a pair of curly braces { }.

A compound statement is syntactically equivalent to a single statement.

If else statements:

- Bi-directional conditional control statement.
- The statement tests one or more conditions and executes the block based on the outcome of the test.
- Any non-zero value is regarded as true, whereas 0 is regarded as false.

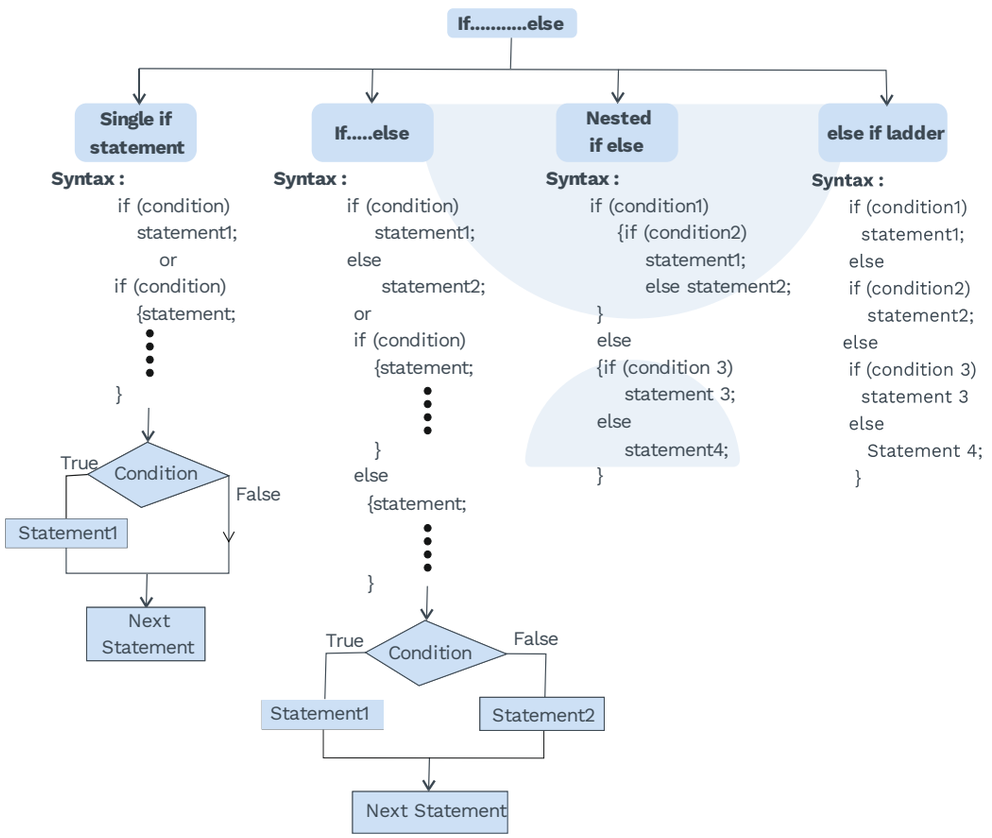


Fig. 1.23 Types of If-else Statements

Loops:

- Used when one wants to execute a part of a program or a block of statements several times.
- With the help of loops, one can execute a part of program repeatedly till some condition is true.

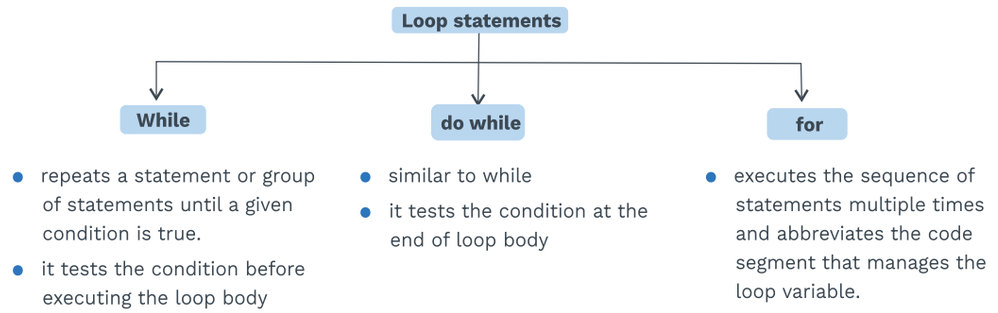


Fig. 1.24 Types of Loop Statements

While loop:

- First the condition is evaluated, if it is true, then the body of loop is executed.

Syntax: While (condition)
Statement;

OR

While (condition)
{ Statement;
Statement;
⋮
⋮
}

Note: While loops are used when the number of iterations are not known in advance.

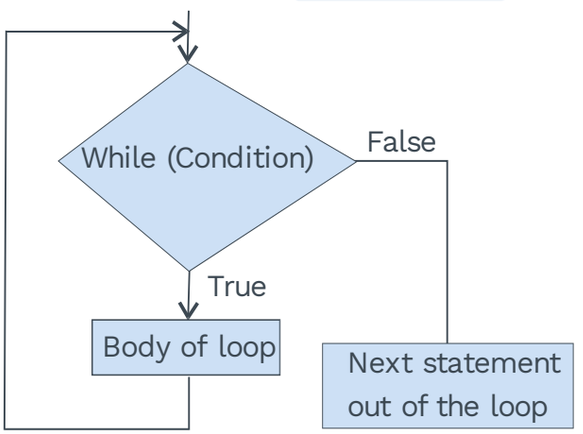


Fig. 1.25 Flow Chart for While Loop

- Each execution of loop body is called an iteration.

Example: #include <stdio.h>
main()

```
{ int i = 1;
        while(i <=5)
        { printf("%d\t", i);
        i++;
        }
}
```

Output: 1 2 3 4 5

This C-program prints the numbers from 1 to 5. The variable i is initialized to 1 and then the condition is checked whether (i < = 5) if true, then the printf statement is executed. The value of i is incremented by 1 each time the loop is executed.

Previous Years' Question

Consider the following C-program:

```
#include <stdio.h>
int main()
{float sum=0.0, j=1.0, i=2.0;
While(i/j>0.0625)
{
j=j+j;
sum=sum+i/j;
printf("%f\n", sum);
}
 return 0;
}
```

The number of times the variable sum will be printed, when the above program is executed is _____.

Sol: 5 to 5 **(GATE-2019)**

Do-while loop:

- Similar to while loop just that in do while loop first the statements are executed, then the condition is checked.
- This results in the execution of the statements at least once even if the condition is false for the first iteration.

Syntax:

```
do
   Statement;
while (condition);
```

or

```
do
{ Statement;
      Statement;
      ⋮
} while (condition);
```

- In do while loop, a semicolon is placed after the while (condition).

Example:

```
#include <stdio.h>
main()
{ int i = 1;
     do
     { printf ("% d\t", i);
       i++ ;
     } while(i <=5);
}
```

output: 1 2 3 4 5

For a program to count the no. of digits, it is better to use do while loop over while loop. This program if written using while loop returns the number of digits as 0 for n = 0. Therefore, here using do while loop is preferred.

```
#include <stdio.h>
main()
{    int n, count = 0;
     printf("enter a no.");
     scanf("%d",&n);
     do {    n/=10;
             count ++;
         } while(n > 0);
     printf("no. of digits = %d", count);
}
```

For loop

- Combined of three expression separated by semicolons.

Syntax:

```
for(expression 1; expression 2; expression 3)
      Statement;
            Or
for(expression 1; expression 2; expression 3)
      {  Statement;
         Statement;
            ⋮
      }
```

- The body of for loop can have single statement or a block of statements.

- In for loop:

Expression 1: Initialization expression, executed only once, and is generally an assignment expression.

Expression 2: Test expression or condition, tested before each iteration, generally uses relational or logical operators.

Expression 3: Update expression or updation, executed each time for body of loop is executed.

- Firstly, the initialization expression is executed, and the loop variables are initialized, then the condition is checked.
- If the condition expression is true, then the body of the loop is executed, and the updation expression is executed.
- This continues till the condition expression is true, and once the condition expression becomes false, the loop terminates and control is transferred to the statement following the for loop.

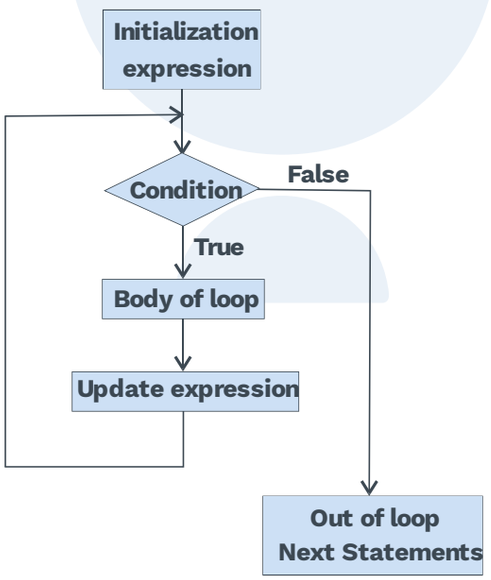


Fig. 1.26 Flow Chart of "for" Loop

- For loops are used when the number of iterations are known in advance.

Example:

```
# include <stdio.h>
main()
{ int i;
    for(int i = 1; i < = 5; i++)
    printf("%d\t",i);
}
```

Output: 1 2 3 4 5

Note

- The variable/loop variable i can be declared before or at the time of initialization in expression 1.
- All three expressions of for loop are optional. One can avail any one or all three expressions.
- The two separating semi-colons are mandatory.
- Expression 1 can be omitted if initialization is done before the loop.
- Expression 2 which is the condition if omitted is always considered true. Hence, loop will never stop executing resulting in an infinite loop.
- Expression 3 which is the updation can be omitted if it is present inside the loop body.

Some loops that are valid:

- for(; n > 0; n/=10)//initialization of n should be mentioned before loop.
- for(i = 0, j = 10; i < =j; i++, j--)//multiple expression separated by comma & semi-colon.
- for(; ;)//infinite loop.

Example:

```
#include<stdio.h>

main()
{   int n, sum = 0;
    printf("Enter the no.:");
    scanf("%d", & n);
    for(; n > 0; n/=10)
    {   int t = n% 10;
        sum + = t;
    }
printf(" sum = %d", t);

}
Example:
        #include<stdio.h>
        main()
        {   int i, j;
            for (i = 0, j = 10; i<=j; i++, j–=2)
                {printf ("i =%d\t    j = %d\n", i, j);
                }
        }

Output:                    i = 0                    j = 10
              i = 1              j = 8
              i = 2              j = 6
              i = 3              j = 4
```

Previous Years' Question

Consider the following C code. Assume that unsigned long int type length is 64 bits.

```
unsigned long int fun(unsigned long int n)
    {unsigned long int i, j=0, sum=0;
    for(i=n;i>1;i=i/2) j++;
    for(;j>1;j=j/2) sum++;
    return(sum);
    }
```

The value returned when we call fun with the input 240 is:

a) 4 **b)** 5 **c)** 6 **d)** 40

Sol: b)

Hint : when n=240, j=log(240) **(GATE-2018)**

Nesting of loops:

- Loop within a loop
- Any type of loop can be nested inside any other type of loop.

Infinite loops:

- Loops that go on executing infinitely.
- The loops that do not terminate are called infinite loops.

Example:

while(1)
{.....
.....
}
 	
Infinite while loop
	for (; ;)
	{.....
	.....
	}
	
	Infinite for loop
		do
		{.....
		.....
		} while (1);
		
		Infinite do while loop

- To come out of such loops break or goto statements are used.

1.6 BREAK, CONTINUE AND GOTO STATEMENTS

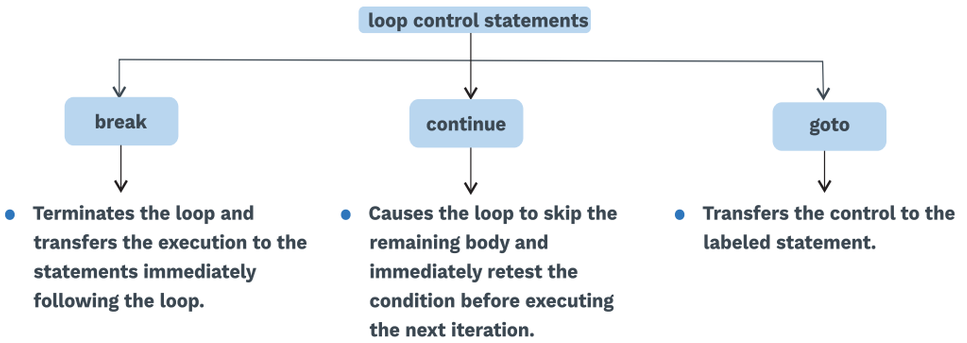


Fig. 1.27 Types of Control Statements

- These loop control statements enable the control of programs/loop to be transferred.

Break statement:

- Used when it becomes necessary to come out of the loop even before the loop condition becomes false.
- Break statement causes the immediate exit from the loop.

Syntax: break;

- Popularly used in switch statements.

While loop:

```
while (condition)
    {  Statement;
       if (condition to break){
          break;
       }
       Statement;
    }
Control
```

Note

As soon as the break statement is encountered, the control is transferred to the next statements after the loop.

Do-While Loop

```
do{
        Statement;
        if (condition to break){
        break;
        }
        Statement;
    } while (condition);
Control
```

For Loop

```
      Initialization   Condition      Updation
           ↑              ↑             ↑
for (expression 1; expression 2; expression 3)
   {  Statement;
      if (condition to break) {
      break ; }
      Statement;
   }
Control
```

Continue statement:

- Used when one wants to skip some statements and execute the next iteration.

Syntax: continue;

- Generally used with condition statements, when a condition statement is encountered and is true, then the statements after the keyword continue are skipped & the loops condition is checked for next iteration.

Note

In break, the loop terminates and control is transferred to outside. In continue the current iteration terminates and the control is transferred to the beginning of the loop.

Example:

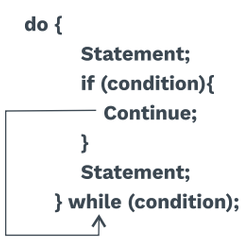
While Loop

```
while (condition) {
        Statement;
        if (condition)
        }
        Statement;
```

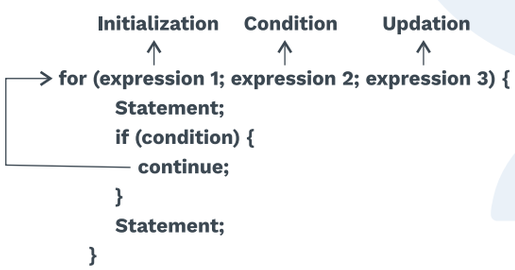
Note

As soon as the condition of if is true, the continue statement is executed. This results in control being given to the while condition and next statements are skipped.

Do-while Loop:



For Loop



Goto statement:

- Unconditional control statement
- Transfers the flow of control to another part of the program.

Syntax: goto **label;**

Statement;
Statement;
.....
.....
.....

label:
Statement;
Statement;
.....
.....

Note

The control is transferred immediately after the label.

- The label can be placed anywhere:

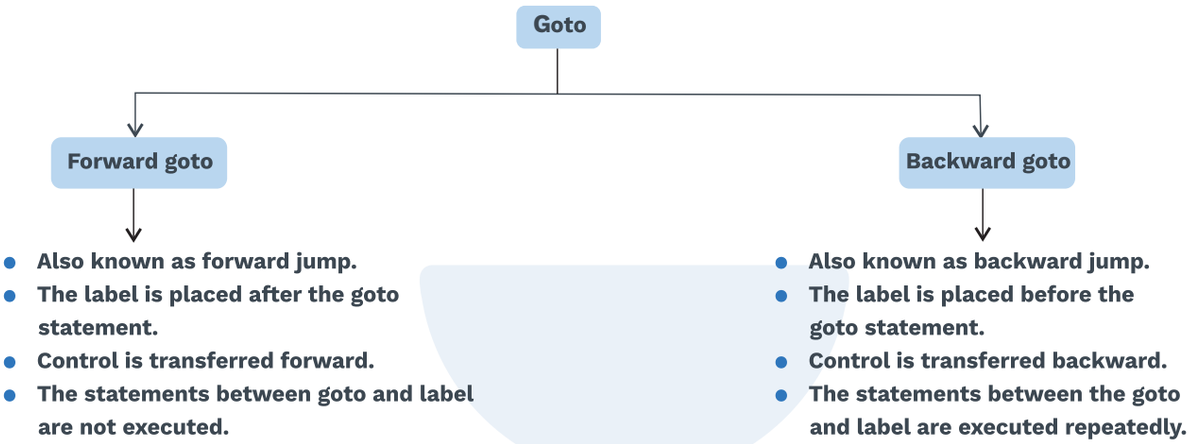


Fig. 1.28 Types of Goto Statements

Note

The control in goto can be transferred within a function.

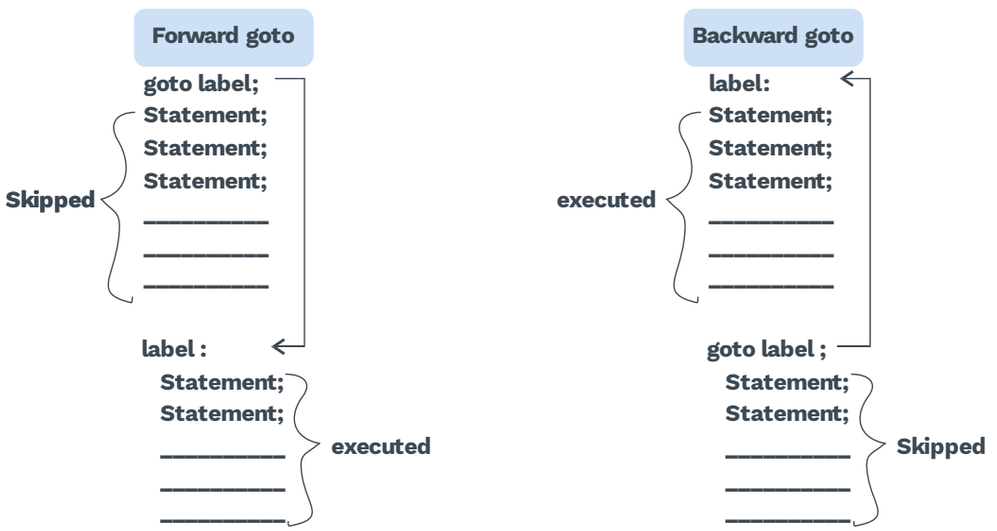


Fig. 1.29

Note

Use of goto statements is not preferred as it makes it difficult to understand where the control is being transferred.

Often leads to spaghetti code (Ambiguous & not understandable)

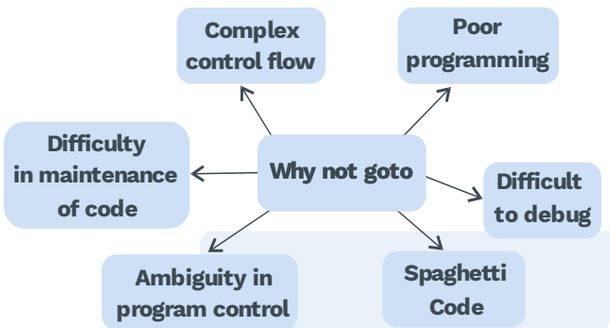


Fig. 1.30

Although there might be situation such as complex nested loops or deeply nested loops where goto or jump statements can improve the readability of code.

Example:

```
for (initialization; Condition; updation)
  {
     while (condition)
     {   for (initialization; condition; updation)
           { do
              { _____
                _____
                _____
              }while(condition);
           while (condition)
           { if  (condition)
              {goto stop;
              }
           }
        }
     }
  }
  Stop :
  _____                  * exit from the deeply
  _____                    nested loops.
  _____
```

Return statement:

- ends the execution of function and returns the control of execution to the calling function.
- does not mandatorily need any conditional statements.
- Depending on the type of function it may or may not return values.

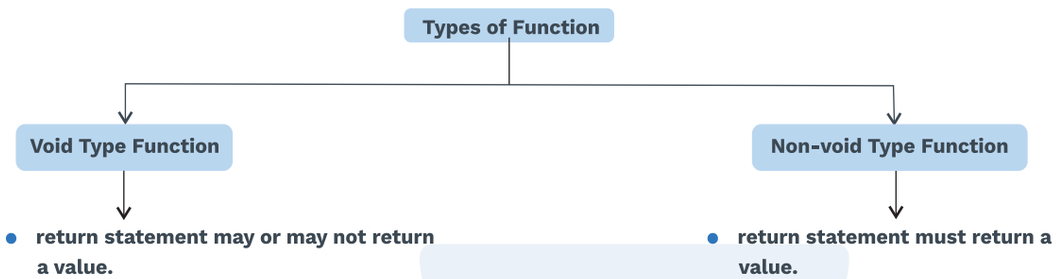


Fig. 1.31 Types of Functions

Syntax: return expression;

Example:

Void Function

```
void func()
   {
   }
```

Non-void Function

```
return_datatype func()
        { return value;
        }
```

1.7 SWITCH CASE

- Multi-directional conditional control statement.
- Popularly used for menu driven programs.
- **Uses three keywords:** switch, case, break, default.

Syntax:

```
switch(expression)
{ case constant 1: Statement;
                   break;
case constant 2: statement;
                   break;
_____
                _____
                _____
default: Statements;
}
```

- **Expression:** can be anything:
  1) An expression yielding an integer value.
  2) Value of any integer
  3) Character variable
  4) Function call returning a value.

- **Constants:**
  1) Should be integer or character type.
  2) can be constants or constant expressions.

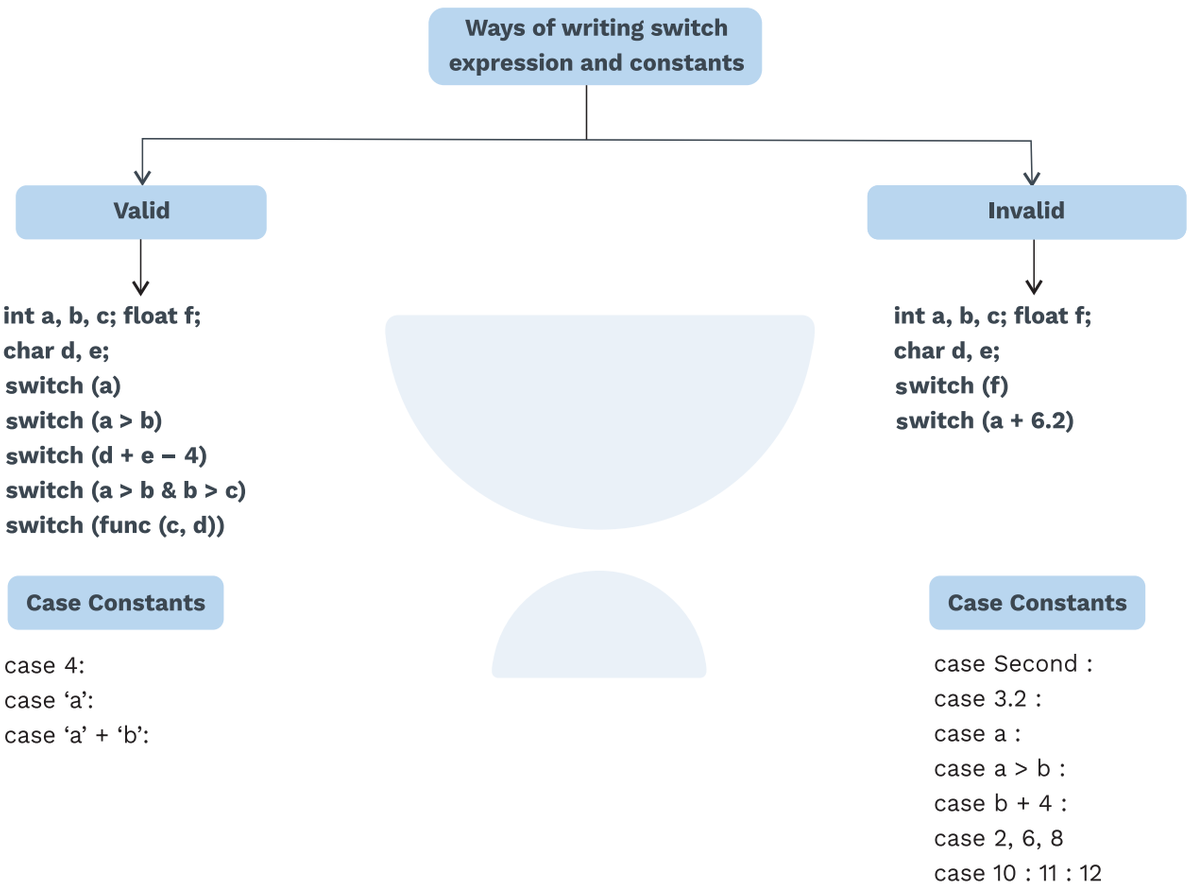


Fig. 1.32

Note

- If break statements are not used, then the control falls through the case in switch. As soon as one case matches, the cases following it are also executed in the absence of break statement.
- This is popularly known as fall-through cases.
  Default can be executed if no other case constant matches the switch expression.

Previous Years' Question

What will be the output of the following C program segment?

```
char inchar='A';
switch(inchar)
{
        case 'A': printf("choice A\n");
        case 'B':
        case 'C': printf("choice B");
        case 'D':
        case 'E':
        default: printf("No choice");
}
```

No choice

a) choice A
b) choice A
c) Choice B No choice
d) program gives no output as it is erroneous.

Sol: c) **(GATE-2012)**

1.8 FUNCTIONS AND PROGRAM STRUCTURES

1.8.1 Basics of functions:

- Self-contained sub-program with a well-defined task.
- Although function calls are overhead but are still used.

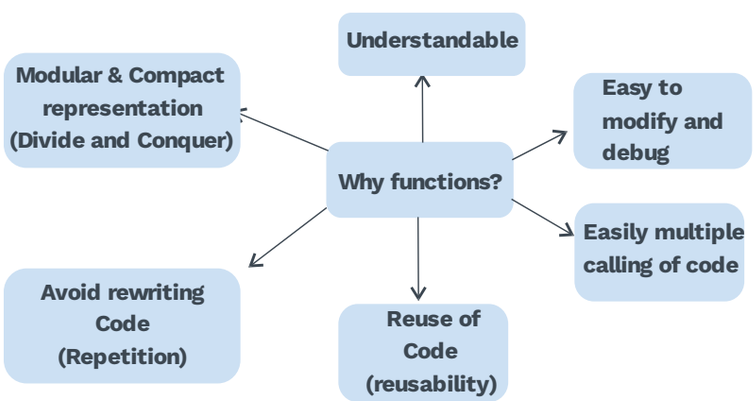


Fig. 1.33

Types of functions:

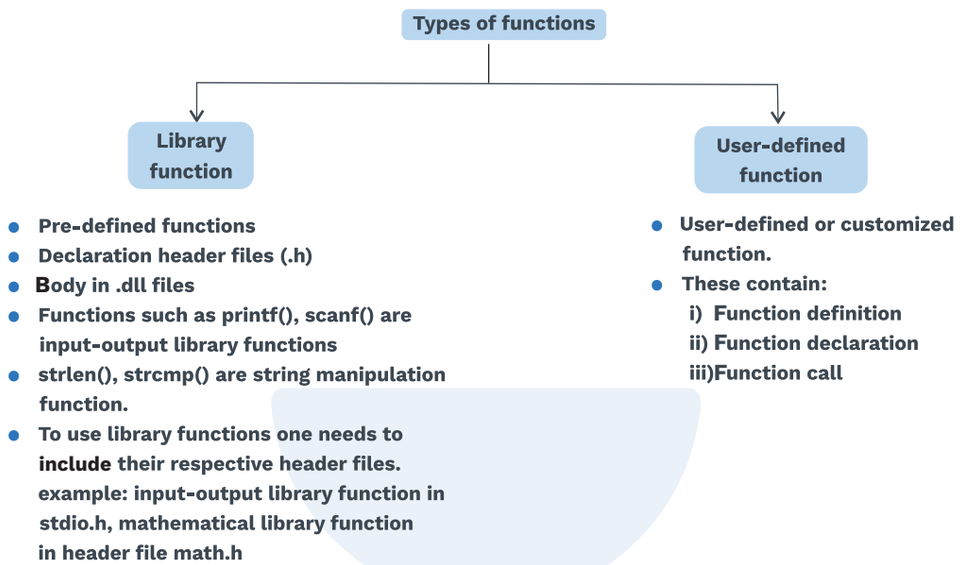


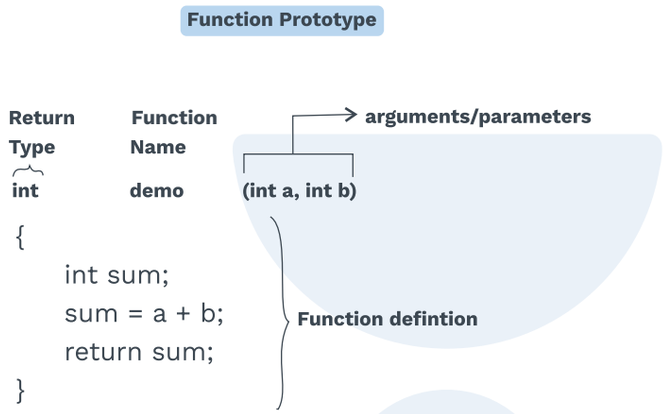
Fig. 1.33 Types of Functions

S.No.	Function Aspects	Syntax	Meaning
1)	Function declaration	return_type function_name (argument)	• Must be declared globally for the compiler to know about the function parameters and return type.
2)	Function call	function_name (argument)	• Can be called from anywhere in a program. • Parameter should be same for function declaration & functions call.
3)	Function definition	return_type function_name (argument) {function body; }	• Contains actual statements which are to be executed.

Table 1.22

Syntax:

```
return_type function_name (arguments)
{
        Statement;
        _____
        _____
        _____
}
```



Example: Program to find sum of two numbers:

```
#include <stdio.h>
int sum (int x, int y); //function declaration
main ()                                          Calling function ?
{       int a, b, s;                             → main()
        printf("enter the value of a & b:");     Called function?
        scanf("%d%d", &a, &b);                          → sum()
        s = sum(a, b); //function
        printf("sum=%d",b);
}
int sum (int x, int y) //function definition
{
        int s;
        s = x + y;
        return s;
}
```

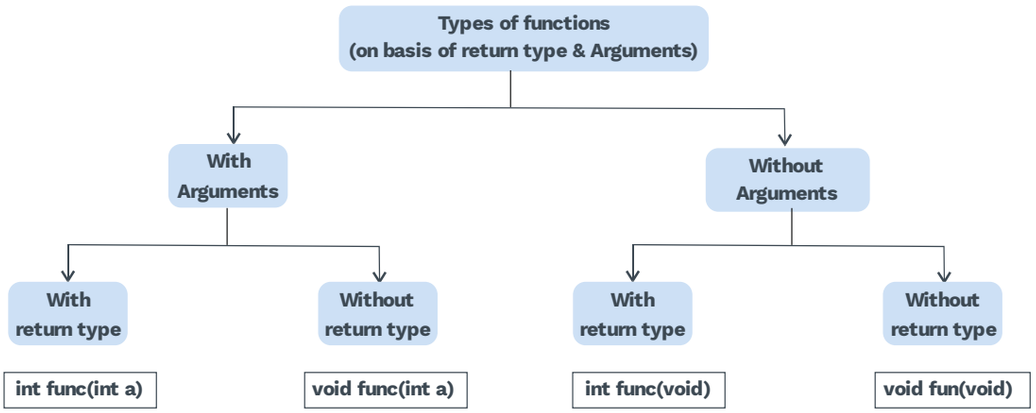


Fig. 1.34 Types of Functions Basis of Return Type and Arguments

Function arguments:

- The calling function sends some values to the called function, these values are known as arguments or parameters.

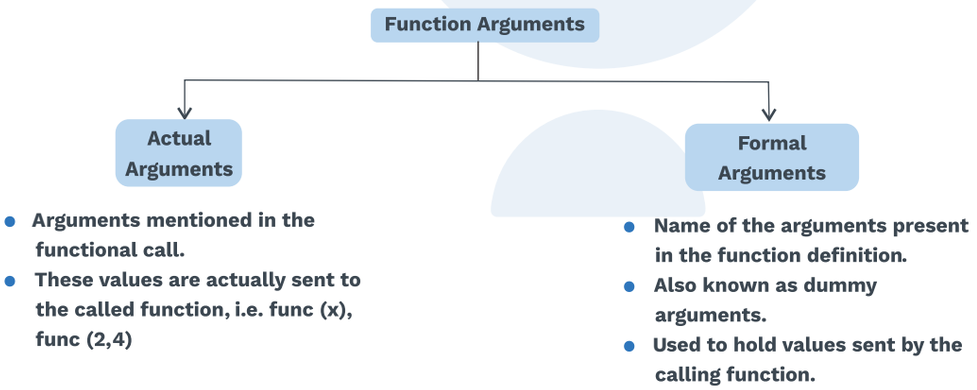


Fig. 1.35 Types of Function Arguments

Note

The order, number and type of actual arguments in the function call should match with the order, number and type of formal arguments in the function definition.

Example: Program showing formal and actual arguments:

```
#include <stdio.h>
mul(int p, int q)//formal arguments
{       int prod;
        prod = p*q;
        return prod;
}
sum (int p, int q)//formal arguments
```

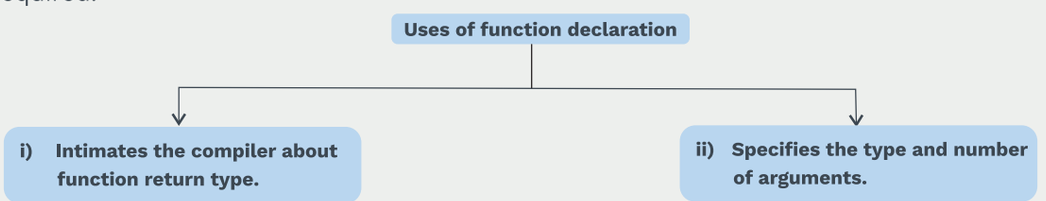
```
{       int s;
        s = p + q;
        return s;
}
main()
{int a = 2, b = 3;
        printf("%d\t", mul(a, b));//actual arguments
        printf("%d\t", mul(5, 4));//actual arguments
        printf("%d\t", mul(a+b, b–a));//actual arguments
        printf("%d\t", sum(a, b));
        printf("%d\t", sum(mul(a, b), b));
}
```

Output: 6 20 5 5 9

Grey Matter Alert!

1) If function definition occurs before the function call, then function declaration is not required.

2)



3) Function declaration is optional, but is a GOOD PRACTICE.

4) Declaration is absent.

Case I: Actual Arguments more than formal arguments.

- extra actual arguments are ignored.

Case II: Actual Arguments are less than formal arguments.

- extra formal arguments receive garbage value.

Example : function (int x, int y, int z)

```
{ _____
  _____
  _____
}
```

Case I:

```
function (1, 2, 3, 4, 5);
                  └──→Actual Arguments ignored.
```

Case II:

```
function (1, 2):
          └──→Missing actual argument:  The formal
              arguments takes garbage value.
```

Case II: function (1, 2):
→ Missing actual argument: The formal arguments takes garbage value.

5) Order of evaluation of function arguments:

- unspecified and compiler dependent

Example: int a = 4, m ;
m = multiply(a, a++);.....(1)

Let function multiply(a,b) be performing a * b, and returning the product, then if (1) is evaluated from left to right, answer would be 16.

But if (1) is evaluated from right to left, then answer would be 20.

Result is unpredictable and varies from compiler to compiler.

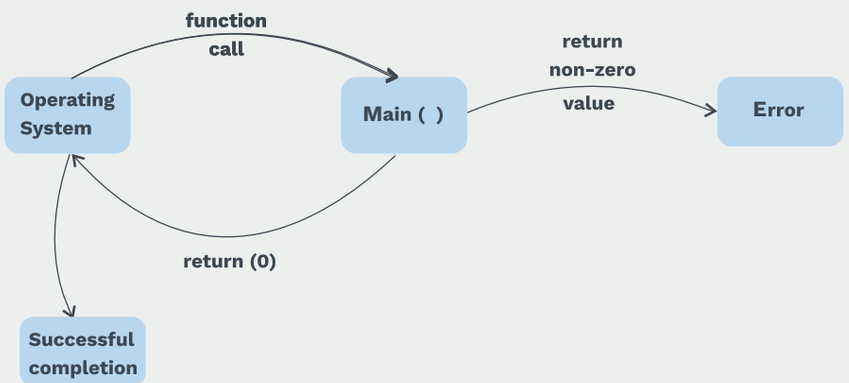
"Avoid such argument expression."

6) main() function

- Only one main() function.
- Each function is directly or indirectly called main().
- Each function after execution returns the control back to the main().

S.No.	Task	Performed by
1)	Function Declaration	C-compiler
2)	Function Definition	Programmer
3)	Function call	Operating System

- main () can also take arguments.



- Calling an exit() function with an integer return type is equivalent to returning values from main().
- If no return and if type specified for main () then any garbage value is returned automatically.

automatically.

7) L brary functions are not formally a part of C-language but, are supplied with every C-compiler.

8) One can define their own C-function with the same name and arguments as that of a pre-defined function. Here the user-defined function, then takes precedence over the library function.

String library functions:

- There are several library functions to manipulate strings, all present in the header file strings.

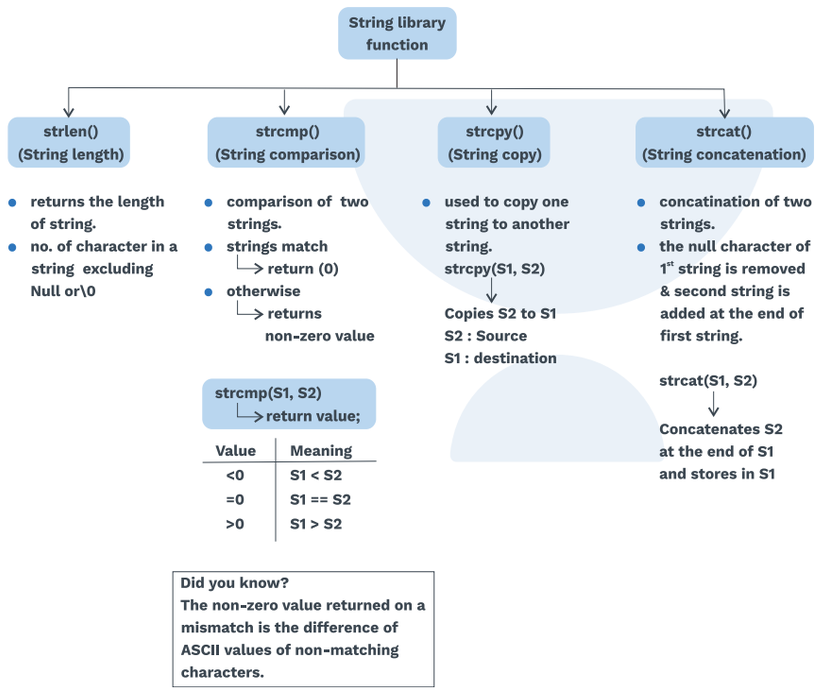


Fig. 1.36 String Library Functions

Example: Consider the given C-program

```
#include < stdio.h>
#include < string.h>
main ()
{       char S1 = “Banglore”;
        char str S3[10];
        char S2 = “Manglore”;

        int length1=strlen (S1);
        int length2=strlen (S2);
        printf (“length S1: %d\n”,length1);
        printf(“length S2:%d\n”,length2);
        if ((strcmp(S1, S2)) == 0)
```

```
            printf("strings are same\n");
        else
            printf("strings are different\n");
        strcpy (S3, S1);
        printf ("S3 string: %s\n", S3);

        strcpy (S1, S2);
        printf ("new string: %s\n", S1);
}
```

Output:

```
length S1: 8
length S2: 8
Strings different
S3 string: Banglore
New string: Banglore, Manglore
```

Local, global and static variables:

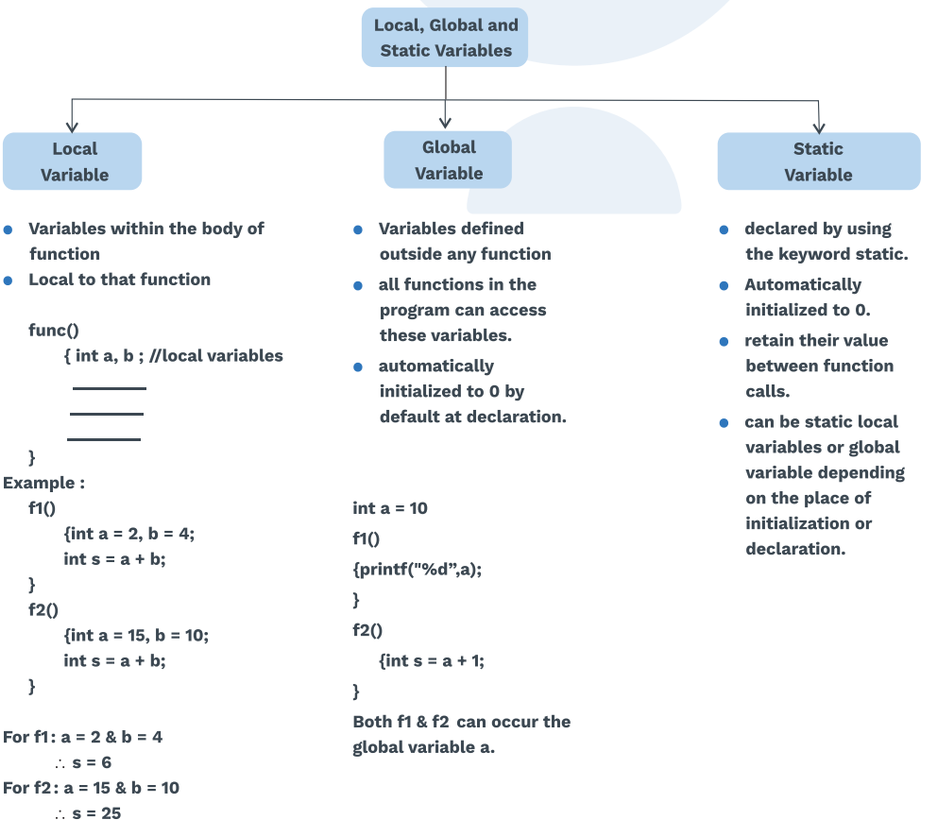


Fig. 1.37 Types of Variables

```
# include <stdio.h>
   double x;  ------>global variable
   static int y ;  ------>Static global variable

   void func1 ( int a, int b )  ------>  parameter variables
       {static int z;  ------>  Static Local variable
        double sum;  ------>  Local variable
          _____
          _____
          _____
          _____
       }

   void func2 ( double a, double b )  ------>parameter variables
       { float P;  ------>  local variable
          _____
          _____
          _____
       }
```

Recursion:

- It is the process when a function calls itself.
- Powerful technique whose complicated algorithms are solved by the divide and conquer approach.
- The sub-problems are defined in terms of the problem itself.

A function should have the capability of calling itself.

> The function that calls itself, again and again, is called a recursive function.

Example:

```
main ( )
{  _____
   _____
   rec ( ) ;
   _____
}
rec ( )
{  _____
   _____
   rec ( );  ------>recursive call
}                    |
                     v
```

As rec ( ) is called within the body of function rec ( ).

Point to remember:

- One should be able to define the solution of the problem in terms of a similar type of smaller problem.
- There should be a termination condition; otherwise, the recursive call would never terminate.

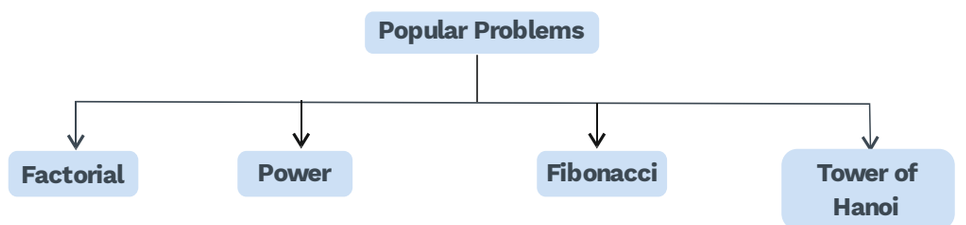


Fig. 1.38

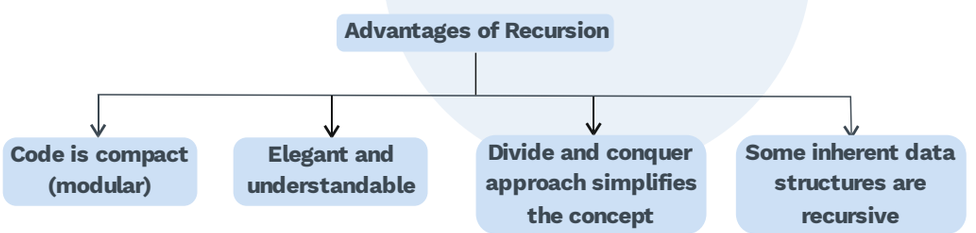


Fig. 1.39

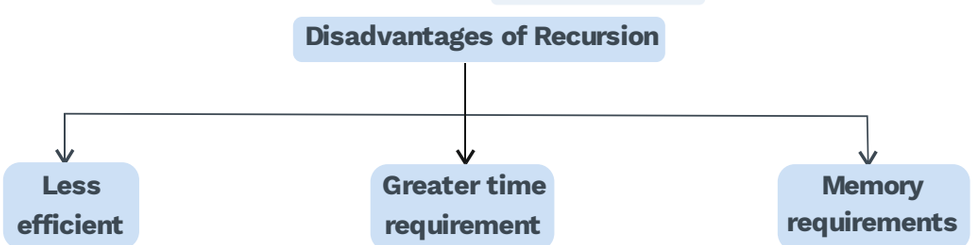


Fig. 1.40

Factorial:

Factorial of a positive integer n can be represented as a product of all integers from 1 to n.

n! = 1 * 2 * 3 * ..... *n

It can be written as:

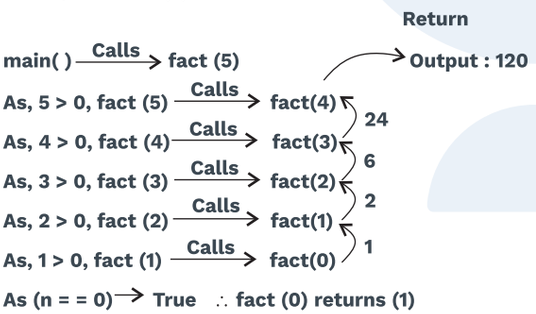
n! = n * (n – 1)!

Here: $n! = \begin{cases} 1 & ;n = 0 \\ n * (n-1)! & ;n > 0 \end{cases}$

Program to find the factorial:

```
#include <stdio.h>
long fact (int n);
main()
{       int num;
        printf("enter the number");
        scanf("%d" & num);
        printf("factorial: %ld", fact(num));
}
long fact(int n)
{       if (n = = 0)
          return(1);
        else
          return (n*fact(n–1));
}
```

Let num = 5, then fact(5) :



Fibonacci

```
int fib (int n)
    {if (n = = 0 || n = = 1)
        return (1);
    else
        return (fib (n – 1) + fib (n –2));
    }
```

$$fib(n) = \begin{cases} 1 & ; (n = 0) \text{ or } (n = 1) \\ fib(n-1) + fib(n-2) & ; (n > 1) \end{cases}$$

Tower of hanoi

```
TOH (source, temporary, destination, n)
{   if (n > 0)
        TOH (source, destination, temporary, n – 1);
        printf("move disk from %d%c →%c\n", n, source destination);
    TOH (temporary, source, destination, n – 1);
    }
```

Rack Your Brain

1) GCD of two numbers.
2) Finding second maximum of three distinct numbers.
Hint. **1)** GCD of two numbers:

```
f(a, b)
{ if(a == 0)
    return b;
if(b = = 0)
    return a;
if(a == b)
    return a;
else
    if(a > b)
    return f(a – b, b);
else
    return f(a, b – a)
}
```

3) Second maximum of three distinct numbers:

```
int func(int a, int b, int c)
            {   if((a >= b) && (c < b))
                  return b;
                else
                if(a > = b)
                  return func(a, c, b);
                else
                return func(b, a, c); }
```

Value passing in functions:

Type of Value passing	Actual Arguments changed	Formal Arguments changed	Analogous value passing
1) Call by value	x	√	—
2) Call by reference	√	√	—
3) call by constant	x	x	—
4) Call by need	x	√	Call by value
5) Call by result	√	√	Call by reference
6) Call by restore	√	√	Call by reference
7) Call by text	√	—	Call by reference
8) Call by name	√	—	Call by reference

Table 1.23

Grey Matter Alert!

- Call by reference and call by name parameter passing produces different results when address of an array element is passed as an argument (parameters).
- If actual arguments are:
  - **Scalar variable : Call by name is equivalent to call by reference.**
  - **Constant expression : Call by name is equivalent to call by value.**
- Value passing is often referred to as parameter passing or argument passing.
- Here is a parameter to the name for the data that goes into a function.
  Pre-defined function. Here the user-defined function then takes precedence over the library function.

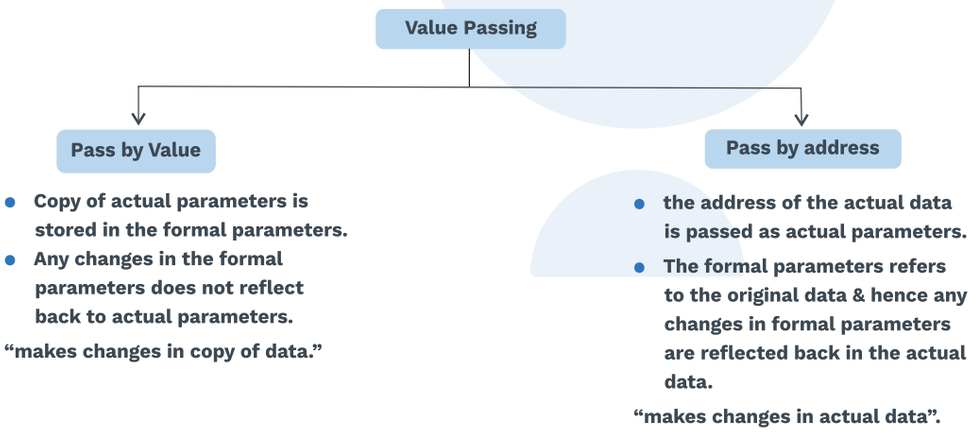
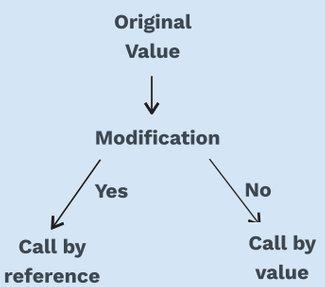


Fig. 1.41 Types of Value Passing Techniques

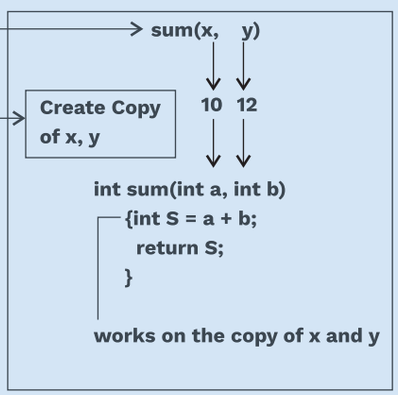
Note

- Words such as argument & parameter are used interchangeably. Similarly, call and pass are used. Call by value is the same as pass by value.
  Arrays are always passed by reference.
- **Address variable:** stores the address of another data variable. They are usually pointer variable.
  Example: int i = 10; //Data variable;
  int *ptr = &i;//pointer variable ptr storing address of variable i.
- **Reference variable:** alias to another variable.
  Example: int i = 10; //a data variable i
  int &r = i;//reference r to the data variable i.
- C only supports call by value and call by address. Call by reference is a feature of C++.

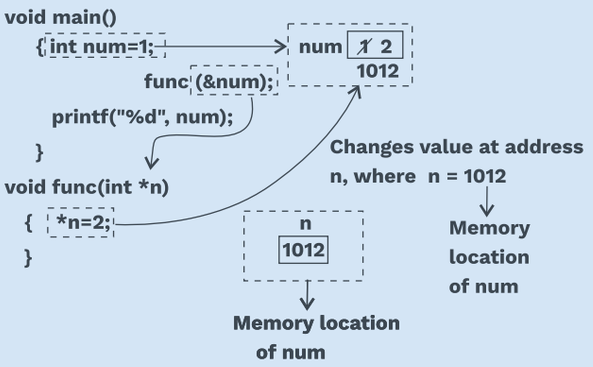


Example: Call by Value

```
(All by value)
# include <stdio.h>
int sum(int a, int b);
    main()
    { int x = 10, y = 12;
      int S = sum(x, y);
      printf("Sum=%d", S);
    }
int sum(int a, int b)
    { int S = a + b;
      return S;
    }
Output : Sum = 22
```



Example: Call by address



	Call by value	Call by reference	Call by address
1) **Definition**	A way of calling function where the actual arguments are copied to formal arguments.	A way of passing the arguments by copying the reference of an arguments into the parameters.	A way of passing values where the address of actual arguments are passed and is copied to formal arguments.
2) **Memory**	Memory is allocated for both actual and formal arguments.	Memory is allocated only for actual arguments. The formal arguments share the memory.	Memory is allocated to both actual arguments and formal arguments.
3) **Actual parameters**	Variables or constants	Variable	Address of variable
4) **Formal parameters**	Variables	Reference variable	Pointer variable (Address variable)

5) **Example**	void main()
	{int i=10;
	func(i);
	printf(”%d”,i);
	}
	func(int x)
	{x=x+1;
	printf(”%d”,x);
	}
		void main()
		{int i=10;
		func(i);
		printf(”%d”,i);
		}
		func(int &p)
		{p=20
		printf(”%d”,p);
		}
			void main()
			{int i=10;
			func(&i)
			printf(”%d”,i);
			}
			func(int *n)
			{ *n=20;
			print(”%d”,n);
			}

6) **Output**	10 11	20 20	20 20

Table 1.24 Call by Value vs Call by Reference vs Call by Address

Previous Years' Question

Consider the following C program:

```
#include < stdio.h>
    int r()
    {   static int num = 7;
        return num--;
    }
    int main()
    {   for(r() ; r(); r())
            printf("%d", r());
        return 0;
    }
```

Which one of the following values will be displayed during the execution of the programs?

a) 41 **b)** 630 **c)** 63 **d)** 52

Sol: d) **(GATE-2019)**

Hint:

for(expression 1 ; expression 2 ; expression 3)

- expression 1: executed only once
- expression 2: executed every iteration before code block execution
- expression 3: executed every iteration after code block execution

Previous Years' Question

Consider the following C function:

```
void convert(int n ) {
    if(n < 0)
    printf("%d", n);
else {
        convert(n/2);
        printf("%d",n%2);
    }
}
```

Which one of the following will happen when the function convert is called with any positive integer n as an argument?

a) It will print the binary representation of n in the reverse order and terminate.
b) It will print the binary representation of n but will not terminate.
c) It will not print anything and will not terminate.
d) It will print the binary representation of n and terminate.

Sol: d) **(GATE-2019)**

Previous Years' Question

Consider the following recursive C function:

```
void get(int n) {
    if(n < 1) return;
    get(n – 1);
    get(n – 3);
    printf("%d", n);
}
```

If get(6) function is being called in main(), then how many times will the get() function be invoked before returning to the main()?

a) 15 **b)** 25 **c)** 35 **d)** 45

Sol: b) **(GATE-2015 (Set-3))**

Previous Years' Question

Consider the following C program:

```
#include <stdio.h>
int f1(void);
int f2(void);
int f3(void);
int x = 10;
int main()
{   int x = 1;
    x+ = f1() + f2() + f3() + f2();
    printf("%d", x);
    return 0;
}
int f1(){int x = 25; x++; return x;}
int f2(){static int x = 50; x++; return x;}
int f3(){x * = 10; return x;}
```

The output of the program is _____.

Sol: 230 to 230 **(GATE-2015 (Set-3))**

1.9 POINTERS AND ARRAYS

1.9.1 Pointers and addresses:

- A pointer is a variable which stores the memory address of another variable in the memory.
- Real power of C lies in pointers.
- The memory of the computer is made up of bytes arranged in a sequential.
- Each byte has an index number called the address of that byte.
- If there are n bytes, the address would vary from 0 to (n – 1) bytes.

Example: 64 MB RAM

then, 64 MB

$\Rightarrow$ 64×2^{20} bytes

$\Rightarrow$ 67108864 bytes

Then: the address of these bytes would be: 0 to 67108863.

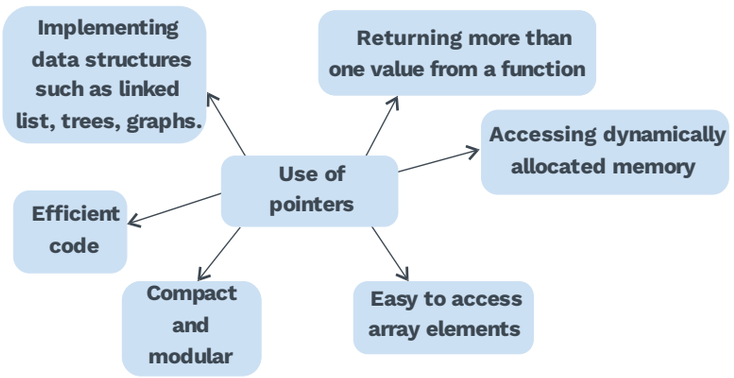


Fig. 1.42

Address operator:

- denoted by '&' (Ampersand) and read as 'address of'.
- returns the address of a variable when placed before it.

Example: &a: address of a

- Popularly used in scanf() function.

Program to print address of variable.

```
#include <stdio.h>
main ()
        {       int var = 10;
                float demo = 11.1;
                printf("value of var = %d, address of var = %u\n", var, &var);
                printf("value of demo= %f, address of demo = %u\n",
        demo, & demo);
        }
```

Output: Value of var = 10, address of var = 65524
Value of demo = 11.100000, address of demo = 65520

Note

- %u control sequence to print address; it doesn't mean that addresses are unsigned. Since there are no specific control sequences for addresses and addresses are also just whole numbers.
- The addresses printed may be different each time one runs the program. It depends on the part of the memory allocated by the operating system.
- Address operator cannot be used with constant or expression:
  valid : &j; &arr [1];
  Invalid : &12; &(j+k);

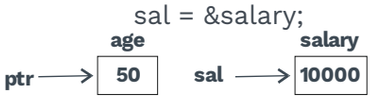
Pointer variables:

- Stores the memory address.
- Like all variables, it has a name, is declared and occupies space in memory.
- **Pointer:** Because it points to a memory location.

Syntax: data_type *pointer_name;
int *ptr

- * (asterik) often read as "value at".

Example: int *ptr, age = 50;
int *sal, salary = 10000;
ptr = &age;
sal = &salary;



- *ptr means value at variable ptr, since ptr stores the address of age.

∴ *ptr = * (&age) which is value at address of age.

This returns the value of variable age = 50

- Similarly: as sal = &salary

∴ *sal= * (&salary), which means the value at the address of salary.

- One can access the variable 'age' by using (*ptr) since ptr is storing the location of age, hence (*ptr), can be used in place of the variable name. This is called **dereferencing pointer variables.**

Example: int a = 10;
int b = 21;
int *ptr1=&a;
int *ptr2=&b;
Now:

Statement	Equivalent to
*ptr1 = 9	a = 9;
(*ptr1)++;	a++;
x=*ptr2+1;	x = b + 1;
printf("%d%d",*ptr1, *ptr2);	printf("%d%d", a, b);
scanf("%d%d", ptr1, ptr2);	scanf("%d%d", & a, & b);

Table 1.25

- Often referred to as indirection operation translating to "value at the address."

Note

- Value at and address of operator cancel each other out. (*, & operators cancel out each other.)
  Example: int age = 10;
  int *ptr = &age;
  then: *(&age) translates to value at address of age which is the value of variable age.
  hence: * (&age) = age

Example:

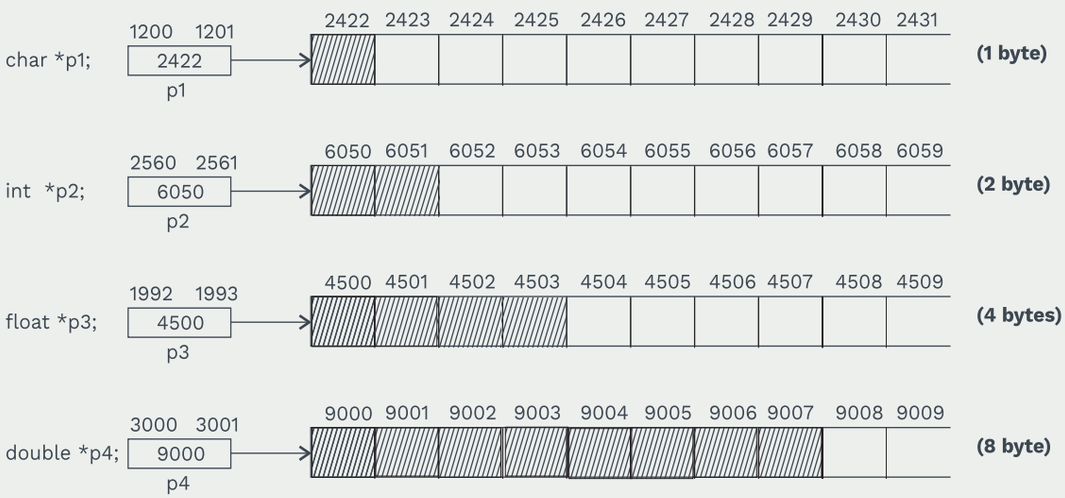
```
#include <stdio.h>
main()
{       int a = 67;
    float b = 45.6;
    int*ptr1=&a;
    float *ptr2 = &b;
    printf("Value of ptr1 = Address of a = %u\n", ptr1);
    printf("Value of ptr2 = Address of b = %u\n", ptr2);
    printf("Address of ptr1 = %u\n", &ptr1);
    printf("Address of ptr2 = %u\n", &ptr2);
    printf("Value of a = %d%d%d\n", a, *ptr1, *(&a));
    printf("Value of b = %f%f%f\n", b, *ptr2, *(&b));
}
```

Output:

```
Value of ptr1 = Address of a = 65524
Value of ptr2 = Address of b = 65520
Address of ptr1 = 65518
Address of ptr2 = 65516
Value of a = 67 67 67
Value of b = 45.600000 45.600000 45.600000
```

Grey Matter Alert!

- The size of the pointer variable is the same for all types of pointers but the memory that will be accessed while dereferencing is different.



size of the pointer variable	size of the value the pointer points to
sizeof(p1)=2	sizeof(*p1)=1
sizeof(p2)=2	sizeof(*p2)=2
sizeof(p3)=2	sizeof(*p3)=4
sizeof(p4)=2	sizeof(*p4)=8

- The size of int, float, and double are compiler, and architecture-dependent. This example is of a 16-bit compiler.

Pointer arithmetic:

- All types of operations are not possible with pointers.

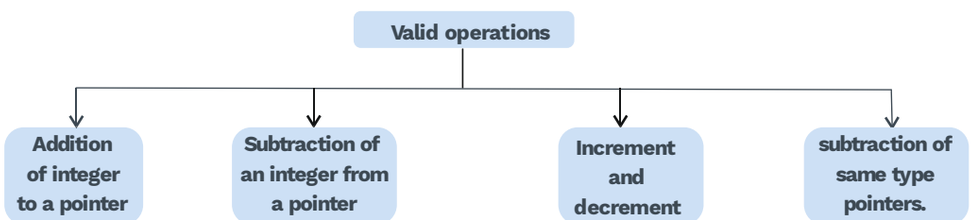
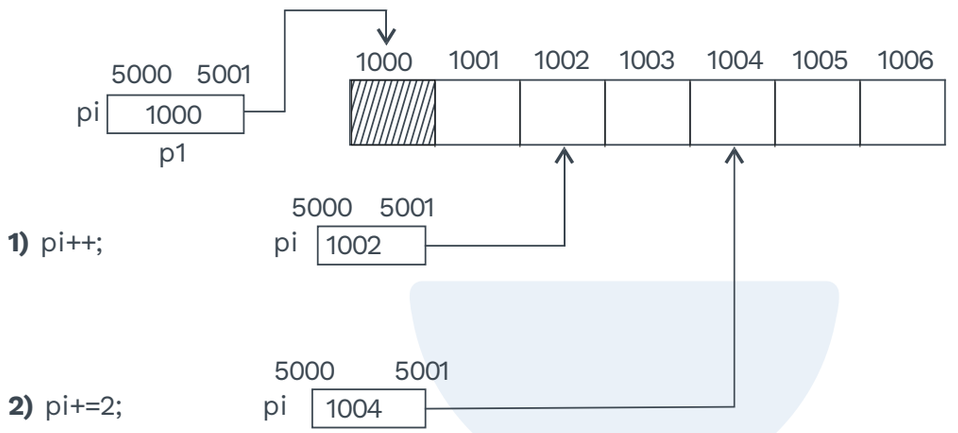


Fig. 1.43 Types of Value Passing Techniques

- Pointer arithmetic is different since it is performed relative to the size of the base type of pointer.

Example: int *pi;
if: pi = 1000, let int size be 2 bytes, then (pi)++; is 1002 and not 1001.
This is because the data type int occupies 2 bytes considering 16-bit computer.



pi = (pi + 2) = 1000 + 2*(size of int)
pi = (pi + 2) = 1000 + 2*2
pi = (pi + 3) = 1004

Note

Precedence and Associativity of Dereferencing operator (*), increment (++) and decrement (––) operator is same and from right to left respectively.

S.No.	Pointer Expression	Meaning

1)	x = *ptr++;	x = *(ptr++);
		Since associativity is from right to left.
		∴ (++) associativity first to ptr.
		x = *(++ptr);

2)	x = *++ptr;	x = *(++ptr);
3)	x=++*ptr;	x = ++(*ptr); Here (++) is applied to (*ptr) not ptr.
4)	x = (*ptr)++;	First assign value of (*ptr) to x and then increment (*ptr); since it is postfix increment.

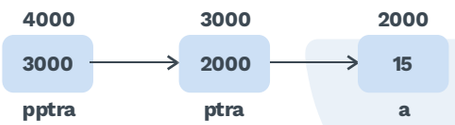
Table 1.26

Pointer to pointer:

- When the address of a pointer variable is stored in another variable, it is called pointer to a pointer variable.
- Similarly, one can have a pointer to pointer to a pointer variable, and this can be extended to any limit.

Syntax: data_type **ptr;
here: ptr is a pointer to pointer

Example:
```
int a = 15;
int *ptra = &a;
int **pptra = &ptra;
```



	Representation			Meaning/value
Value of a	a	*ptra	**pptra	5
Address of a	&a	ptra	*ptra	2000
Value of ptra	&a	ptra	*ptra	2000
Address of ptra		&ptra	pptra	3000
Value of pptra		&ptra	pptra	3000
Address of pptra			pptra	4000

Table 1.27

Previous Years' Question

Consider the following C-program:

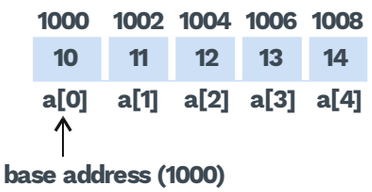
```
#include <stdio.h>
int main() {
static int a[] = {10, 20, 30, 40, 50};
static int *p[] = {a, a+3, a+4, a+1, a+2};
int **ptr = p; ptr++;
printf(" %d %d", ptr-p, **ptr);
}
```

The output of the program is _____.

Sol: 140 to 140 **(GATE-2015 (Set-3))**

Pointers and arrays:

- Elements of an array are stored in contiguous memory location.
  int a[5] = {10, 11, 12, 13, 14};
  Let int occupy 2 bytes.
  Then:



- There is a close-knit relationship between a pointer and an array. Array name is a pointer that only refers to the initial address of the array known as the base address of the array.
- Compiler accesses the array elements by converting subscript notation to pointer notation.
- Since the name of the array is a constant pointer that points to the first element of the array, therefore by pointer arithmetic when pointer variable is incremented, it points to fix the location of its base type, thereby making it possible to access all the elements of the array.

Example:
int a[5] = {10, 11, 12, 13, 14};

Representation	Equivalent to	Value
a	&a[0]	1000
a+1	&a[1]	1002
a+2	&a[2]	1004
a+3	&a[3]	1006
a+4	&a[4]	1008
*a	a[0]	10
*(a+1)	a[1]	11
*(a+2)	a[2]	12
*(a+3)	a[3]	13
*(a+4)	a[4]	14

Table 1.28

Note

- Accessing arrays by pointer notation is faster than accessing them by subscript notation since the compiler also ultimately changes the subscript notation to pointer notation.
- Array sub-scripting is commutative i.e. a[i] = i[a]

Pointer to an array:

- Useful in multidimensional Arrays

Syntax:

data_type pointer[array size];
int *ptr[10];//ptr is a pointer to an array of size 10.

Note

Here ptr is a pointer to an whole array of 10 elements and not to the first element of array.

Example:

Program to differentiate between pointer to integer and pointer to an array of integer.

```
#include <stdio.h>
main() {
    int *p;
    int(*ptr)[5];
    int a[5] = {1, 2, 3, 4,};
    p = a; // points to the first element of array a
    ptr = a; // points to the whole array a.
    printf(" p = %u , ptr = %u \n", p, ptr);
    p++; ptr++;
    printf(" p = %u, ptr = %u \n", p, ptr);
}
```

Output:

```
//(let int occupy 2 byte)
p = 1000, ptr = 1000
p = 1002, ptr = 1010
```

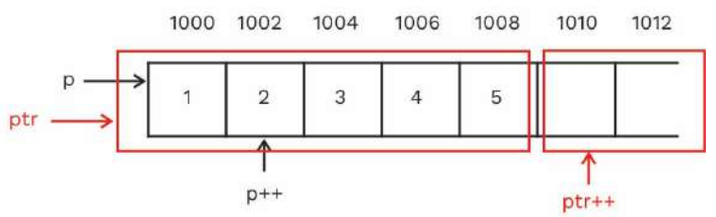


Fig. 1.44

Also, sizeof(p) = 2 sizeof(*p) = 2
sizeof(ptr) = 2 sizeof(*ptr) = 10

Pointers and 2D arrays:

- In 2-D arrays, the first subscript represents rows and the second subscript represents column.

Example: int a[3][4] = {{10, 11, 12, 13}, {20, 21, 22, 23}, {30, 31, 32, 33}};

	Column 0	Column 1	Column 2	Column 3
Row 0	10	11	12	13
Row 1	20	21	22	23
Row 2	30	31	32	33

- 2D arrays are stored in Row major order, i.e. rows are placed next to each other.

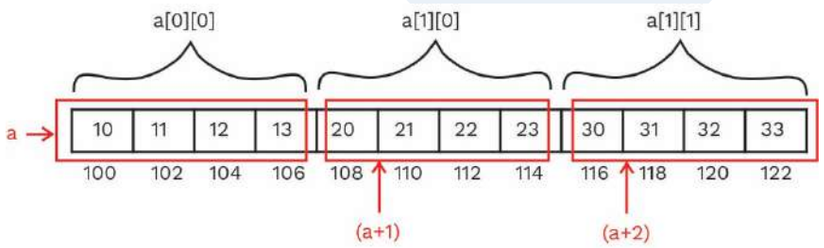


Fig. 1.45

a: points to 0th 1-D array →address(100)
(a+1): points to 1st 1-D array →address(108)
(a+2): points to 2nd 1-D array →address(106)

Hence: (a+i) points to ith element of a or points to ith 1-D array of a

Now: *(a+i) gives the base address of ith 1-D array.

Both(a+i) and *(a+i) are pointers but their base types are different, here:
(a+i) is an array of 4 integers
*(a+i) is equivalent to a[i] which is integer value here.

Representation	Meaning
a	Points to 0^{th} 1-D array
*a	Points to 0^{th} element of 0^{th} 1-D array
(a+i)	Points to i^{th} 1-D array
*(a+i)	Points to 0^{th} element of i^{th} 1-D array
*(a+i)+j	Points to j^{th} element of i^{th} 1-D array
((a+i)+j)	value of j^{th} element of i^{th} 1-D array

Table 1.29

Considering the above Example:

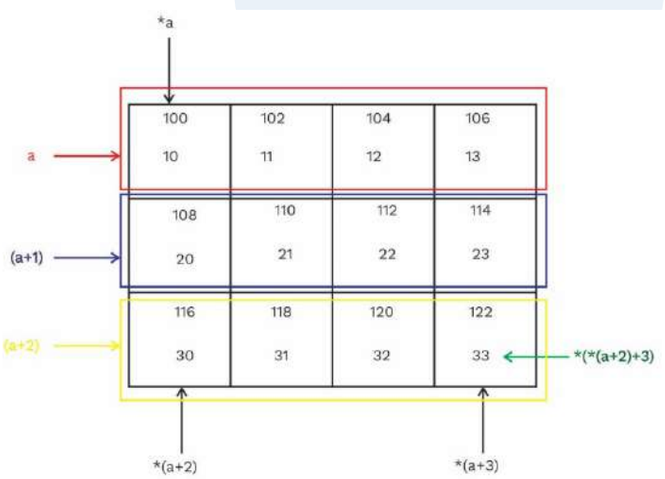


Fig. 1.46

Pointers and 3-D arrays:

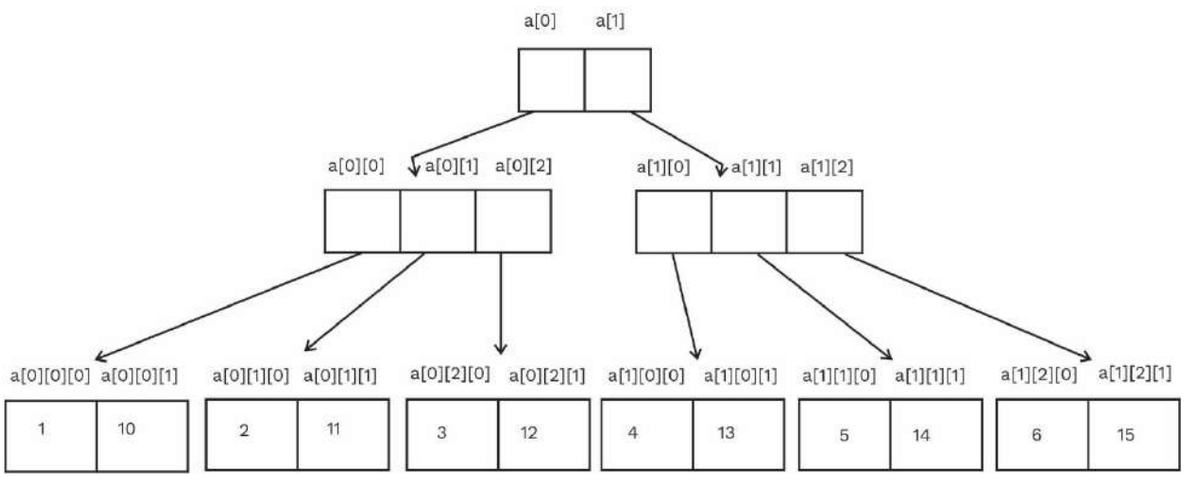
- Elements are accessed using 3 sub-scripts

int a[2][3][2] = {{{1, 10}, {2, 11}, {3, 12}},
{{4, 13}, {5, 14}, {6, 15}}};

- A 3-D array is considered to be an array of 2-D arrays, where each elements is a 2-D array.

Representation	Meaning
a	points to 0^{th} 2-D array
(a+i)	points to i^{th} 2-D array
*(a+i)	gives base address of i^{th} 2-D array (points to 0^{th} element of i^{th} 2-D array). Since each 2-D array is a 1-D array ∴ it points to the 0^{th} 1-D array of i^{th} 2-D array. (equivalent to a[i])
*(a+i)+i	points of j^{th} 1-D array of i^{th} 2-D array
((a+i)+j)	gives base address of j^{th} 1-D array of i^{th} 2-D array so points to 0^{th} element of j^{th} 1-D array of i^{th} 2-D array. (equivalent to points to a[i][j])
((a+i)+j)+k	points to k^{th} element of j^{th} 1-D array of i^{th} 2-D array.
((a+i)+j)+k	gives value of k^{th} element of j^{th} 1D array of i^{th} 2-D array. (equivalent to a[i][j][k]).

Table 1.30



Grey Matter Alert!

a→ address of a[0] ≈ &a[0]

a[0] → address of a[0][0]≈&a[0][0]

a[0][0]→address of a[0][0][0]≈&a[0][0][0]

Rack Your Brain

Let int a[3][2][2] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};
(Assume int of 2 bytes and array start from 100), then fill the table:

Representation	Meaning
a	
a+1	
*a	
*a+1	
&a+1	
**a	
***a	
**a+1	
***a+1	

Table 1.31

Previous Years' Question

Consider the following C-program:

```
#include <stdio.h>
    int main()
    {int a[] = {2, 4, 6, 8, 10};
    int i, sum = 0, *b = a+4;
    for(i=0; i <5; i++)
                    sum = sum +(*b-i) – *(b-i);
    printf("%d \n", sum);
    return 0;
    }
```

The output of the above C program is _____.

Sol: 10 to 10 **(GATE-2019)**

Previous Years' Question

Consider the following C-program:

```
#include <stdio.h>
        int main() {
        int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 5},
        *ip = arr +4;
        printf("%d\n", ip[1]);
        return 0;
}
```

The number that will be displayed on execution of program is _____ .

Sol: 6 to 6 **(GATE-2019)**

Pointers and functions:

- Arguments to the functions can be passed in two ways:
  1) Call by value.
  2) Call by reference.
- function can return pointers too.

Syntax:

```
type *func(type1, type2 ....);
float *func(int, char); // function returns pointer to float.
```

Example:

```
int *fun()
{int x = 5;
    int *p = &x;
       return p;
}
main()
{int *ptr;
    ptr = fun();
       _____
       _____
       _____
    }
```

Let x be stored at 1000 address, then p = 1000.

Character pointer and functions:

- Character pointers are used to initialize string constant.

Example: char *ptr = “Demo”;

Here ptr is a character pointer which points to the first character of the string constant “Demo”, i.e, the base address of this string constant.

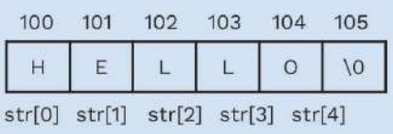
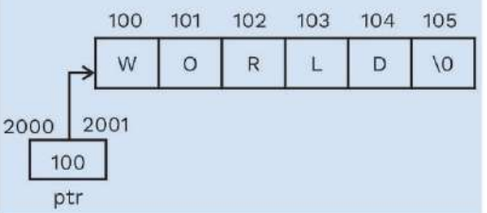
String as arrays	String as pointers
char str[] = “Hello” ; char str[] = { ‘ H’, ‘E’, ‘L’, ‘L’, ‘O’, ‘\0’};  	char *ptr = “World” ; 

Table 1.32 String as Arrays vs String as Pointers

Rack Your Brain

1) Consider the given C-program segments and assume header files are included.

```
main()
    {int i , arr[5] = {2, 3, 5, 4, 6}, *p;
    p = &arr[4];
    for(i = 0; i <5; i ++)
    printf("%d \t%d \t", *(p-i), p[-i]);
}
```

2)

```
main()
    {int i, j;
        int arr[10] = {3, 2, 4, 1, 5, 9, 8, 10, 7, 6};
        for(i=0; i <10; i++)
        for(j=0; j <10-i-1; j++)
        if(*(arr+j) > *(arr + j+1))
        swap(arr+j, arr+j + 1);
        for(i=0; i <10; i++)
        printf("%d \t", arr[i]);
        printf("\n");
    }
        swap(int *b, int *c)
    {int temp;
    temp = *b, *b= *c, *c = temp;
    }
```

Hint : Sorting using bubble sort.

3) If a is declared in C programming language as a 1-D array then which of these statements is/are correct?

1) *(a+i) is same as *(&a[i])
2) *(a+i) is same as *a+i
3) &a[i] is same as a+i-1
4) *(a+i) is same as a[i]

Previous Years' Question

Consider the following C-program:

```
#include <stdio.h>
    void fun1(char *s1, char *s2) {
    char *tmp;
    tmp = s1;
    s1 = s2;
    s2 = tmp;
}
    void fun2(char **s1, char **s2) {
    char *tmp;
    tmp = *s1;
    *s1 = *s2;
    *s2 = tmp;
}
    int main() {
    char *str1 = "Hi", *str2= "Bye";
    fun1(str1, str2); printf("%s %s", str1, str2);
    fun2(&str1, &str2); printf("%s %s", str1, str2);
    retrun 0;
}
```

The output of the program is:

a) Hi Bye Bye Hi
b) Hi Bye Hi Bye
c) Bye Hi Hi Bye
d) Bye Hi Bye Hi

Sol: Option a) **(GATE-2018)**

Previous Years' Question

Consider the following C program

```
#include <stdio.h>
#include <string.h>
int main()
    {char *c = "GATE CSIT 2017";
    char *p = c;
printf("%d",(int) strlen(c+2[p] – 6[p]-1));
return 0;
}
```

The output of the program is _____.

Sol: 2 to 2 **(GATE-2017 (Set-2))**

Previous Years' Question

Consider the following C program

```
#include <stdio.h>
#include <string.h>
        void printlength(char *s, char *t)
        {unsigned int c = 0;
        int len =((strlen(s) – strlen(t)) > c) ? strlen(s): strlen(t);
        printf("%d\n", len);
}
        void main()
        {char *x = "abc";
        char *y = "defgh";
        printlength(x,y);
}
```

Recall that strlen is defined in string.h as returning a value of type size_t, which is an unsigned int. The output of the program is _____.

Sol: 3 to 3 **(GATE-2017 (Set-1))**

1.10 Structures:

Basics of structures:

- A structure is a group of items in which each item is identified by its own identifier, each of which is known as a member of the structure.
- Used to store related fields of different data types.
- Capable of storing heterogeneous data.

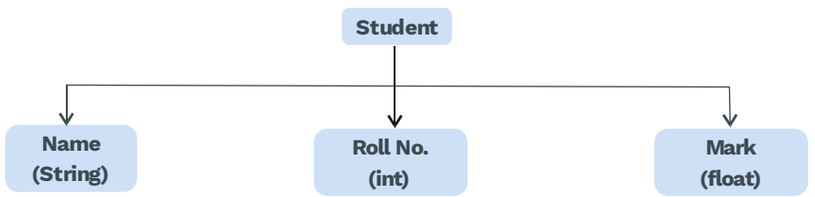


Fig. 1.47

Syntax:

```
struct student{
    char name[20];
    int roll no;
      float marks;
};
```

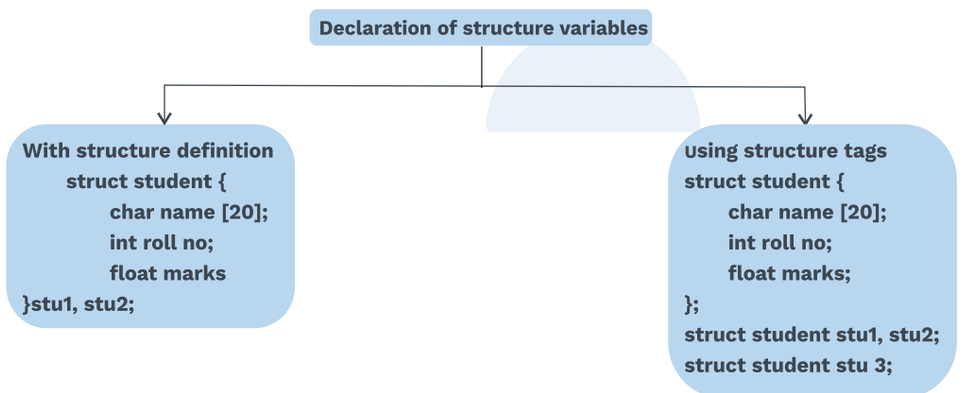


Fig. 1.48

Both declaration creates three variable which enables access individual structure members as:

stu1.name, stu1.rollno, stu1.marks;

Let char occupy 1 byte, int 2 bytes and float 4 bytes then the entire structure student would reserve (1×20+2+4) = 26 bytes.

Structure initialization:

- The number, order, and type of values must be same as the structure template/ definition.

- The initializing values can only be constant expressions.

```
struct student {
char name[20];
int roll no;
float marks;
} stu1 = {“Ravi”, 13, 98};
struct student stu2 = {“Ravi”, 24, 86};
```

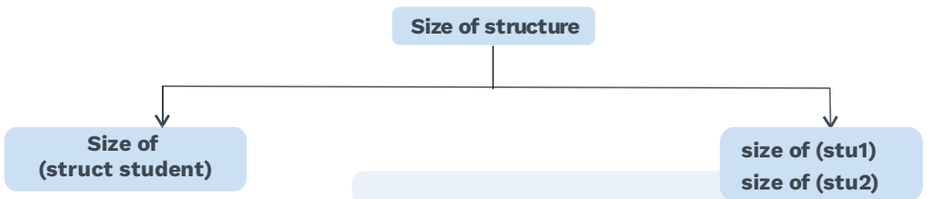


Fig. 1.49

1.10.2 Array of structures:

- each element of an array is a structure type.

Example: struct student stu[10];
Here stu is an array of 10 elements where each element is a structure having three members name, roll no &marks.

stu[0].name	stu[0].roll no	stu[0].marks
stu[1].name	stu[1].roll no	stu[1].marks
.		
.		
.		
.		
.		
.		
.		
stu[9].name	stu[9].name	stu[9].name

Example:

Program to sort by roll no.(bubble sort)

```
#include <stdio.h>
    struct stud
    {int roll;
    char code[25];
    float marks;
    };
```

```
main()
{struct stud class[100], t;
int j, k, n; scanf ("Enter the number of students", & n);
for (k=0; k<n; k++)
scanf(\"Enter roll no., dept code and marks"
%d %s %f", &class[k].roll, &class[k].code, &class[k].marks);
for(j=0; j<n-1; j++)
for(k=j+1; k<n; k++)
{if(class[j].roll > class[k].roll}
{t=class[j];
class[j]=class[k];
class[k]=t;
}
}
for(k=0; k<n; k++){
printf("roll no %d code %s marks%f\n", class[k].roll, class[k].code,
class[k].marks);
}
```

Arrays within structure:

- C allows the use of arrays as structure members.

Example:

```
struct student {
char name[20];
int rollno;
int submarks[4];
};
```

The array submarks represent subject marks. Each student has Four subject and each subject mark is denoted by submarks[0], submarks[1], submarks[2], submarks[3].

Note

one can have nested structures also, i.e. structure within structure.

Rack Your Brain

```
#include <stdio.h>
   struct class
     {int students[7];
     };
int main()
{    struct class c={20, 21, 30, 31, 40, 41};
     int *ptr;
     ptr=(int*) &c;
     printf("%d", *(ptr+2));
     return 0;
}
```

What is the output of the c –program_____ ?

Pointers to structures:

- A pointer to a structure stores the start address of a structure variable.

Example:

```
struct student {
char name[20];
int roll no;
int marks;
};
struct student stu, *ptr;
```

- ptr is a pointer that can point to the variable of type struct student.

```
ptr = &stu;
```

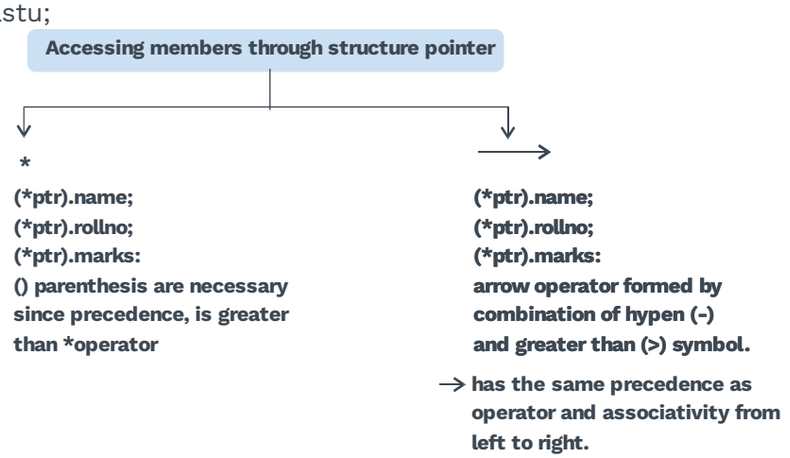


Fig. 1.50

Rack Your Brain

1) i) ptr → roll
   ii) (*ptr).roll
   iii) *ptr.roll
   Does I, II and III mean the same thing?

2) What does the follow represent incrementing ptr or roll no.?
   i) ++ptr → roll
   ii) (++ptr) → roll

Structures and functions:

- structures can be passed as arguments to functions. A function can have a return value as a structure.

Pass as argument to function:	Function can return:
1) pass individual members	1) a structure member
2) whole structure variable	2) whole structure variable
3) structure pointers	3) pointer to a structure

Table 1.33

Example:

Program to add two complex numbers.

```
#include <stdio.h>
struct complex {float re; //real part
                float in; //imaginary part
        }
struct complex x, y;
struct complex add(x,y)
{struct complex t;
t.re=x.re+y.re;
t.im=x.im+y.im;
return(t);}
main()
        {struct complex a, b, c; // variables a, b, c
printf("enter two complex numbers");
scanf("%f %f\n", &a.re, &a.im);
```

```
scanf("%f %f\n", &b.re, &b.im);
c=add(a,b);
printf("The addition of real no. a and b are %f %f \n", c.re, c.im);
}
```

Here the complex no. a has two parts real represented as a.re and imaginary represented as a.im. Similarly, for complex no. b and the result of the addition of complex no. c.

Example:

Program to add complex no. using pointers.

```
#include <stdio.h>
        struct complex{
        float re;
        float im;
} struct complex *x, *y, *t;
void add(x, y, t)
{       t → re = x → re + y → re;
        t → im = x → im + y → im;
}
main()
{       struct complex a, b, c;
printf("enter two complex no.");
scanf("%f %f\n" &a.re, &a.im);
scanf("%f %f", &b.re, &b.im);
add(&a, &b, &c);
printf("the addition of a and b are %f %f", c.re, c.im);
}
```

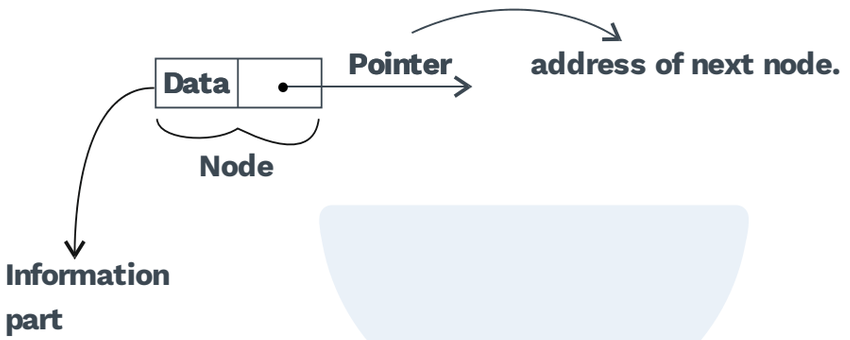
Self-referential structures:

- Structure that contains pointers to structures of its own type

Syntax:

```
struct tag {
        datatype member1;
        datatype member2;
        _________
        _________
        _________
        struct tag *ptr1;
        struct tag *ptr2;
        }
```

Here struct tag is a self-referential structure as it contains Two pointers ptr1 and ptr2 of type struct tag.
Linked lists are the most popular application of self-referential data structures.
A linked list has two parts, a data and a pointer, which together form a node.



Syntax: struct node {int info;
struct node *link;
}

Union:

- It is also a collection of heterogeneous data types.
- The only difference between struct and union is the way memory is allocated to its members.
- In structure: each member has its own memory location
- In union: members share the same memory location.
- When a or the union is declared, the compiler allocates sufficient memory to hold the largest member in the union.

Note: Union is used for saving memory

Syntax: union name

```
{
   datatype member1;
   datatype member2;
   ____
   ____
};
```

Similar to structures, unions can have union variables to access the members

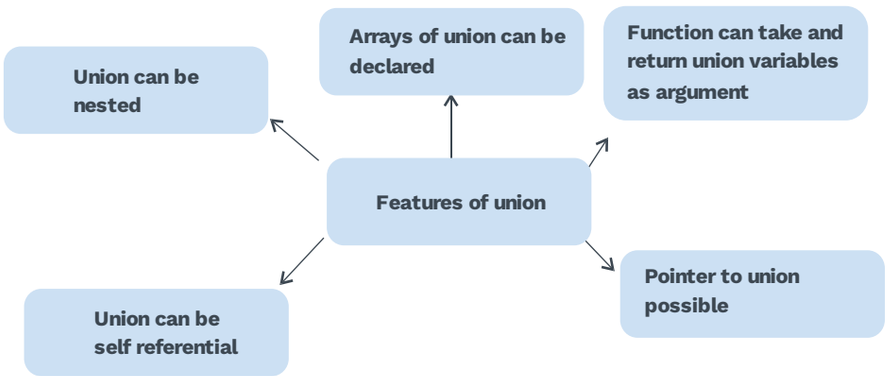


Fig. 1.51

Previous Years' Question

Consider the following C-programs ?

```
#include <stdio.h>
struct ournode
{char x, y, 2;
};

int main() {
struct ournode p = {'1', '0', 'a'+2};
struct ournode *q=&p;
printf("%c, %c",((char*)q+1), *((char*)q+2));
return 0;
}
```

The output of the program is:

a) 0, c **b)** 0, a+2 **c)** '0', 'a+2' **d)** '0', 'c'

Sol: Option a) **(GATE-2018)**

Previous Years' Question

The following c declaration

```
struct node {
int I;
float j;
};
struct node *s[10]:
```

Define s to be.

a) An array, each element of which is a pointer to a structure of type node.
b) A structure of 2 fields, each field being a pointer to an array of 10 elements.
c) A structure of 3 fields: an integer, a float, and an array of 10 elements.
d) An array, each element of which is a structure of type node.

Sol: Option a) **(GATE-2000)**

Typedef:

- allows you to define a new name for an existing datatype.

Syntax: typedef datatype_name new_name;

Example: typedef int marks;

- now marks is a synonym for integer i.e marks sub1, sub2;

	Syntax
1) Pointers	typedef datatype *pointer_name; **Example :** typedef float *fptr; fptr is synonym for float pointer **Valid :** fptr p, q, *r;
2) Arrays	typedef datatype arrayname[size]; **Example :** typedef int arr[10]; arr is a synonym for integer array of 10 elements. **Valid :** arr a, b, c [10];
3) Functions	typedef datatype func_name(arguments) **Example :** typedef floatfunc(float, int); func is synonym for any function taking two argument one float & other integer. **Valid :** func add, sub, null;

4) Structures	typedef struct structure_namevariable_name; **Example :** typedef struct student std; now structures student has a synonym std. **Valid :** std stu 1, stu 2;

Rack Your Brain

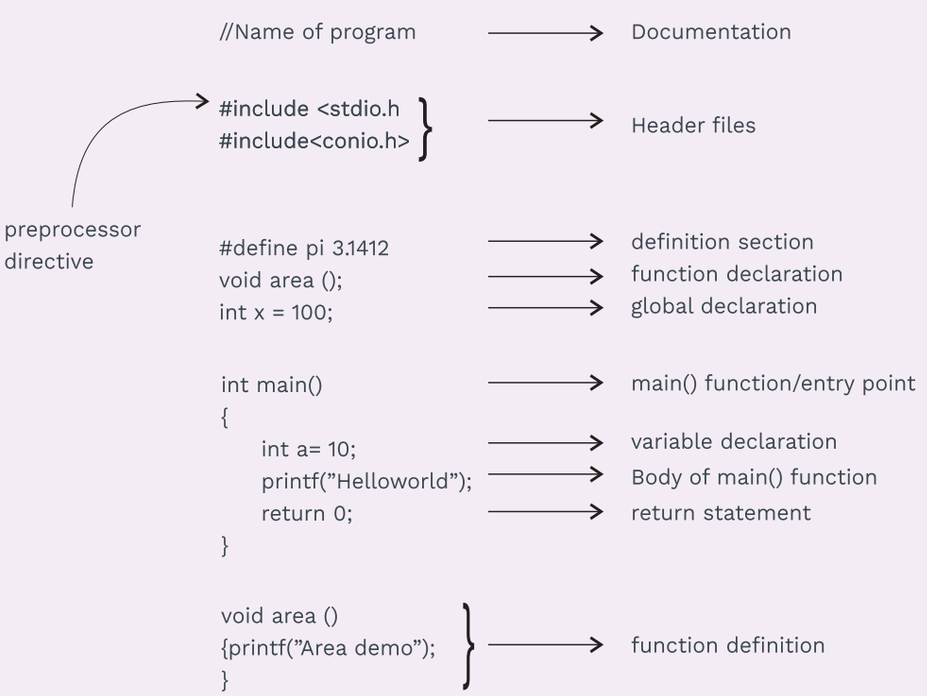
1) Let:
```
struct student
{char name [20];
 int marks;
}
```
then: typedef struct student std;
Which of the following variable declaration is true?
1) student stu1, stu2;
2) struct student stu1, stu2;

2) typedef int a arr; then what would arr A[15]; mean?

3) typedef float *fptr;
Then what would fptr *r; mean?

Chapter Summary

- C-programming given by Dennis Ritchie in 1972.
- Some of its features are portability, syntax based, case-sensitive, middle level, simple yet powerful, fast and efficient.
- The character set includes alphabets, digits and symbols. C has a total of 32 keywords.
- Components of basic C-program;

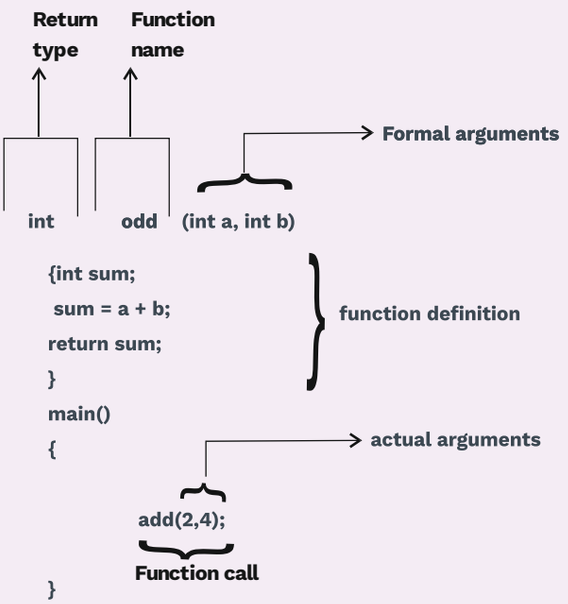


- C has integer, float, double and character as primary data types and structure, union, and enumeration as secondary or user-defined data types.
  int will normally be the natural size for a particular machine. short is often 16 bits long, and int is of 16 or 32 bits. Each compiler allocates storage to each data type based on the hardware, subject only to the the restriction that shorts and ints are at least 16 bits, longs are at least 32 bits, and short is no longer than int which is no longer than long.
- Datatype size is machine-dependent.
- C has four storage classes, namely automatic, static, external and register, that specify the lifetime, scope, initial value & place of storage of a variable.

- C has Arithmetic, Assignment, increment decrement, relational, logical, conditional, comma, sizeof, and bitwise operators to perform various operations.
- Operators are either unary(one operand) or binary(two operands)
- The conditional operator is a ternary operator(?:)
- C supports both implicit and explicit type conversion.
- Precedence and associativity play a vital role in expression evaluation.

- C has three control statement categories namely: 1. selection statements: if, if-else, switch, 2. iteration statements: for, while, do while, 3. jump statements: break, continue, goto, return.
- C has two type of function: user-defined and library functions.

- The library functions are in the header files; hence to use these functions in a program one, needs to include the header files.
- Functions that are defined by the programmer are called user-defined functions.
- Functions aim at making the code modular, compact, understandable, easy to modify, reusability and altogether avoid repetition of code.
- Function can be called with some argument; these are actual and formal arguments/ parameters.
- Actual arguments / parameters are the ones in the function calls.
- Formal arguments / parameters are the ones in the function definition.

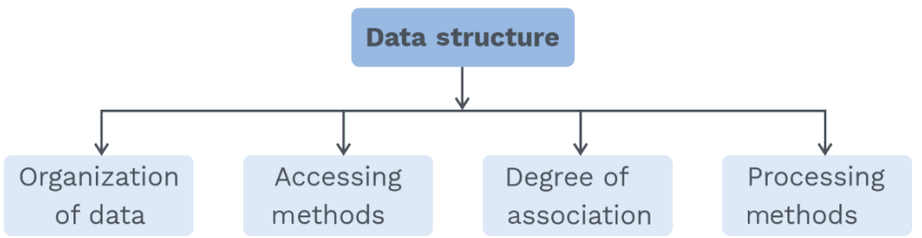


- Recursion is a major application of function but is not preferred as a good programming practice due to the memory and time requirement overhead.
- Pass by value/call be value and pass by reference/ Call by reference are two major value passing techniques in C.
- In pass by value, a copy of actual arguments is stored in formal arguments. The changes in formal argument not reflected back to the actual arguments.
- In pass by reference, the address of actual arguments is passed, and changes in formal arguments are reflected in actual arguments.
- A pointer is a variable which stores the memory address of another variable. Pointers are what make C-programming language really powerful.
- Pointers help in implementing various data structures such a linked lists, trees, and graphs, helps in returning one or more values from functions, provide easy access to array elements, and play major role in dynamic memory allocations. Also makes the code compact, modular and efficient.
- Some of the valid operations on pointer include the addition of integer to a pointer, subtraction of an integer from a pointer, increment and decrement of pointers, subtraction of same type pointers.
- Pointers are used in function in the call by reference function calling.
- Structure and unions are user-defined structures which are heterogeneous. Both can store multiple data types.
- The only difference is that structures reserve memory locations for each datatype, whereas unions reserve memory location for only the largest number of union(members share the memory location).
- Self-referential structures are used to implement data structures such as linked lists.

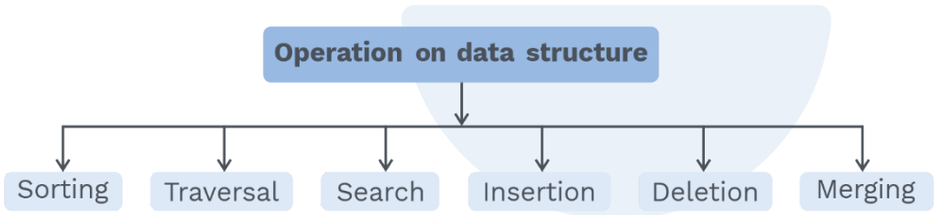
2 Array

DATA STRUCTURES

Way of organizing data in computer memory so that the memory can be efficiently used in terms of both time and space.

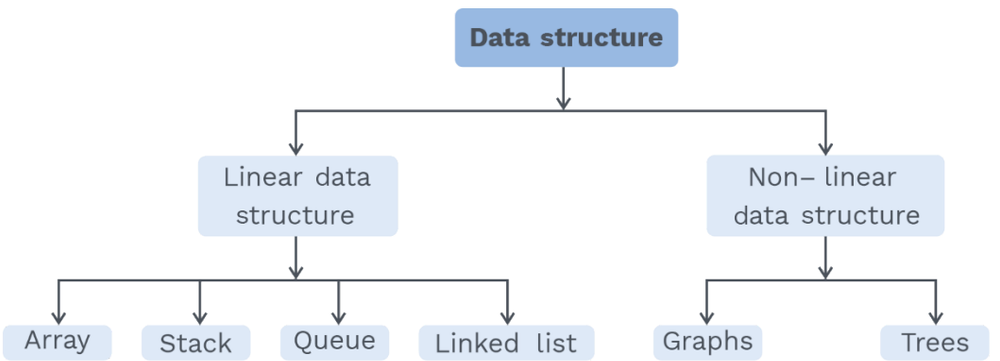


Operations on data structures:



1) Traversal: Visiting each element in the list
2) Search: Finding the location of an element with a given value of the record with a given key.
3) Insertion: Adding a new element to the list.
4) Deletion: Removing an element from the list.
5) Sorting: Arranging the elements.
6) Merging: Combining two lists into one.

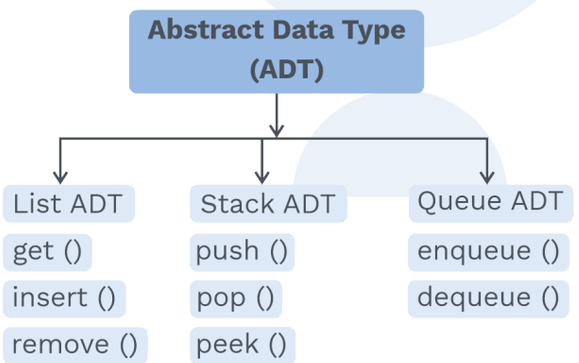
Types of data structures:



Linear data structure:
A Linear relationship between the elements is represented by means of contiguous memory locations. Another way of representing this linear relationship is by using pointers or linked list.
E.g. Array, stack, queue, linked list

Non–linear data structures:
Non–linear data structures are not arranged sequentially in the memory. The elements in non–linear data structures cannot be traversed in a single run. Although they are more memory efficient than linear data structures.
E.g. graphs, trees.

Abstract data type (ADT):
- It is a class or datatype whose objects are defined by a set of values and a set of operations
- Combining data structure with its operation



Previous Years' Question

An Abstract Data Type (ADT) is :
a) same as an abstract class
b) a data type that cannot be instantiated
c) a data type for which only the operations defined on it can be used, but nothing else.
d) all of the above
Sol: c) **(GATE 2005 : 1–Mark)**

2.1 BASICS OF ARRAY

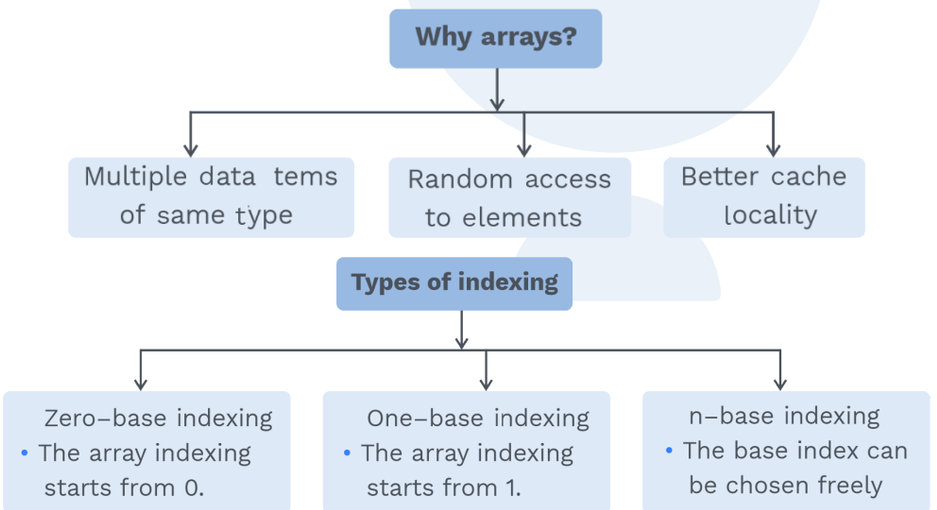
Introduction:

- An array is a collection of similar types of data items.
- Each data item is called an element of an array
- Often referred to as homogeneous collection of data elements.
- The datatype of elements may be any valid data type like char, int or float.
- The elements of the an array share same variable name, but each variable has a different index number. This index number is called the subscript.

Syntax: datatype array_name [Size];

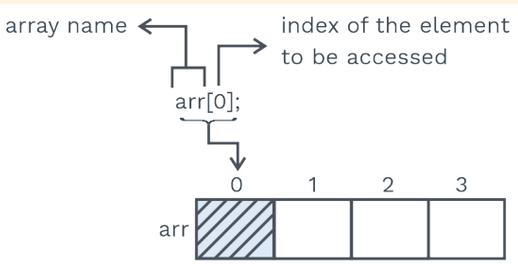
E.g. int arr[10]; //represents an array of 10 elements each of integer datatype.

- The array index starts from 0 in C language; hence arr[0] represents the first element of the array.



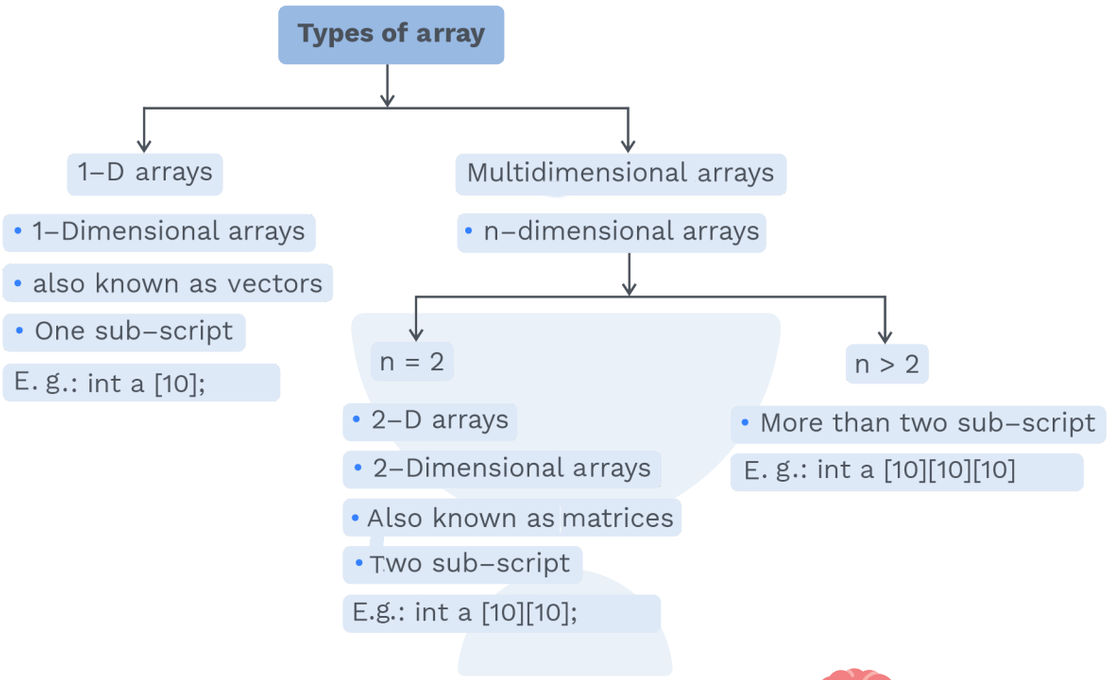
Note:

Programming languages that allow n–base indexing,also allow negative base index values.



- One disadvantage of arrays is that they have fixed sizes. Once an array is

declared, its size cannot be changed. This is due to static memory allocation.



Rack Your Brain

What is the purpose of using multi–dimensional arrays?

Properties of array:

1) A one-dimensional array is a linear data structure which is used to store similar types of elements.

2) The two basic operations that access an array are:

i) Extraction operation: It is a function that accepts an array 'a' and an index 'i' and returns an element of the array.

ii) Storing operation: It accepts an array 'a', an index 'i' and an element 'x'.

a[i]	a[i] = x
Extraction operation	Storing operation

3) The smallest element of an array's index is called the lower bound. In C lower bound is always 0.
4) The highest element is called the upper bound and is usually one less than the number of elements.
5) The number of elements in the array is called range.

```
Range = Upper bound – lower bound + 1
```

E.g. An array 'a' whose lower bound is '0' and the upper bound is 99, then the range would be

⇒ 99–0+1
⇒ 100

6) Array elements are stored at subsequent memory locations
7) 2-D arrays are by default stored in row- major order.
8) In C Programming, the lowest index of a 1-D array is always zero, while the upper index is the size of the array minus one.
9) Array name represents the base address of the array.
10) Only constants and literal values can be assigned as a value of a number of elements.

E.g.

```
const int MAX = 100 ;
int a[MAX] ;
int b[100] ;
```

VALID DECLARATION

```
int MAX = 100 ;
int a[MAX] ;
```

INVALID DECLARATION

Different types of array:

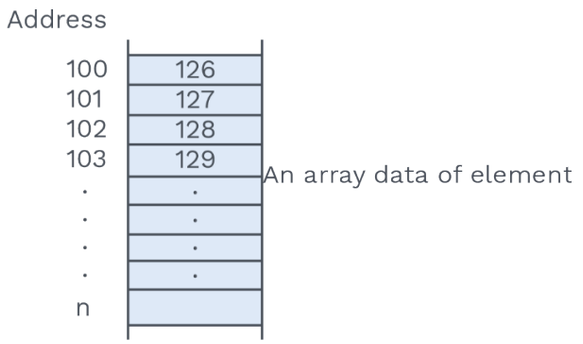
1–D arrays

- One–dimensional array is used when it is necessary to keep a large number of items in memory and reference all the items in a uniform manner.

Syntax: int b[100];

- This declaration reserves 100 successive memory locations, each containing a single integer value.
- The address of the first location is called the base address and denoted by base(b).
- The size of each element be w then address of an element i in an array b is given as:

```
b[i] = base(b) + i *w
```



Rack Your Brain

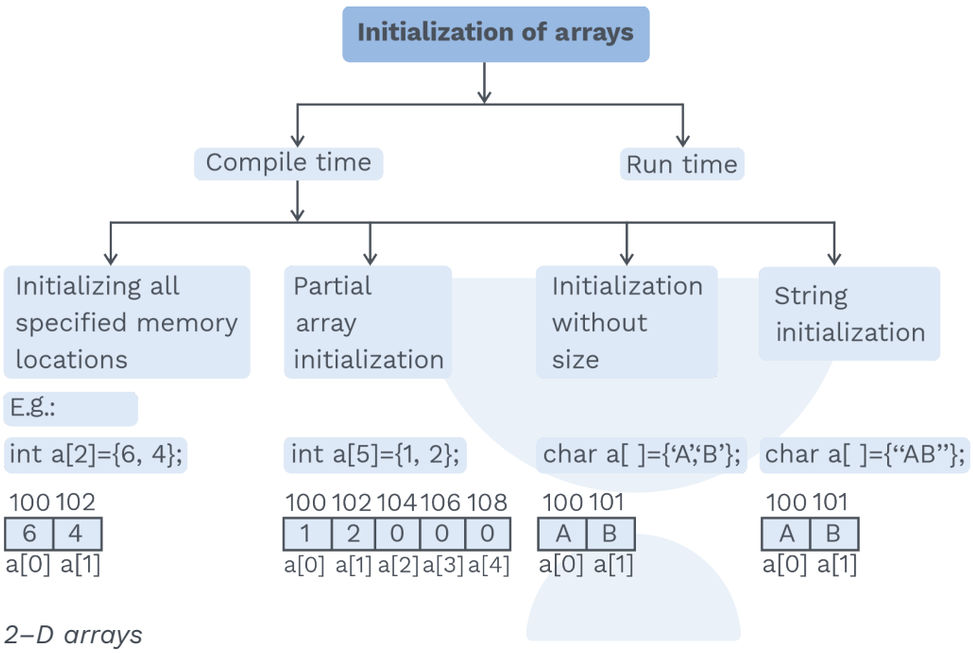
Which of the following is an absolute array declaration in C?

I) int a[3];
II) int a[3] = {1, 2, 3}
III) int a[3] = { };
IV) int a[3] = {0};
V) int a[3] = {[0...1] = 3};
VI) int a[ ] = {[0...1] = 3};
VII) int *a ;

E.g. Program to print the average of 100 integers and how much each deviates from the average.

```
#include <stdio.h>
#define Num 100
average( )
{    int n[Num];
     int i ; int total = 0;
     float avg, diff;
     for (i = 0; i < Num ; i++)
{    scanf(" %d", & n[i]);
     total + = n[i];
}
     avg = total/Num ;
     printf("Number difference");
     for (i = 0; i < Num ; i++)
{
     diff = n[i] – avg;
     printf ("\n %d %f", n[i], diff);
}
```

```
printf ("\n average is: %f", avg)};
main()
{    average();
}
```



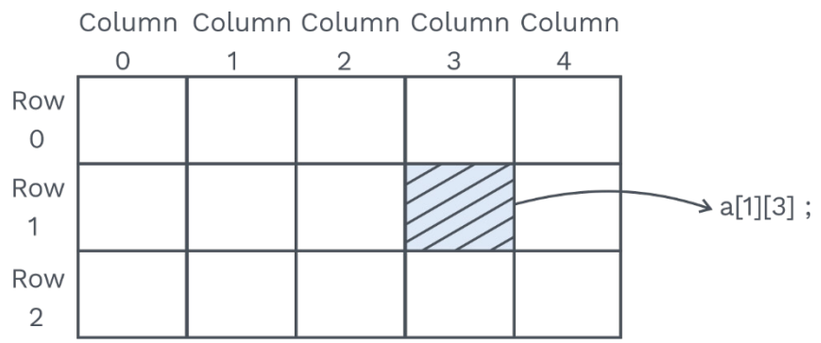
2–D arrays

- A 2–dimensional array is a logical data structure helpful in solving problems.

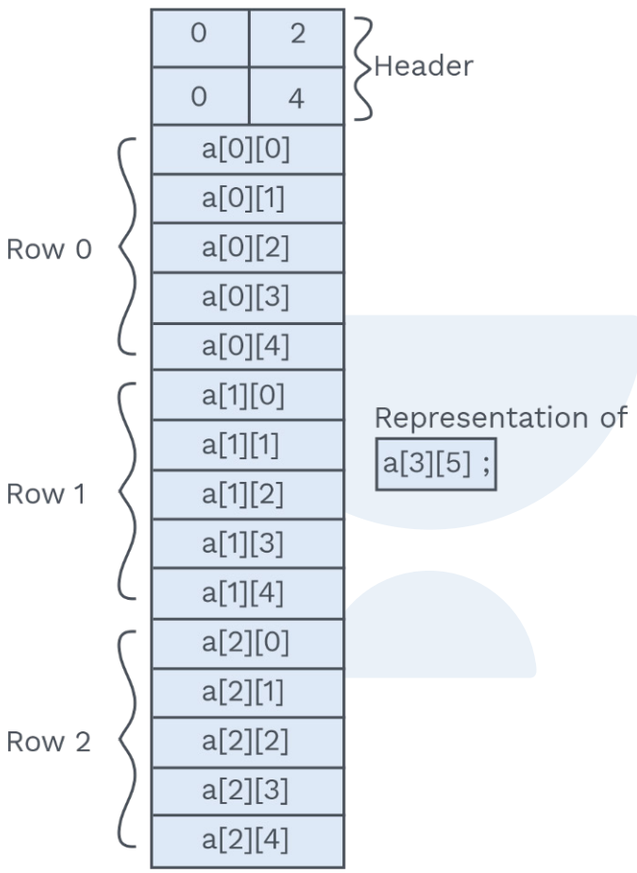
Syntax: int a[3][5];

here [3] is the number of rows, and [5] is the number of columns.

- This represents a new array containing 3 elements. Each of these elements in itself is an array containing five integers.
- The 2–D array has 2–indices called row and column.
- The number of rows or columns is called the range of the dimension.



- The arrays are, by default stored in row- major order. i.e. the first row of the array occupies the first set of memory locations, the second row occupies the next set so on and so forth.



Accessing element a[i][j] from a 2–D array a[m][n] then,
(Here, starting index of array is 0)
Row–major order:

a[i][j] = base(a) + (i*n + j) *w

Column–major order:

a[i][j] = base(a) + (i + j * m) *w

Where:

m : no. of rows

n : no. of columns

w : element size

base (a): base address of array a[m][n]

Initialization of 2D arrays

- One can initialize a 2–D array in the form of a matrix.

E.g. int a[2][3] = { {0, 0, 0},
{1, 1, 1} };

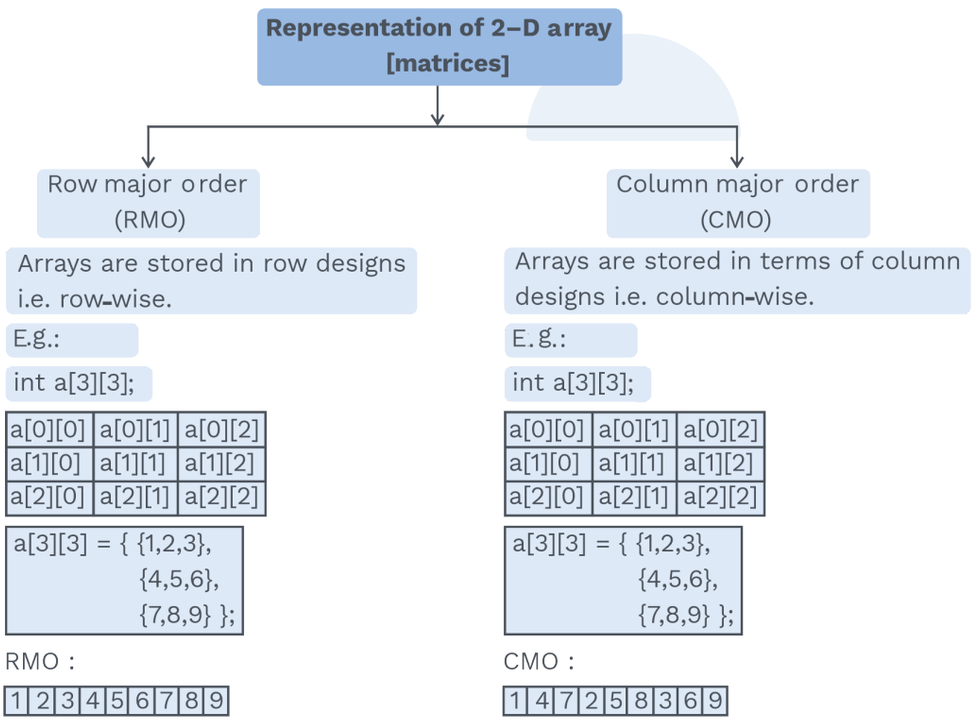
- When the array is completely initialized with all the values, then we do not need to specify the size of the first dimension.

E.g. int a[ ][3] = { {0, 0, 3},
{1, 1, 1} };

Note:

1) If the values are missing in an initialization, then they are by default considered as 0.

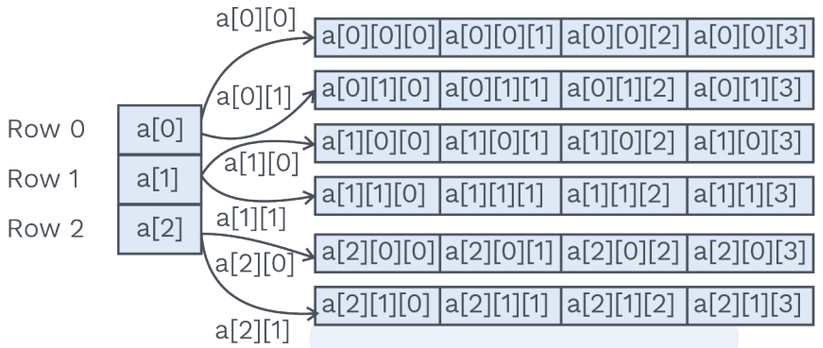
2) The first dimension is optional but the secong dimension must be specified. i.e. rows are optional in a 2-D array, but initialization of columns is mandatory in 2–D array initialization.



Multi–dimensional arrays

- Arrays with more than two dimensions

Syntax: int a[3][2][4] ;// 3–D array



- The first subscript specifies the plane number, the second specifies the row number, and the third specifies the column number.

int a[p][m][n] ;

p : Plane number

m : row number

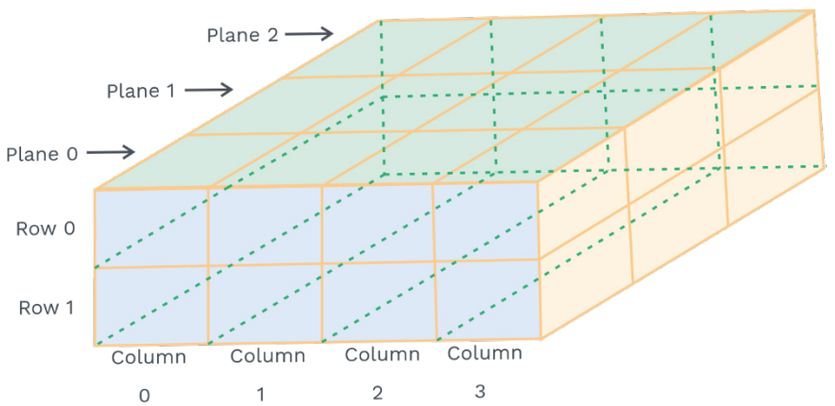
n : column number

E.g. a[3][2][4] ;

no. of plane = 3

no. of rows = 2

no. of columns = 4



Note:

C does allow an arbitrary number of dimensions.

E.g.: There can be a 6–dimensional array which has six subscripts.

2.2 ACCESS ELEMENT

1–D array:

Number of elements = Upper bound – Lower bound + 1

- Upper bound: Index of last element of array
- Lower bound: Index of first element of array

- Let the base address of an array a be b and the lower bound be represented as L.B, and this size of each element be w.
- Then the address of K^{th} element of an array a is given as:

$$a[K] = b + [K - L.B] * w$$

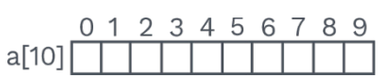
- If array indexing starts from 0 then:

$$a[K] = b + K*w$$

SOLVED EXAMPLES

Q1 **Let the base address of the first element of the array be 200 and each element occupy 4 bytes in the memory, then the address of the fifth element of a 1-D array a [10]?**

Sol: Given,
base address (b) = 200
size of each element (w) = 4 bytes



since L.B not given, it is by default 0. Here the fifth element refers to the element at array index 4
fifth element = a[4]
∴ K = 4
a[K] = b+ K *w
a[4] = 200 + 4 * 4
a[4] = 216

Q2 **Consider an array declaration as A[–7....7] where each element is of 4 bytes; if the base address of the array is 3000, then find the address of A[0]?**

Sol: Given,

array : A[–7....7]

W = 4 bytes

b = 3000

no. of elements = U.B – L.B + 1

= 7 – (–7) + 1

= 14 + 1

= 15

∴ A[K] = b+[(K–L.B)] *w

A[0] = 3000 + [(0–(–7))] *4

= 3000 + 28

A[0] = 3028

OR

Since: A[–7....7] = A[0.....14]

∴ A[0] = A[0–(–7)]

A[0] = A[7]

∴ A[7] = b + K*w

= 3000 + 7*4

A[7] = 3028

Previous Years' Question (GATE 2005: 2-Mark)

A program P reads in 500 integers in the range [0.....100] representing the scores of 500 students. It then prints the frequency of each score above 50. What would be the best way for P to store the frequencies?

a) An array of 50 numbers

b) An array of 100 numbers

c) An array of 500 numbers

d) A dynamically allocated array of 550 numbers

Sol: a)

Rack Your Brain

1) Let the base address of the first element of an array be 100, and each element occupy 2 bytes in the memory, then what is the address of a[5].

2) Let an array be a[-12.....10] where each element occupies 3 bytes, then if the base address of the array a is 2000 then, what is the address of element a[7]?

2–D array:
Two ways of array implementation

Row major order
Let an array A[m][n], then address of element A[i][j] in row major order is given by:

$$A[i][j] = b + (i*n + j) *w$$

when: A[0.....m–1] [0.....n–1]

$$A[i][j] = b + [(i–1)*n + (j–1)] *w$$

when: A[1.....m] [1.....n]
Similarly:
$A[L_1.....U_1]\ [L_2.....U_2]$
then:
$A[i][j] = b+[(U_2–L_2+1)\ (i–L_1) + (j–L_2)] *w$
Where:
L_1: lower bound of rows
U_1: upper bound of rows
L_2: lower bound of columns
U_2: upper bound of columns

SOLVED EXAMPLES

Q3 **Let an array A[5.....15, –8.....8] be A[11][17] stored in row major order. Then if the base address of the array is 500 and each element occupies 3 bytes of memory, what would be the address of A[8][5]?**

Sol: given,
A[11][17] which is A[5.....15, –8.....8]
m = 11 //no. of rows
n = 17 //no. of columns
b = 500, w = 3 bytes
or
$L_1 = 5$, $L_2 = –8$
$U_1 = 15$, $U_2 = 8$

this means ⇒ A[0.....10] [0.....16]

↓ ↓

11 rows 17 columns

i.e. A[8][5] = A[8−(5)] [5−(−8)]

≈ A[3][13]

A[i][j] = b + (i*n + j) *w

A[3][13] = 500 + (3 * 17 + 13) * 3

= 500 + 192

A[3][13] = 692

or

A[8][5] = b + [(U_2−L_2 + 1) (i−L_1) + (j−L_2)] *w

= 500 + [(8−(−8)+1) (8−5) + (5−(−8))] *3

= 500 + [17 × 3 + 13] *3

= 500 + 192

A[8][5] = 692

Q4 **Consider a 2−D array Arr of range [−5.....5, 3.....13] where each element occupies 4 four memory cells, and the base address of the array is 200. Then what would be the address of location arr[5][6]?**

Sol: Given,

Arr[−5.....5, 3.....13] , b = 2000

Arr[0.....10, 0.....10] , w = 4bytes

↓ ↓

11 rows 11 columns

1st Approach:

Arr[−5.....5, 3.....13]

L_1 U_1 L_2 U_2

A[i][j] = b + [(U_2−L_2+1) (i−L_1) + (j−L_2)] *w

A[5][6] = 2000 + [(13−3+1) (5−1(−5)) + (6−3)] *4

= 2000 + [(10×11)+3] *4

= 2000 + 452

A[5][6] = 2452

2nd Approach:

Arr[0.....10, 0.....10]

A[5][6] ≈ [5–1(–5)] [6–3]
≈ A[10] [3]
A[10][3] = b + (i*n + j) *w
= 2000 + ((10×11)+3) *4
= 2000 + 452
A[10][3] = 2452

Previous Years' Question (GATE 2000: 1-Mark)

An n x n array v is defined as follows :
v[i, j] = i–j for all i, j, $1 \le i \le n$, $1 \le j \le n$
The sum of elements of the array v is
a) 0 **b)** n–1 **c)** n^2-3n+2 **d)** $n^2(n+1)/2$
Sol: a)

Previous Years' Question (GATE 2014 : 1-Mark)

Let A be a square matrix of size n x n. Consider the following program. What is the expected output?

```
C=100 ;
for i = 1 to n do
for j = 1 to n do
{
    Temp = A[i][j] + C ;
    A[i][j] = A[j][i] ;
    A[j][i] = Temp – C ;
}
for i = 1 to n do
for j = 1 to n do
    output (A[i][j]) ;
```

a) The matrix A itself
d) Transpose of matrix A
c) Adding 100 to the upper diagonal elements and subtracting 100 from lower diagonal elements of A
d) None of the above
Sol: a)

Previous Years' Question (GATE 2015 Set-2 : 2-Marks)

A Young tableau is a 2–D array of integers increasing from left to right and from top to bottom. Any unfilled entries are marked with ∞, and hence there cannot be any entry to the right of, or below a ∞. The following Young tableau consists of unique entries.

1	2	5	14
3	4	6	23
10	12	18	25
31	∞	∞	∞

When an element is removed from a Young tableau, other elements should be moved into its place so that the resulting table is still a Young tableau (unfilled entries may be filled with a ∞). The minimum number of entries (other than 1) to be shifted, to remove one from the given young tableau is ..... .

Sol: 5

Previous Years' Question (GATE 1998 : 2-Marks)

Let A be a two–dimensional array declared as follows:
A : array [1....10] [1....15] of integer;
Assuming that each integer takes one memory location. The array is sorted in row-major order and first element of the array is stored at location 100. What is the address of the element A[i][j]?

a) 15i + j + 84 **b)** 15j + i + 84 **c)** 10i + j + 89 **d)** 10j + i + 89

Sol: a)

Column–major order

Let any array A[m][n] then address of element A[i][j] in column major order is given by:

$$A[i][j] = b + (i + j * m) * w$$

when: A[0.....m–1] [0.....n–1]

$$A[i][j] = b + ((i-1) + (j-1) * m) * w$$

when: A[1....m] [1.....n]

Similarly: A[L_1.....U_1] [L_2.....U_2]
then:

$$A[i][j] = b + [(i - L_1) + (j - L_2)(U_1 - L_1 + 1)] * w$$

where:
L_1: Lower bound of rows
U_1: Upper bound of rows
L_2: Lower bound of columns
U_2: Upper bond of columns

Rack Your Brain

1) ..... is the logical and mathematical model of a particular organization of data.
(a) Structure (b) Variables (c) Function (d) Data structures

2) Which of the following is not a primitive data structure?
a) Boolean **b)** Integer **c)** Arrays **d)** Character

SOLVED EXAMPLES

Q5 **An array A[20][30] is stored column-wise with, each element occupying 4 bytes of memory space. If the base address is 100,then what is the memory location of the element A[5][15]?**

Sol: Given,
A[20][30] stored in column major order
w = 4 bytes
b = 100
A[5][15] = ?
A[i][j] = b+ (i + j * m) *w
A[5][15] = 100 + (5 + 15×20) *4
A[5][15] = 1320

Q6 **An array A[–2....8] [–2....5] is stored in the memory in column major order such that each element occupies 6 bytes by memory. What would be the memory locations of A[3][2]? Consider the base address to be 5000.**

Sol: Given,

A[–2....8] [–2.....5]

b = 5000

$A[i][j] = b+[(i-L_1) + (j-L_2)(U_1-L_1+1)] *w$

$A[3][2] = 5000 + [(3-(-2)) + (2-(-2))(8-(-2)+1)] *6$

$= 5000 + [5 + 44] *6$

$= 5000 + 294$

A[3][2] = 5294

or

Convert A[–2....8] [–2.....5] to 0 indexing.

i.e. A[0....10] [0.....7] then A[3][2] also

maps to A[5][4] , m = 11, n = 8

$\therefore A[5][4] = b+ (i + j *m) *w$

$= 5000 + (5 + 4\times11) * 6$

$= 5000 + 294$

A[5][4] = 5294

Multi–dimensional arrays:

- In multi–dimensional structure the terms row and columns cannot be used, since there are more than 2 dimensions.

$\therefore$ Let a N–Dimensional array:

$A[L_1.....U_1][L_2.....U_2][L_3.....U_3][L_4.....U_4].....[L_N.....U_N]$

then location of A[i, j, k .....x] is given by:

$= b + \{(i-L_1)[(U_2-L_2+1)(U_3-L_3+1)(U_4-L_4+1).....(U_N-L_N+1)]$

$+ (j-L_2)[(U_3-L_3+1)(U_2-L_4+1)....(U_N-L_N+1)]$

$+ (k-L_3)[(U_4-L_4+1) ..... (U_N-L_N+1)] ..... + (x-L_N)\} *w$

3–D Array

Let a 3–D array $A[L_1....U_1][L_2.....U_2][L_3.....U_3]$

then location of A[i][j][k] is given by:

$A[i][j][k] = b + \{(i-L_1)[(U_2-L_2+1)(U_1-L_1+1)]$

$+ (j-L_2)[(U_1-L_1+1)]$

$+ (k-L_3)\} *w$

SOLVED EXAMPLES

Q7 **Suppose a 3–D array A[2...8, –4....1, 6....10] has elements each occupying 4 bytes of memory, if the base address of the array is 100,then what would be the address of element at A[6][2][5]?**

Sol: Given,

A[2.....8, –4.....1, 6.....10], b = 100, w = 4 bytes

L_1 U_1 L_2 U_2 L_3 U_3

then:

$A[i][j][k] = b + \{(i-L_1)\ [(U_2-L_2+1)\ (U_1-L_1+1)]$
$+ (j-L_2)\ [(U_1-L_1+1)]$
$+ (k-L_3)\} * w$
$A[6][2][5] = 100 + \{(6-(2))\ [(1-(-4)+1)\ (8-2+1)]$
$+ (2-(-4))\ [(8-2+1)]$
$+ (5-6)\} * 4$
$= 100 + \{(4*6*7) + (6*7) + (-1)\} * 4$
$= 100 + 836$
$A[6][2][5] = 936$

Sparse matrices:

- Matrices with relatively high proportions of entries with zero are called sparse matrices.
- There are two general n–square sparse matrices.

1) Triangular matrix

- For a square 2-D array, if all the elements beneath(above) principal diagonal are zero, then the matrix is called as triangular matrix.
- If all elements beneath the principal diagonal are zero, then the matrix is upper triangular.
- While for a square matrix, if all the elements above the main diagonal are zero then matrix is upper triangular.

2) Tridiagonal Matrix

- For a square matrix, if all the elements are zero except the elements on the principal diagonal and all the elements immediately above and immediately below the principal diagonal, then the matrix is tridiagonal.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 5 & 2 & 0 & 0 \\ 6 & 7 & 3 & 0 \\ 8 & 9 & 10 & 4 \end{bmatrix}_{4\times4}$$

Lower triangular matrix

$$\begin{bmatrix} 1 & 5 & 8 & 10 \\ 0 & 2 & 6 & 9 \\ 0 & 0 & 3 & 7 \\ 0 & 0 & 0 & 4 \end{bmatrix}_{4\times4}$$

Upper triangular matrix

$$\begin{bmatrix} 1 & 5 & 0 & 0 \\ 8 & 2 & 6 & 0 \\ 0 & 9 & 3 & 7 \\ 0 & 0 & 10 & 4 \end{bmatrix}_{4\times4}$$

Tridiagonal matrix

Note:

A strictly lower or upper triangular matrix are those where the diagonal are also zero entries.

Triangular matrices:

1) Lower Triangular Matrix nxn , a[i][j]	
Range	$\frac{n(n+1)}{2}$
Row Major Order (RMO)	$b+\left[(j-1)+\frac{i(i-1)}{2}\right]*w$
Column Major Order (CMO)	$b+\left\{(i-j)+\left[(j-1)n-\frac{(j-1)(j-2)}{2}\right]\right\}*w$
2) Upper Triangular Matrix nxn, a[i][j]	
Range	$\frac{n(n+1)}{2}$
Row Major Order (RMO)	$b+\left\{(j-i)+\left[(i-1)n-\frac{(i-1)(i-2)}{2}\right]\right\}*w$

Column Major Order (CMO)	$b+\left[(i-1)+\frac{j(j-1)}{2}\right]*w$
3) Strictly Lower Triangular Matrix [ ]nxn, a[i][j]	
Range	$\frac{n(n-1)}{2}$
Row Major Order (RMO)	$b+\left[(j-1)+\frac{(i-1)(i-2)}{2}\right]*w$
Column Major Order (CMO)	$b+\left[(i-1)+\frac{(j-1)(j-2)}{2}\right]*w$
Tridiagonal matrix $[\ ]_{nxn}$, a[i][j]	
Range	3n–2
Row Major Order (RMO)	b+ (2i+j–3) *w
Column Major Order (CMO)	b+ (i+2j–3) *w

Rack Your Brain

1) Let an array A[1.....6] [1.....10] be a 2–D array. The first element of the array is stored at location 100. Let each of the element occupy 8 bytes. Find the memory location of the element in the second row and fourth column when the array, (I) is stored in RMO and (ii) stored in CMO.

Hint : Find location of A[2][4]

2) Consider the following multi–dimensional array :
x[–5 : 5, 3 : 33], y[3 : 10, 1 : 15, 10 : 20]

a) What would be the number of elements in x and y.

Hint : Range = (U.B –L.B + 1)

b) Let the base address of array y be 400, and each element occupies 4 bytes of memory location. Than what is the address of memory location y[5, 10, 15] when y is stored in row major order and column major order.

Chapter Summary

- Data structure are the logical or mathematical model of a particular organization of data.
- Data structures are chosen such that they can represent the actual relationship of data in real world, while being simple enough that the data can be processed effectively.
- Arrays are simplest type of data structure.
- Arrays are of three types :
  1) Linear or 1–D arrays
  2) 2–D array
  3) n–D array or multidimensional arrays.
- A linear array or one-dimensional array is a list of finite number of similar data elements represented by a set of n consecutive numbers.
- An array A can be represented as :
  1) Subscript notation
  $A_1, A_2, A_3 \ldots\ldots A_n$
  2) Parenthesis notation
  A(1), A(2), A(3) ..... A(n)
  3) Bracket notation
  A[1], A[2], A[3] .....A[n]
  Here : the number N in A[N] is the subscript.
- Arrays are by default stored in Row major order.
- The address of K^{th} element of 1–D array a is given as:
  a[K] = b + [K – L.B] *w
- The address of the element at i^{th} row and j^{th} column of a 2–D array is given as:
  RMO : $A[i][j] = b + [(U_2 - L_2 + 1)(i - L_1) + (j - L_2)] * w$
  CMO : $A[i][j] = b + [(i - L_1) + (j - L_2)(U_1 - L_1 + 1)] * w$
- The address of the element at i^{th}, j^{th} and K^{th} dimension of a 3–D array is given as:
  $A[i][j][K] = b + \{(i - L_1)[(U_2 - L_2 + 1)(U_1 - L_1 + 1)] + (j - L_2)[(U_1 - L_1 + 1)] + (K - L_3)\} * w$

3 Stack & Queues

3.1 BASICS OF STACK

- A stack is a linear data structure in which the items can be added or removed only at one end is called top of stack.
- Stacks are also called last in first out (LIFO).
- This means that elements are removed from the stack in reverse order of the way they were inserted.
- Two basic operations associated with stacks:
  - **a)** PUSH or Push: Insert an element into the stack.
  - **b)** POP or Pop: Delete an element from the stack.

Some of the stack representations can be given as:

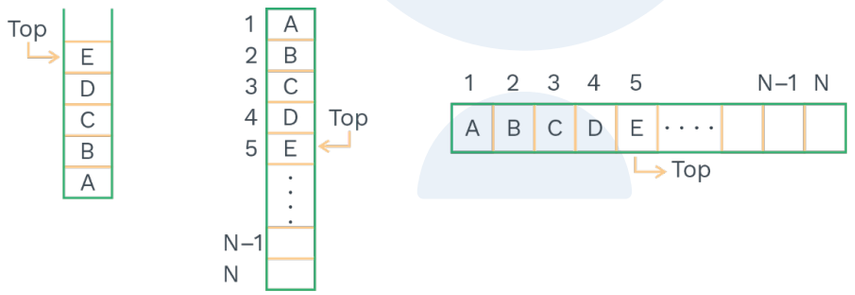


Fig. 3.1

Consider the above shown representation that 5 elements (A, B, C, D, E) are pushed onto an empty stack. The only implication is that rightmost element, i.e. E is the top element. This is emphasised regardless of the way the stack is described since it is the property of stack that insertion and deletion can only occur at the top of stack. This means that D cannot be deleted before E.

Grey Matter Alert!

- LIFO (Last in first out) is also referred to as FILO (First In last out)
- Some other names for stacks are "Piles" or "Push-down lists"

Note:

Stacks can either be used to postpone certain decisions or the order of the processing of data can be postponed until certain conditions get true.

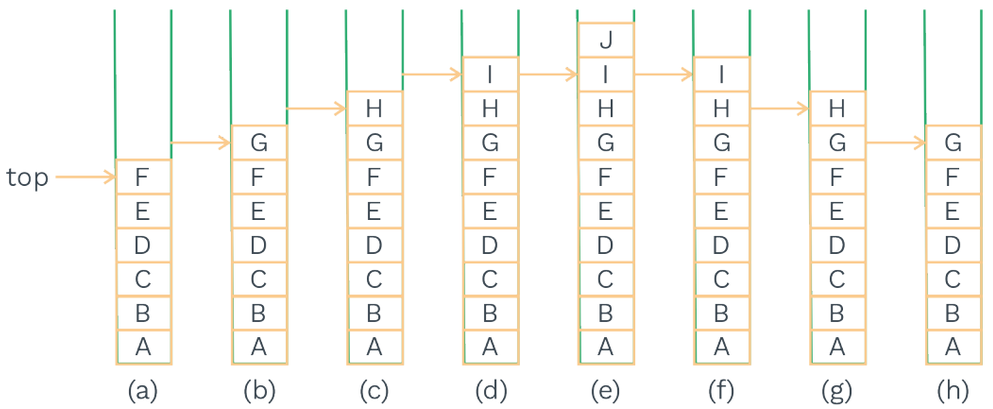


Fig. 3.2

In the above figure, F is the top of the stack. Any elements inserted after F are placed on top of F. The arrow with the word 'top' represents the top of stack.

a) represent the stack with F as the top.
b) Push (G), G is pushed on top of F.
c) Push (H), H is pushed on top of G.
d) and (e) similarly, I and J are pushed onto the stack, respectively.
f) POP (J), the element J is deleted from top of the stack.
g) and (h), POP (I), and POP (H) from the stack, respectively.

3.2 OPERATIONS ON STACK

The basic operations used for manipulation of stack data structure is:

a) PUSH: To insert an item into a stack
b) POP: To delete an item from a stack

Another operation used to know the top element of the stack is PEEK(). Therefore when an item is added to a stack, it is pushed onto the stack, and when an item is removed, it is popped out from the stack.
Consider a stack S and an item a then:

Operation	Meaning
PUSH (s, a)	Push the item a onto the top of the stack s
POP(s)	Removes the item from the top of the stack.
x=POP(s)	Assigns the value of the top of the stack to x and removes it from the stack s.
PEEK(s)	Return the top of the stack.

Table 3.1 Stack Operations

3.3 STACK CONDITIONS

- If a stack has no item, this stack is called an empty stack.
- However, PUSH operation can be performed on an empty stack but a POP operation cannot be applied.
- Hence before applying a POP operation, one needs to make sure that the stack is not empty.
- The two stack conditions to be checked before performing an operation on stack are:

a) Underflow condition:

When a POP operation is tried on an empty stack, it is called underflow condition. This condition should be false for successful POP operation and PEEK operation.

Note:

Due to the push operation, the stack is sometimes referred to as push down lists.

Note:

Empty determines whether stack s is empty or not. If the value returned by the operation is true then the stack is empty.

Operation	Return value	Meaning
empty(s)	TRUE	Stack s is empty
empty(s)	FALSE	Stack s is not empty

b) Overflow condition:

An overflow condition checks whether the stack is full or not, i.e. whether memory space is available to push new elements onto the stack. It is an undesirable condition where the program requests more memory space than what is allocated to a particular stack.

Underflow condition: (empty (s) == TRUE)
Or
(TOP == -1)
Overflow condition: TOP == sizeof(s)

Where s is the stack on which PUSH and POP operations are to be performed.
Here TOP is a special pointer which always points to the top of the stack and plays a major role in checking the overflow and underflow conditions of the stack.

Push operation on stack:
Let, n be the maximum size of stack and p be the element to be inserted onto the stack s.

```
Push(s, Top, n, p)
{       if(Top = = n – 1)
                { printf("stack overflow");
                        exit(1);
                }
                Top ++;
                S[Top] = p; //element p inserted on top of the stack
}
```

Pop operation of stack:
Let n be the maximum size of stack and p be the variable which will contain the value of top of the stack.

```
Pop(s, Top, n)
{       int p;
                If(Top = = –1)
                        { printf("under flow condition");
                                exit(1);
                        }
                p = s[Top];
                Top - - ;
                return(p);
}
```

Application of stack:
Some of the most popular applications of stacks are:

a) Expressions evaluation
e) Fibonacci series
b) Balanced parenthesis check
f) Permutation
c) Recursion
g) Subroutines
d) Tower of hanoi

Balanced parentheses:

- For checking balanced parentheses for every open brace, there should be a closing brace.

Rack Your Brain

Which data structure does the UNDO function of the test editors use?
How does the compiler work for recursions?

- The types of parenthesis used are:
  a) Simple or common bracket ( )
  b) Square bracket [ ]
  c) Curly bracket { }

Example: {{} [] ()} Balanced parenthesis

(((((() ()))))(()))

{[](){}(()) Imbalanced parenthesis

[[][](()

1) Push the open parenthesis onto the stack
2) Whenever a close parenthesis is encountered then pop the top of the stack which should be the matching parenthesis.

Note:

The two conditions that must hold if the parenthesis in an expression forms an admissible pattern:

1) The parenthesis count at the end of the expression should be 0. This implies that there are as many left parentheses as right parentheses.
2) The parenthesis count should be non-negative at each point in the expression.

Also, nesting depth at a particular point in an expression is the number off scopes that have been
Considered, (((()())(()))

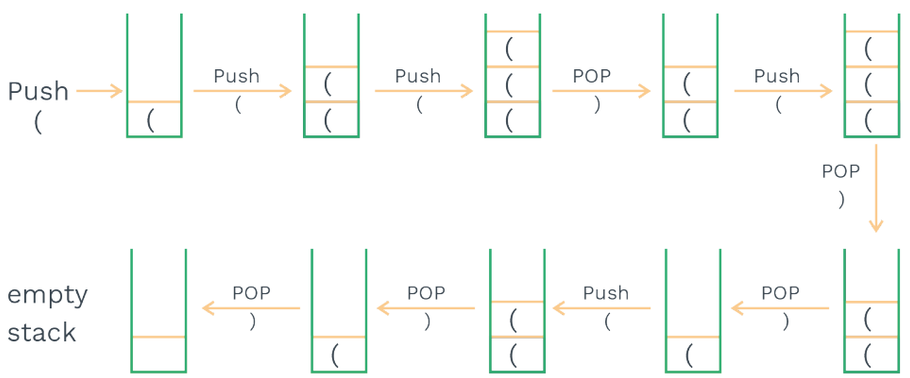


Fig. 3.3

The state of stack at various stages of processing the expression for balanced parenthesis

$\{a + (b - [c + d]) * e - [(f + g)]\}$

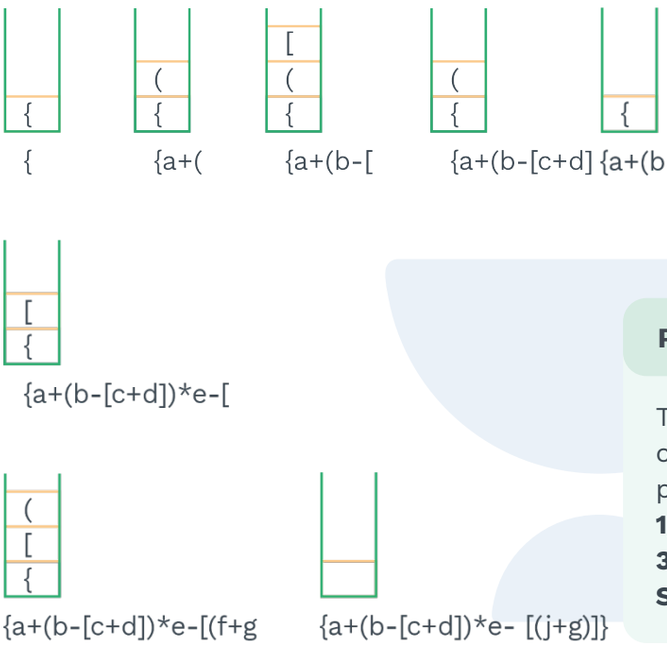


Fig. 3.4

Previous Year's Question

The best data structure to check whether on arithmetic expression has balanced parentheses is a,

1) Queue **2)** Tree
3) Stack **4)** List
Sol: Option 3) **(GATE- 2004)**

3.4 EXPRESSION EVALUATION

There are three notations for an expression representation:

a) Infix notation: In this type of notation the operator lies between the operands. Example: sum of A and B can be given as A+B

b) Prefix notation: In this type of Notation the operators come before the operands.
Example: Sum of A and B can be given as +AB

c) Postfix notation: In this type of notation the operator comes after the operands.
Example: Sum of A and B can be given as AB+

There are five binary operations:

a) Addition (+)
b) Subtraction (-)
c) Multiplication (*)
d) Division (/)
e) Exponentiation ($)

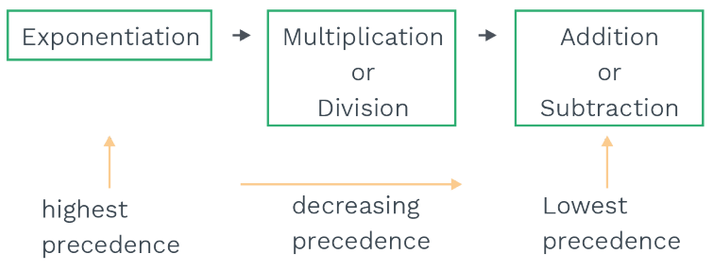


Fig. 3.5

Example:

Infix	Prefix	Postfix
A+B*C	+A*BC	ABC*+
A+B	+AB	AB+
A-B/(C*D$E)	-A/B*C$DE	ABCDE$*/–
(A+B) * (C+D)	*+AB + CD	AB + CD +*

Table 3.2 Infix vs Prefix vs Postfix

Note:

Prefix notation is also known as polish notation postfix notation is also known as reverse polish notation.

The expression sum of two variables A and B can be given as A + B where A and B are the operands, and '+' is the operator.

- While expression evaluation, the precedence of operators plays a vital role.
- Parenthesis are of great help in getting the order of evaluation.

Example:

A + (B * C)	(A + B) * C
• Here, parenthesis emphasises that multiplication is converted first and then addition.	• Here, parenthesis emphasises that addition is converted first and then multiplication.

Table 3.3

When unparenthesized operators having the same precedence are scanned then the order by default is considered from left to right.

- But in the case of exponentiation, the default order of evaluation is considered from right to left.

Example:

A + B + C	A $ B $ C
(Left to Right)	(Right to Left)
(A + B) + C	A $ (B $ C)

Note:

Default precedence can be overridden by using parenthesis.

Expression conversion:

- The prefixes 'pre-', "post-", and "in-" refer to the relative position of the operator w.r.t. the operand.
- The only rules to always remember while expression conversion to any form is that operations with higher precedence are converted first and after a portion of the expression is converted to a form, then it is considered as a single operand.

Example: A + B * C (Infix)

A + (BC*) (Postfix)
↓
Treat as single operand

A + (*BC) (Prefix)
↓
Treat as single operand

Note:

Rules of expression conversion:

1) The operations of higher precedence are converted first.

2) Once an operation in converted to postfix or prefix form it is considered as a single operand.

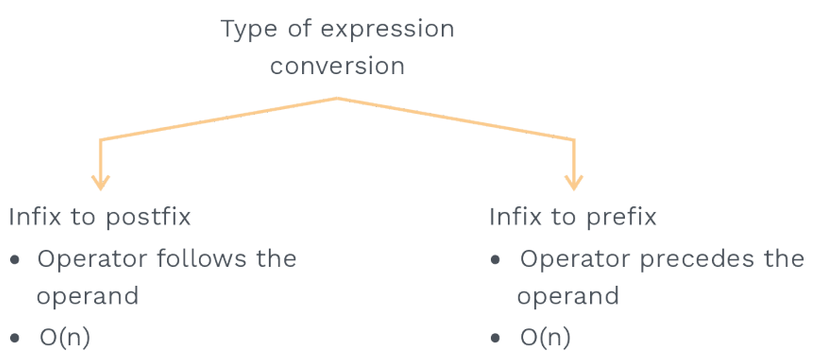


Fig. 3.6

Note:

The order of the operators in the postfix expression determines the actual order of operations in evaluating the expression, hence making the use of parenthesis in postfix notation is unnecessary.

Infix	Postfix
A + (B * C)	ABC * +
(A + B) * C	AB + C *

Grey Matter Alert!

The prefix form of an expression is not the mirror image of the postfix form.

Example:

Infix to postfix conversion:

1) Check for parenthesis;, the operations enclosed within parenthesis are to be converted first.

2) The other operations in the expression are converted according to the operator precedence.

Example: A + (B * C)

1) Parentheses (B * C) are converted first.

A + (BC *)

2) Now (BC *) is treated as a single operand, and then the entire expression is converted.

3) The postfix expression obtained: ABC * +

Infix to postfix conversion using stack:

An algorithm than can be designed to convert an infix expression to postfix express is given as:

```
Operator_stack = empty stack;
while(!End of input)

    {       symbol = next input character;
                if (symbol == operand)
                add symbol to the postfix string;
                else
                { while (!empty (operator_stack) & precedence (Top
```

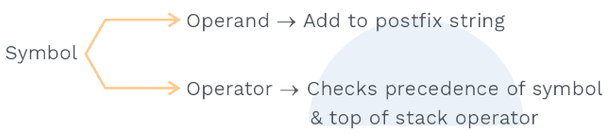
```
(operator_stack), symbol))
                    {       temp = pop (operator_stack);
                            add temp to the postfix string;
                    } //end of while
                Push (operator_stack, symbol);
            } //end of else
} //end of while
while(!empty(operator stack))
                {       temp = pop (operator_stack);
                        Add temp to the postfix string;
                }       //output all remaining operators
```

The algorithm given can be understood in step as:

1) The operator-stack is the stack which has been initialized, to push operators from the input string.

2) Symbol is the variable which reads the next input symbol of the infix expression (input string).

3) The if condition checks whether the input symbol is an operand or operator.



4) If the precedence of the operator on top of the stack is greater, then the operator in the input string (symbol) then pops the top of the stack and add to postfix expression, then PUSH the symbol onto the stack. Otherwise, simply push the symbol onto the stack.

5) If the input string is completely scanned, then pop all operators from the operator-stack and add to the postfix string.

Example: A + B * C

S.no	Symbol	Postfix string	Operator-stack
1)	A	A	+
2)	+	A	+
3)	B	AB	+
4)	*	AB	+*
5)	C	ABC	+*

S.no	Symbol	Postfix string	Operator-stack
6)		ABC*	+
7)		ABC*+	

Table 3.5

Postfix string: ABC * +

Note:

The operator-stack has operators that work according to right associativity of operators:

+ *
  ↳ Top

Example: ((A – (B + C))/D) $ (E + F)

S. no	Symbol	Postfix string	Operator-stack
1)	(		(
2)	(		((
3)	A	A	((
4)	–	A	((–
5)	(	A	((– (
6)	B	AB	((– (
7)	+	AB	((– (+
8)	C	ABC	((– (+
9)	)	ABC +	((–
10)	)	ABC + –	(
11)	/	ABC + –	(/

S. no	Symbol	Postfix string	Operator-stack
12)	D	ABC + – D	(/
13)	)	ABC + – D/	
14)	$	ABC + – D/	$
15)	(	ABC + – D/	$(
16)	E	ABC + – D/E	$(
17)	+	ABC + – D/E	$(+
18)	F	ABC + – D/EF	$(+
19)	)	ABC + – D/EF +	$
20)		ABC + – D/EF + $	

Table 3.5

Postfix string: ABC + – D/EF + $

Evaluating postfix expression using stack:
The following algorithm evaluates an expression in postfix notation:

```
operand_stack = empty stack;
while (! end of input)
{symbol = next input character;
if (symbol == operand)
                PUSH (operand_stack, symbol);
else {
                operand 2 = pop (operand_stack);
                operand 1 = pop (operand_stack);
                value = result of apply symbol on operand 1 and operand
2.
                /* value = operand 1 symbol operand 2 */
                PUSH (operand_stack, value);
                        }
                } return (pop(operand_stack)));
```

Rack Your Brain

Convert these postfix expressions to infix.

XY – P + QRS – + $

PQRST – + $ * AB * –

The algorithm given for postfix expression evaluation can be understood as:

1) The operand_stack is an empty stack where the operands are pushed into.

2) Symbol is the variable which reads the next input symbol of the postfix expression.

3) The if condition checks whether the input symbol is an operand or an operator.

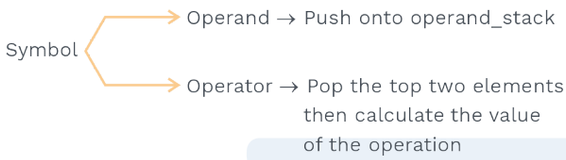


Fig. 3.8

- Keep the value of the top of stack in operand 2 and the next value in the operand 1 then apply operator:
  Value = operand 1 operator operand 2

Example:

Fig. 3.9

Value = 6 + 8

Value = 14

4) Then, push the value onto the operand stack.

5) Once the entire string is traversed or read, then pop the top of the stack which returns the value of the expression.

Example: 26 12 13 + – 3 18 2 / + * 2 $ 3 +

S. no	Symbol	Operand 1	Operand 2	Value	Operand_stack
1)	26				26
2)	12				26 12
3)	13				26 12 13
4)	+	12	13	25	26 25

S. no	Symbol	Operand 1	Operand 2	Value	Operand_stack
5)	–	26	25	1	1
6)	3	26	25	1	1 3
7)	18	26	25	1	1 3 18
8)	2	26	25	1	1 3 18 2
9)	/	18	2	9	1 3 9
10)	+	3	9	12	1 12
11)	*	1	12	12	12
12)	2	1	12	12	12 2
13)	$	12	2	144	144
14)	3	12	2	144	144 3
15)	+	144	3	147	147

Table 3.6

The value of the expression (26 12 13 + – 3 18 2 / + * 2 $ 3 +) is 147.

Infix to prefix conversion:

Some of the steps to follow to convert a complex infix expression into prefix form:

1) Read the expression in reverse from right to left. Convex the open braces as closed and vice versa.

2) Now, the reversed expression obtained is converted into postfix notation using stack.

3) Now, for the final step, reverse the postfix notation i.e. the postfix expression read from right to left results in a prefix notation expression.

Rack Your Brain

Why infix to postfix conversion is required?

Example: (4 + (6 * 2) * 8)

Step I: Reverse the infix expression
(4 + (6 * 2) * 8) ← Reading from right to left
(8 * (2 * 6) + 4)

Step II: Convert (8 * (2 * 6) + 4) into postfix.
Postfix: 826 ** 4 +

Step III: Reverse the postfix notation.
826 ** 4 + ← Read from right to left.
Prefix: +4 ** 628

Evaluation of prefix expression:

- The prefix expression evaluation uses the same steps as the postfix expression evaluation just that the input expression is read from right to left.

Rack Your Brain

Convert the following prefix expression to infix + – $ ABC * D **EFG

Example: +4 * * 628 ← Read

S. no	Symbol	Operand 1	Operand 2	Value	Operand stack
1)	8				8
2)	2				8 2
3)	6				8 2 6
4)	*	2	6	12	8 12
5)	*	8	12	96	96
6)	4	8	12	96	96 4
7)	+	96	4	100	100

Table 3.7

The value of prefix expression (+4 * * 628) is 100.

Previous Years' Question

The following postfix expression with single digit operands are evaluated using a stack: 8 2 3 ^ / 2 3 * + 5 1 * - Note that ^ is the exponentiation operator. The top two elements of the stack after the first * is evaluated are:

1) 6, 1 **2)** 5, 7 **3)** 3, 2 **4)** 1, 5

Sol: 1) **(GATE CSE- 2007)**

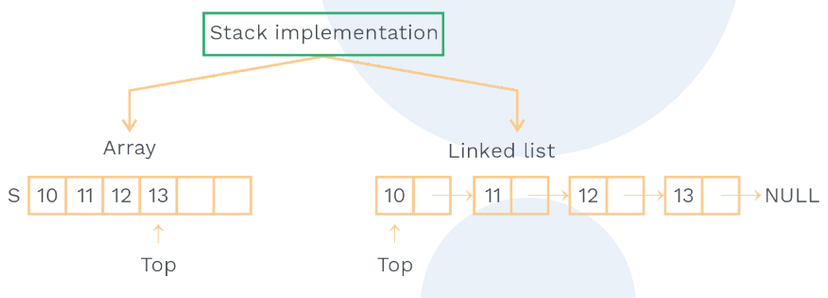
Previous Years' Question

Assume that the operators +, –, ×, are left associative, and ^ is right associative. The order of precedence (from highest to lowest) is ^, ×, +, –. The postfix expression corresponding to the infix expression a + b×c–d^e^f is

1) abc×+def^^-
2) abc×+de^f^-
3) ab+c×d–e^f^
4) – + a×bc^^def

Sol: 1) **(GATE CSE- 2004)**

3.5 IMPLEMENTATION OF STACK



PUSH (S, Top, x)

1) Top = Top + 1;
2) S[Top] = x;

1) Temp = S[Top];
2) Top = Top – 1;
3) Return temp;

- Push & pop both take 0(1) time.

Overflow condition

⇒ (Top == size of array)

Underflow condition

⇒ (Top == –1)

PUSH (Top, x)

1) Temp = x;
2) Temp → next = Top;
3) Top = temp;

POP(S) POP (Top)

1) Temp = Top;
2) Top = Top → Next;
3) a = temp → info;
4) Free (temp);
5) Return (a);

- Push & pop both take 0(1) time.

Overflow condition

⇒ (AVAIL == NULL) linked list have dynamic memory allocation

∴ No limit till memory is available.

Underflow condition

⇒ (Top == NULL)

3.6 APPLICATION OF STACK: TOWER OF HANOI

The tower of Hanoi problem is described as follows:
Suppose there are three pegs labeled A, B and C. On peg A a finite number of n disks are placed in order of decreasing diameter. The objective of the problem is to move the disks from peg A to peg C using peg B as an auxiliary. The rules are as follows:

1) Only one disk can be moved at a time.
2) Only the top disk on any peg can be moved to any other peg.
3) At no time can a larger disk be placed on a smaller disk.

The solution to a Tower of Hanoi problem for n = 3.

n = 3: Move top disk from peg A to peg C.
Move top disk from peg A to peg B.
Move top disk from peg C to peg B.
Move top disk from peg A to peg C.
Move top disk from peg B to peg A.
Move top disk from peg B to peg C.
Move top disk from peg A to peg C.

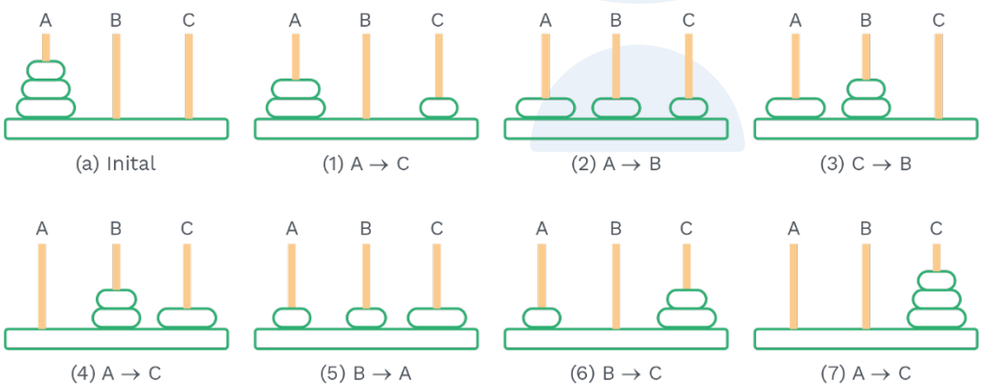


Fig. 3.11 The Steps of Tower's of Hanoi

Tower of hanoi solution:

n	Steps
1)	A → C
2)	A → B, A → C, B → C
3)	A → C, A → B, C → B, A → C, B → A, B → C, A → C,

Generalized Solution for n

1) Move the top (n – 1) disks from peg A to peg B
2) Move the top disk from peg A to peg C
3) Move the top (n – 1) disks from peg B to peg C

Let Peg A referred to as BEG, Peg C as END and Peg B as AUX then:

1) TOH (N – 1, BEG, END, AUX) **2)** TOH (1, BEG, AUX, END)

3) TOH (N – 1, AUX, BEG, END)

The tower of Hanoi (TOH) can be divided into three sub problems and can be solved recursively as:

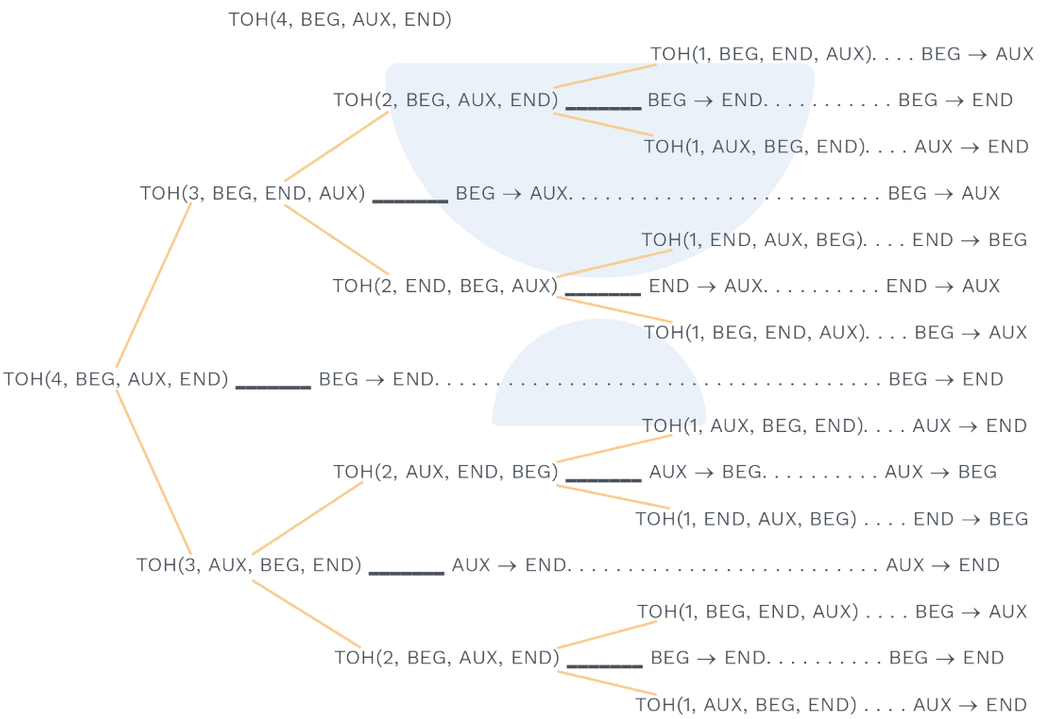
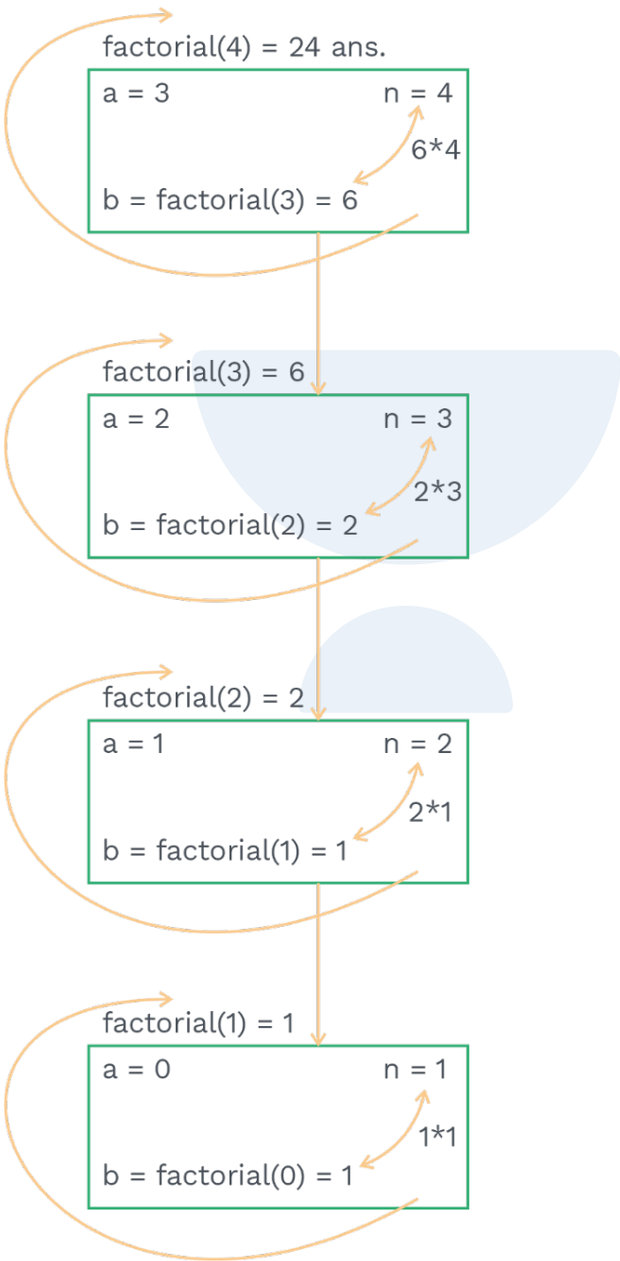


Fig. 3.12 Recursion Tree for Tower's of Hanoi

Factorial:

```
int n;
    factorial(n)
    {    int a, b;
         if(n == 0)
         return(1);
         a = n – 1;
         b = factorial(a);
         return(n * b);
    }
```

Evaluating the above function by taking n = 4, by using recursive tree evaluation of the function code:



Ans. 24

Fig. 3.13

Fibonacci number:

```
int n;
fibonacci(n)

{       int a, b;
            if(n < = 1)
            return(n);
            a = fibonacci(n – 1);
            b = fibonacci(n – 2);
            return(a + b);
}
```

Evaluating the above function by taking n = 4 by using recursive tree evaluation of function code:

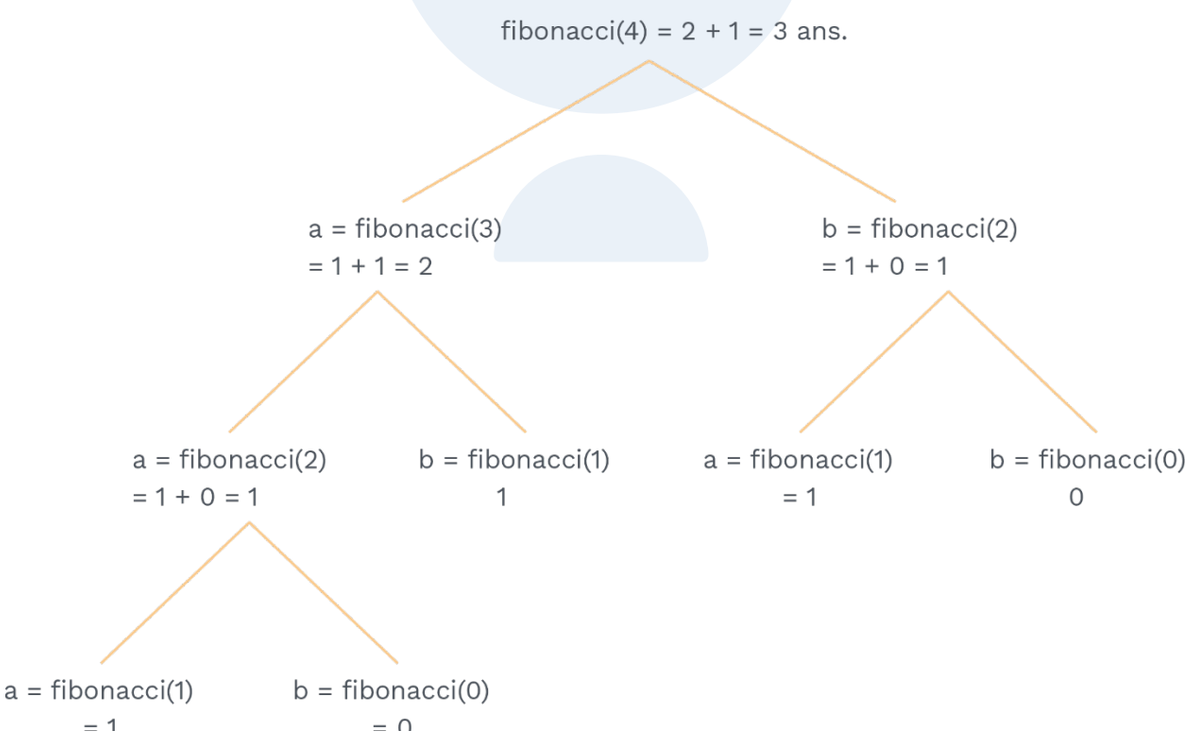


Fig. 3.14 Recursion Tree for Fibonacci(N)

Sol: 3)

3.7 BASICS OF QUEUE

- Queue is a data structure in which we insert and delete data items.
- The fashion in which we insert/delete data items is: 'One side insertion and other side deletion'.
- Queue is represented as:

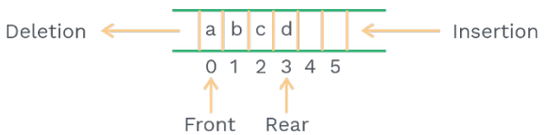


Fig. 3.15

- **Front:** It is a variable that contains position of the element to be deleted.
- **Rear:** It is a variable that contains position of the newly inserted element.
- For the above representation, Let's find the front and rear:
  1) Front is 0 **2)** Rear is 3
- **Property of queue:**

FIFO: First In First Out
OR
LILO: Last In Last Out

In the given queue below, Let's understand FIFO or LILO.
Consider a queue of size '6'.

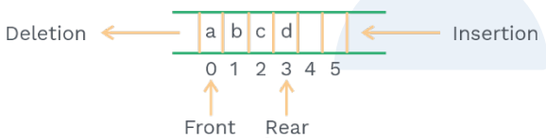


Fig. 3.15

Steps for insertion:

1) First, a is inserted in the queue at position '0'.
2) b is inserted at position 1.
3) c is inserted at position 2.
4) d is inserted at position 3.
Positions 4 and 5 are vacant.

Steps for deletion:

1) The first deletion from queue would be at position 0
2) Second deletion be at position 1
3) Third deletion be at position 2
4) Fourth deletion be at position 3
Clearly, 'a' was inserted first and deleted first and so on.

Note:

If we want to delete b at first from above queue, then it is not possible. Hence, queue works according to the property FIFO or LILO.

Note:

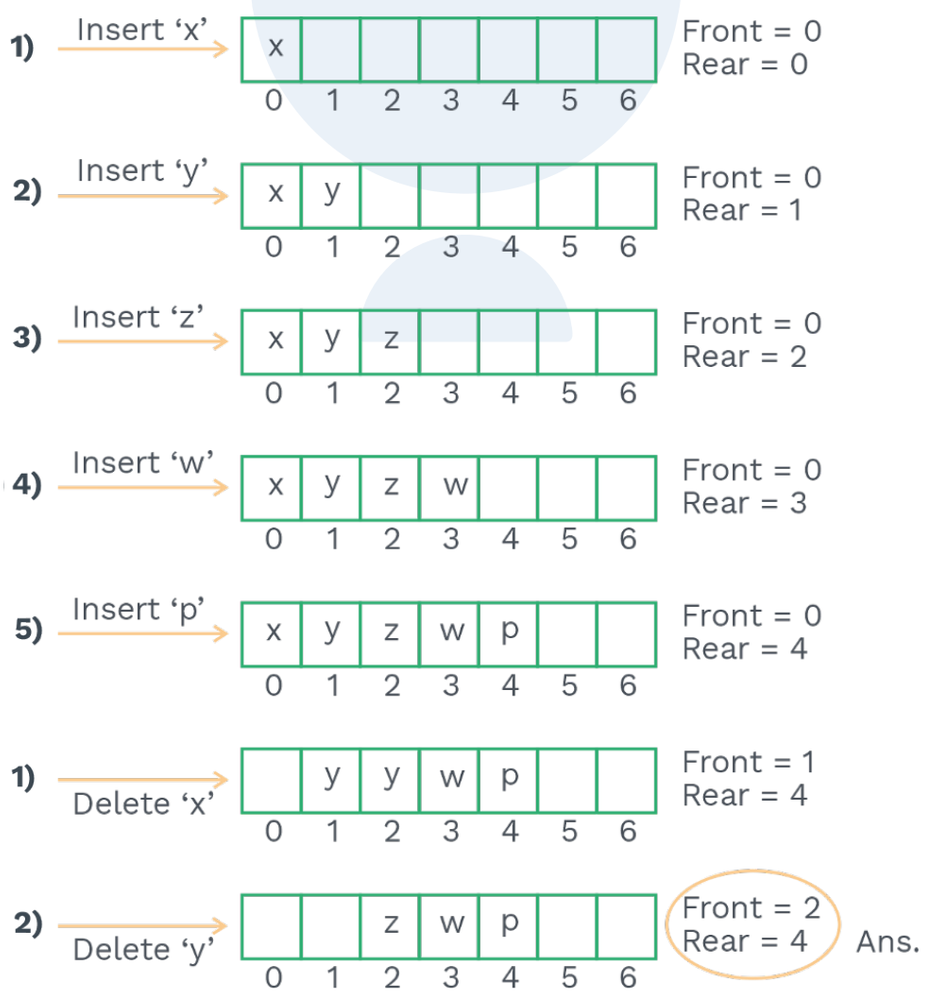
Queue–data structure is used in breadth first search traversal of graphs.

SOLVED EXAMPLES

Q1 **Consider a queue of size '7'. In this queue, five insertions and two deletions have been performed. Find the values of 'Front' and 'Rear'.**

Sol: **Queue size is given as '7'.**

Let's plot the queue and perform 5 insertions and 2 deletions ;



3.8 OPERATIONS ON QUEUE

The operations we can perform on queue is called ADT of queue.

(ADT: Abstract data type)

Rack Your Brain

You are given a queue of size 'n'. It's needed to perform 'n+3' insertions. What would be the result?

- **ADT of queue:**
  1) Enqueue ( )
  2) Dequeue ( )
  We perform two operations on queue, and they are enqueue and dequeue.

1) Enqueue ( ):
- Using enqueue, an element is inserted into the queue.
- To insert 'n' elements there are 'n' enqueue() operations needed.

2) Dequeue ( ):
- Using dequeue an element is deleted from the queue.
- To delete 'n' elements from the queue, 'n' dequeue() operations needed.

For Example:
Consider an empty queue of size 6,

Fig. 3.17

Enqueue **a)**: Will insert 'a' at 0.

Enqueue **b)**: Will insert b at 1 and so on.

Dequeue ( ): Will delete a by default because it was inserted first and so on.

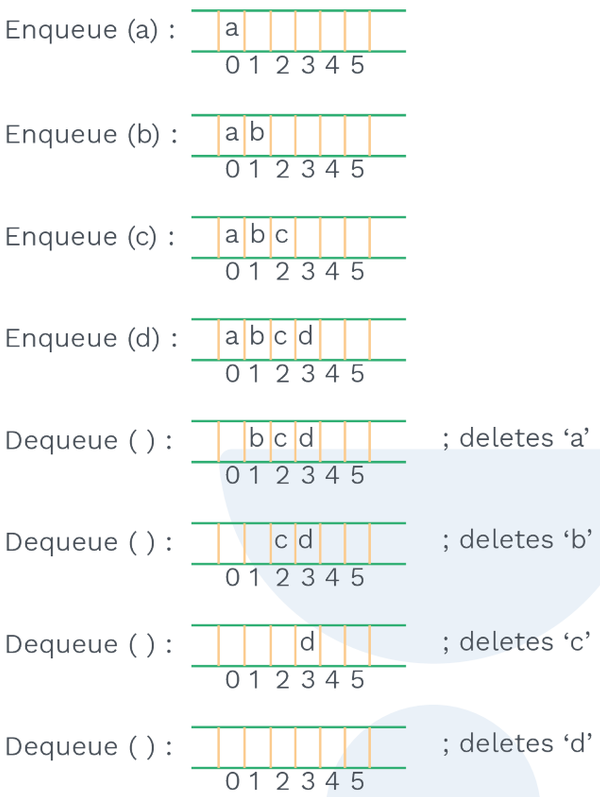


Fig. 3.18

3.9 TYPES OF QUEUE

There are different types of queue; Let's understand them one by one.

Circular queue:

- Linear queue is not that efficient because the rear can't go back once it reaches end of queue.

For example: Suppose a situation as shown in diagram

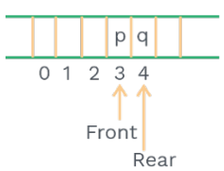


Fig. 3.19

and we need to insert an element to the above queue, but we can't. Even though lot of space is free in the queue. We can't insert any new element in queue because rear is at the last position.

- Circular queue is upgraded form of linear queue in which 'Front' and 'Rear' can be represented on a circular structure keeping first position and last position connected.
- Circular queue is represented as:

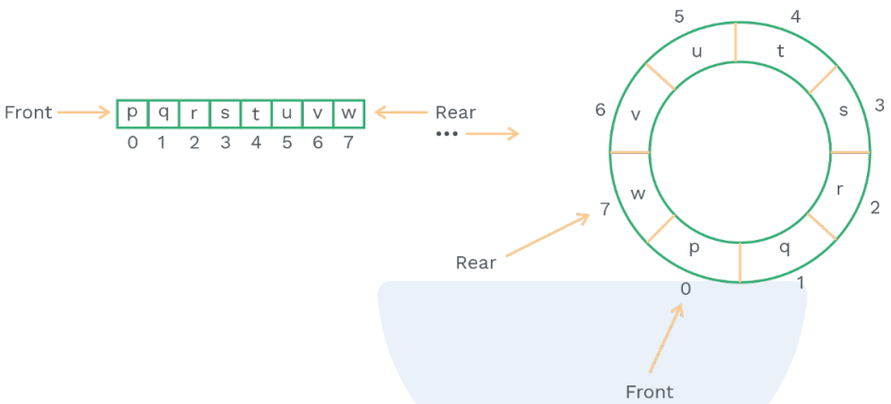


Fig. 3.20 Circular Queue

Priority queue:

In Priority queue, every element is associated with some priority and according to priority an element in the queue is processed, especially on deletion time.

So, while inserting elements on the queue, we can follow any order but, while deleting, we consider the requirement.

Rack Your Brain

Consider a queue of size '5'. Find the value of 'Front' and 'Rear' after – 5 insertions, 2 deletions and 2 insertions respectively.

Types of priority queue:

1) Ascending priority queue
2) Descending priority queue

1) Ascending priority queue:

We insert items arbitrarily in the queue.

While deleting, on the first deletion, it gives the smallest item of all, and so on.

For example: items are 3, 9, 2, 0, 11, 10, 20

- Insertion and elements follow as:

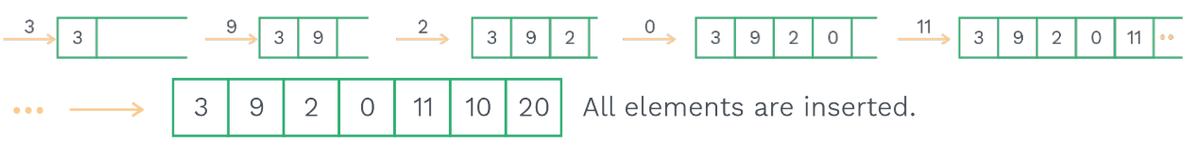


Fig. 3.21

Deletion of elements follow as:

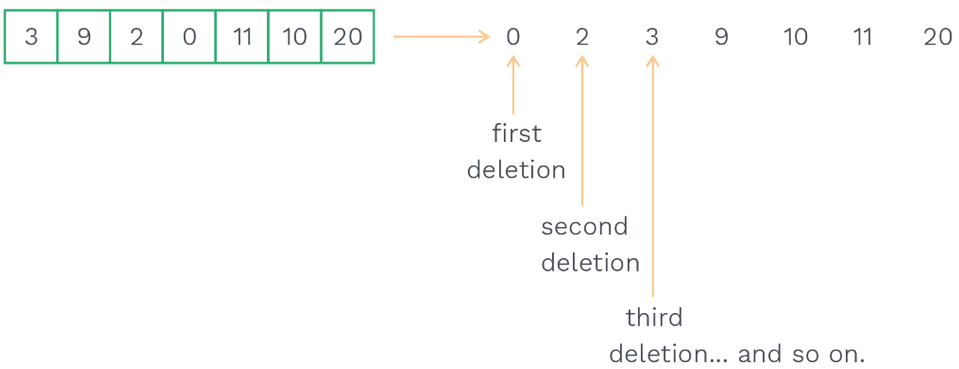


Fig. 3.22

Thus, by ascending the priority queue, we get items from the queue in ascending order.

2) Descending priority queue:

- Insertion of items on queue follows an arbitrary order.
- Deletion follows as: On first deletion, it gives the largest item of all and so on.
  For example: Items are 3, 9, 2, 0, 11, 10, 20.
- Insertion of items follows as:

Fig. 3.23

Deletion of elements/items follow as:

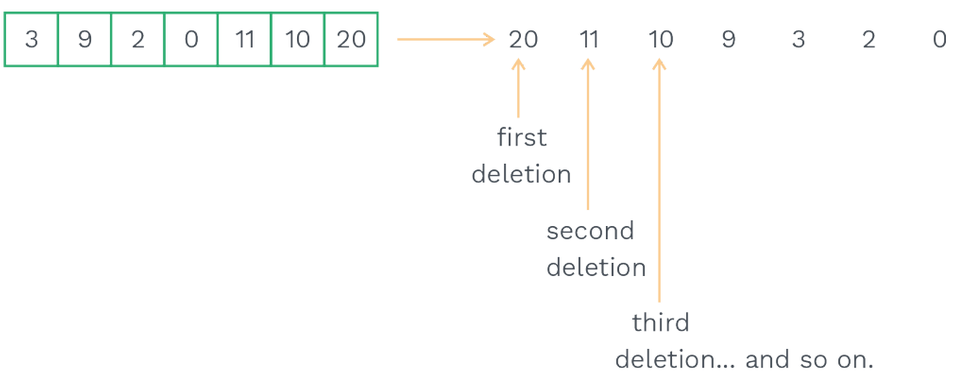


Fig. 3.24

Thus, by descending the priority queue, we get items from the queue in descending order.

Rack Your Brain

There is ascending priority queue of size 10. We are given ten numerals to insert into the queue. After insertion, we performed deletion till 5th position. What is the result? (Take positions of items, starting from 0, 1 and so on).

Previous Years' Question

A priority queue Q is used to implement a stack that stores characters. PUSH (C) is implemented INSERT (Q, C, K) where K is an appropriate interger key chosen by the implementation. POP is implemented as DELETEMIN(Q). For a sequence of operations, the keys chosen are in

a) non–increasing order **b)** non–decreasing order
c) strictly increasing order **d)** strictly decreasing order
Sol: d) **(GATE 1997 - 2 Marks)**

Doubly ended queue (Dequeue):
A doubly ended queue is a kind of queue in which insertion, and deletion operations are performed at both places of 'Front' and 'Rear'.

- Let's understand it with a diagrammatic view:

Consider a queue of size '7', where 'F' represents 'Front' and 'R' represents 'Rear'.

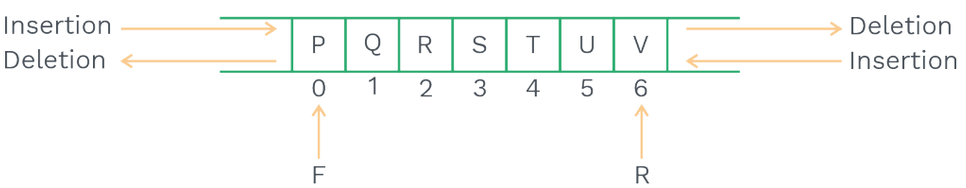


Fig. 3.25

As shown in the diagram, clearly, we can insert or delete items on both 'F' and 'R'.

Rack Your Brain

A doubly ended queue is given below. Calculate the value of 'Front' and 'Rear' after the following sequence of operations :
1) Two–deletion at 'Front'
2) Three–deletions at 'Rear'
3) Two–insertions at Rear'.

Input restricted queue:
Input restricted queue is a kind of queue in which insertion operations are performed only at 'Rear' as in regular queue, but deletion operations are performed at 'Front' and 'Rear' both the positions.

Let's understand it with a diagrammatic view:
Consider the queue, in which 'F' represents 'Front' and 'R' represents 'Rear'.

Fig. 3.26

In the above diagram, it can be seen that items can be deleted from 'Rear' also.

Output restricted queue:
Output restricted queue is a kind of queue; deletion operations are performed only at 'Front' as in regular queues, but insertion operations can be performed at 'Front' and 'Rear' both the positions.

A diagrammatic representation can be shown as:
Consider the queue, in which 'F' represents 'Front' and 'R' represents 'Rear'.

Fig. 3.27

In above diagram, It can be seen that items can be inserted at 'Front' also.

Note:

Priority queue, input restricted queue, output restricted queue, doubly ended queue, all these kinds of queue are not guaranteed to satisfy FIFO property.

3.10 IMPLEMENTATION OF QUEUE

We can implement queue by using other data structures like using array, stack or linked list etc.

Implementation of queue using array:

- Implementing queue means implementing basic operations of queue by using array.
- So, let's take an array first ;
- Array is denoted with 'q', which represents a queue.
- 'S' is the size of array, 'q'.
- Front is represented with 'F'.
- Rear is represented with 'R'.

Enqueue operation:

```
Void insert(x)        // An item 'x' is be inserted.
 {
        If (R+1 == S)     // Checking overflow condition.
          {
              printf ("q is overflow") ;
              exit(1) ;
          }
        else
          {
              if (R == –1)          // Checking if queue 'q' is
              F ++ ;                   empty, if yes–performs
              R ++ ;                   first insertion of an item.
              else

              R = R+1 ;             // Normally, if there is space,
                                       rear increments one by one.

              q[R] = x ;            // item 'x' is stored at location R.

          }
 }
```

Dequeue operation:

```
int remove ( )
  {
     int I ;
     If (F == –1)        // Checking, if 'q' has no element.
       {
          printf ("q is underflow") ;
          exit (1) ;
       }
     else
       {
        I = q[F] ;       // Storing elemented from position 'F'
                            to integer variable I.

        If (F==R)        // It performs last element removal.
        F = R =–1 ;
        else

        F = F + 1 ;      // Normally, if there are some items in queue, front
        return (I) ;        increments one by one after deletion of elements sequentially.

       }
  }
```

Assumptions:

- Array positions start from zero i.e. 0, 1, 2 ..... S–1 (Total S)
- Initially, when there is no item in queue, F = –1 and R = –1.
- If at any stage F = R, there is only one element in the queue.
- In general, 'R' increments by one for each insertion.
- In general, 'F' increments by one for each deletion.
- So, queue can be represented by using an array as:

Here, S = 8 i.e. size of queue.

Rack Your Brain

There is queue 'Q' implemented with an array. In this queue front is represented with 'F', and rear is represented with 'R'. size of 'Q' is S. Write the condition for (i) overflow of queue 'Q'.
(ii) Underflow of queue 'Q'.

Previous Years' Question

A queue is implemented using an array such that ENQUEUE and DEQUEUE operations are performed efficiently. Which one of the following statements is correct (n refers to the number of items in the queue)?

a) Both operations can be performed in O(1) time
b) At most one operation can be performed in O(1) time but the worst case time for the other operation will be $\Omega(n)$
c) The worst case time complexity for both operations will be $\Omega(n)$
d) Worst case time complexity for both operations will be $\Omega(\log n)$

Sol: a) **(GATE CSE 2016 - SET 1)**

Previous Years' Question

Suppose you are given an implementation of a queue of integers. The operations that can be performed on the queue are :

i) isEmpty (Q) – returns true if the queue is empty, false otherwise.
ii) delete (Q) – deletes the element at the front of the queue and returns its value.
iii) insert (Q, i) – insert the integer i at the rear of the queue.

Consider the following function :

```
void f (queue Q) {
    int i ;
    if (! isEmpty(Q) ) {
        i = delete(Q) ;
        f (Q) ;
        insert (Q, i) ;
    }
}
```

What operation is performed by the above function f?

a) Leaves the queue Q unchanged
b) Reverses the order of the elements in the queue Q
c) Deletes the element at the front of the queue Q and inserts it at the rear keeping the other elements in the same order
d) Empties the queue Q

Sol: b) **(GATE CSE - 2007)**

Implementation of circular queue using array:

- Implementing a circular queue means implementing basic operations of the circular queue by using array.
- Let's take an array 'q'.
- 'S' is the size of a circular queue i.e., 'q'.
- Front is represented with 'F'.
- Rear is represented with 'R'.

1) Enqueue operation: (To insert items)

```
void C_insert(x)        // x is an item to be inserted.
{
        if ((R+1) mod S == F)
            {
                printf ("CQ is full") ;        // circular queue is full.

                exit(1) ;
            }
        else
            {
                if (R == –1)
                   {
                       F++ ;
                       R++ ;
                   }
                else
                       R = (R+1) mod S ;  // This condition will use the
                                             array in a circular manner, to
                                             insert an item on the queue.
            }
q [R] = x ;
}
```

The Above code will insert items in a circular manner by the testing status of a queue in circular manner.

2) Dequeue operation: (To remove items)

```
int C_remove( )
   {
      int I ;
      if (F == –1)
        {
           printf (“Underflow”) ;
           exit(1) ;
         }
      else
         {
           I = q [F] ;
           if (F == R)
              F = R = –1 ;
           else
              F = (F+1) mod S ;    // This condition will use the array to
                                      remove items in a circular manner.
           return I ;
         }
   }
```

Assumptions:

- Array position starts from zero i.e. 0, 1, 2 ..... S–1.
- Initially when there is no any item in C_queue F = –1, R = –1.
- If at any stage F = R, there is only one item in the C_queue.

Previous Years' Question

Suppose a circular queue of capacity (n–1) elements is implemented with an array of n elements. Assume that the insertion and deletion operations are carried out using REAR and FRONT as array index variables, respectively. Initially, REAR = FRONT = 0. The conditions to detect queue full and queue empty are

a) full : (REAR+1)
mod n == FRONT
empty : REAR == FRONT

b) full : (REAR+1)
mod n == FRONT
empty : (FRONT+1)

c) full : REAR == FRONT
empty : (REAR+1)
mod n == FRONT

d) full : (FRONT+1)
mod n == REAR
empty : REAR == FRONT

Sol: a) **(GATE CSE - 2012)**

Implementation of queue using stack:

- **A brief note on stack:**

A stack is a data structure which functions according to property as last in first out. Items are pushed onto stack and can be popped out of it.

Stack has two basic operations:

1) Push( ): To push items on stack
2) Pop( ): To pop items from stack.

- Implementation of queue using stack means, performing basic operations of queues by using basic operations of a stack and satisfying the property of queue.

- **Basic operations of queue are:**

1) Enqueue (to insert)
2) Dequeue (to remove)

- Property that should be satisfied for queue is first in first out (FIFO).
- So, we have to satisfy the property FIFO by using a push–pop operation.

For example:

- We want a queue whose items are P, Q, R, S, T, U, V.
  * Let's implement this queue by using stack.

1) Enqueue operation: (To insert items on queue)

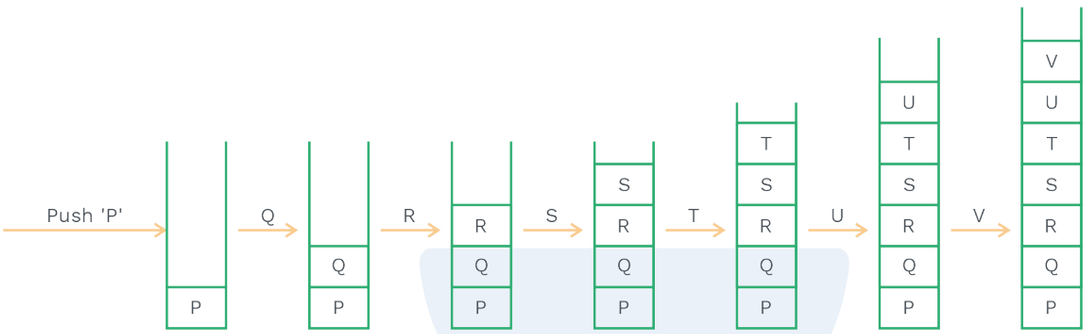


Fig. 3.29

By performing above operations, we enqueued all the items successfully. So, enqueue operation can be given as:

```
Insert (x)
{
   Push(x, ST) ;
}
```

x : is an item to be inserted into queue

ST : is the stack.

Push(x, ST) : performs push item x on stack ST.

2) Dequeue operation: (To delete items from queue)

Till now, we have the stack : (means queue) where we have inserted all the items.

Fig. 3.30

- We need to delete first item, second item and so on from stack which we had pushed. For that, we need two stacks. The first stack is ST, and second stack is S'T.

- We need to pop all items from ST and pushed onto S'T. Thus we will get the first item inserted onto ST as top of stack_S'T, and we can get the first item deleted easily.

Operations:

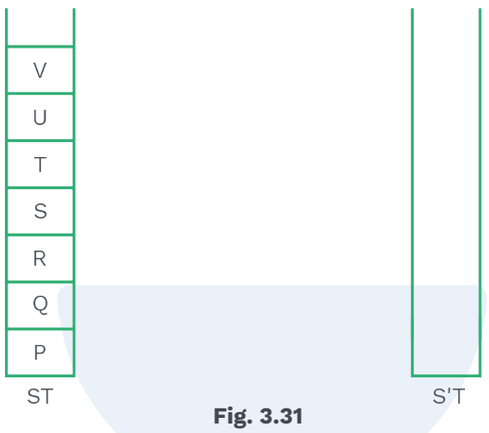


Fig. 3.31

1. Pop (V, ST)
2. Push (V, S'T)
3. Pop (U, ST)
4. Push (U, S'T)
5. Pop (T, ST)
6. Push (T, S'T)
7. Pop (S, ST)
8. Push (S, S'T)
9. Pop (R, ST)
10. Push (R, S'T)
11. Pop (Q, ST)
12. Push (Q, S'T)
13. Pop (P, ST)
14. Push (P, S'T)

- So,

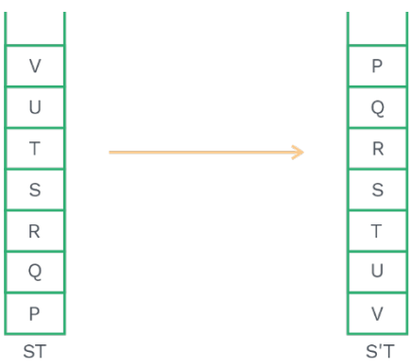


Fig. 3.32

Now, pop (P,S'T) 1st dequeue

- **So, the dequeue operation can be given as:**

```
Remove( )
{
        if (S'T has some elements/items)
            return (pop (x, S'T))
        else
            {
                while (ST has some elements/items)   // While loop used for
                    {                                continuously popping an item
                        I = Pop (x, ST);             from ST which are supposed
                                                     to be pushed onto S'T.
                    }
                return (Pop (x, S'T)) ;
            }
}
```

Thus, it's successfully done using stack. i.e. queue is implemented by using stack.

Rack Your Brain

A queue is considered to be implemented by using stack. No. of items is 5 as shown in the diagram

A	B	C	D	E	

Element–D has to be removed. How many push–pop operations are needed?

Previous Years' Question

An implementation of a queue Q, using two stacks S1 and S2, is given below :

```
void insert (Q, X) {
    push (S1, X) ;
}
void delete (Q) {
    if(stack_empty(S1)) then {
    print ("Q is empty") ;
    return ;
} else while (!(stack_empty(S1))) {
    X = pop(S1) ;
    push(S2, X);
}
X = pop(S2) ;
}
```

Let, n insert and m(≤ n) delete operations to be performed in an arbitrary order on an empty queue Q.
Let, x and y be the number of push and pop operations performed respectively in the process.
Which one of the following is true for all m and n?
a) $n + m \leq x \leq 2n$ and $2m \leq y \leq n + m$
b) $n + m \leq x \leq 2n$ and $2m \leq y \leq 2n$
c) $2m \leq x \leq 2n$ and $2m \leq y \leq n + m$
d) $2m \leq x \leq 2n$ and $2m \leq y \leq 2n$
Sol: a) **(GATE CSE - 2006)**

Implementation of queue using linked list

- **A brief note on linked list:**

Linked list is a data structure which is in the form of list of nodes as shown in the diagram below:

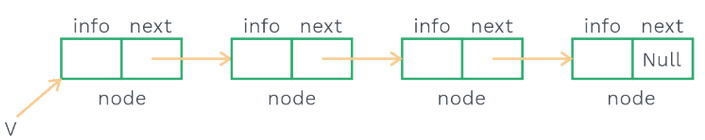


Fig. 3.33

- A node contains two fields:
  1) An information field (we can use an integer data or char data etc.)
  2) Next address field
- An external pointer 'V' that points to the first node, and we can access the entire list by this external pointer. 'V' is a pointer variable which contains the address of the first node of the linked list.
- Implementing queue using linked list means performing enqueue and dequeue operations on the linked list to satisfy the condition of first in first out, which is of queue's property.
- **Definition of two operations of queue using linked list:**
  1) Enqueue/insert operation:
  It means we need to add a node at the end of a given linked list.
  2) Dequeue/remove operation:
  It means we need to delete a node from starting of a given linked list.
- Let's understand with an example:

Consider a linked list:

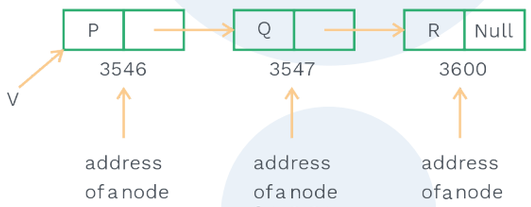


Fig. 3.34

1) Enqueue/insert operation:

Steps: **I)** – Create a node

II) – Attach it at the end of the linked list and so on

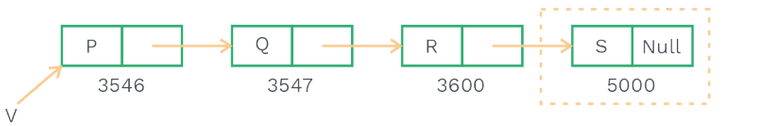


Fig. 3.35

- **Code for enqueue operation:**

```
Insert(x)        // x is an item/data to be inserted.
{
   New_node = malloc( ) ;     // Creation of a dynamic memory space
                                 for a new node.
   if (New_node == Null)      // When memory is over, New_node
                                 won't get any space.
```

```
                {
                printf ("overflow") ;
                exit(1) ;
                }
                S = V ;        // Taking another pointer variable 'S'.

                While (S ⟶ next ! = Null)      // 'S' traverses to the
                {                                 end of the linked list.
                S = S ⟶ next ; }
                S ⟶ next = New_node ;
                S = new_node ;
                }
        }
```

2) Dequeue/remove operation:

Steps: I – Just remove the first node, second node and so on.

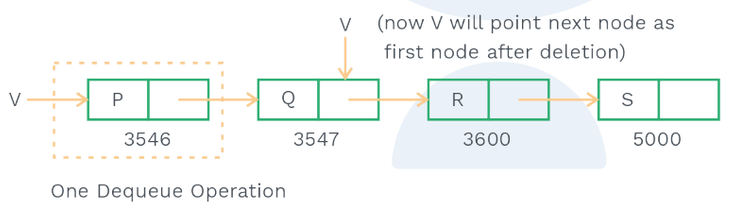


Fig. 3.36

- **Code for dequeue operation:**

```
Remove( )
{
                int I ;
Removed_node = V ;        // assigning the value of pointer variable 'V' to a
                             new pointer variable which is Removed_node.
I = Removed_node ⟶ data ;        // Taking Removed_node data out to integer
                                    variable–I.
V = V → next ;
free (Removed_node) ;        // deallocating the node pointed by Removed_node.
Removed_node = Null ;
return (I) ;
}
```

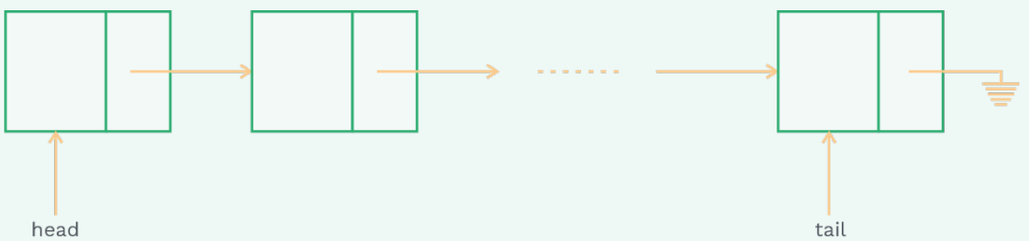
So, now list will appear as:

Fig. 3.37

Thus, we can perform insert/remove operation of the queue using linked list.

Previous Years' Question

A queue is implemented using a non–circular singly linked list. The queue has a head pointer and tail pointer, as shown in the figure. Let n denote the number of nodes in the queue. Let enqueue be implemented by inserting a new node at the head and dequeue be implemented by deletion of a node from the tail.



Which one of the following is the time complexity of the most time–efficient implementation of enqueue and dequeue, respectively, for this data structure?

a) $\theta(1), \theta(1)$
b) $\theta(1), \theta(n)$
c) $\theta(n), \theta(1)$
d) $\theta(n), \theta(n)$

Sol: b)

(GATE CSE - 2018)

Chapter Summary

- Stack is a linear data structure in which the items can be added or removed only at one end called top of stack. Stack satisfied LIFO property.
- Basic operations of stack : **1)** PUSH
  2) POP
- Stack conditions: **a)** Underflow condition
  b) Overflow condition
- Expression evaluation:
  We need to convert infix expression to postfix expression for evaluation.
  → Stack is only used in both conversion and evaluation.
- Stack can be implemented with array as well as linked list.
- Tower of Hanoi is an application of stack.
- Queue is a data structure which follows the first in first out property.
- Operations of queue: **1)** Enqueue/insert operation
  2) Dequeue/remove/delete operation
- Types of queue:
  1) **Circular queue:** A circular queue in which the first position and last position is connected.
  2) **Priority queue:** While inserting elements into the queue we can follow any order, but while deleting, we consider the priority.
  Types: Ascending priority queue and
  Descending priority queue.
  3) **Doubly ended queue:**
  It is a kind of queue in which insertion and deletion operations are performed at front and rear both places.
  4) **Input restricted queue:**
  Insertion operations are performed only at the rear.
  5) **Output restricted queue:**
  Deletion operations are performed only at the front.
- Implementation of queue:
  1) **Using array:**
  It means implementing basic operations of queue to satisfy FIFO using array.
  2) **Using stack:**
  This means implementing queue using Push–Pop operations to satisfy FIFO.
  3) **Using linked list:**
  It means implementing queue by adding or deleting nodes to satisfy FIFO.

4 Linked List

4.1 BASICS OF LINKED LIST

Introduction:

- A brief about why linked list came into the picture:

Basically, there are two reasons behind taking the linked list into the picture.

1) Insertion and deletion is easier:

In Linked List, insertion and deletion take O(1).

2) Size:

When a conventional array is used, the size of the array is fixed once declared. We cannot increase or decrease the array size based on the elements inserted or deleted.

- Linked list is a data structure which is in the form of list of nodes, as shown in the diagram below:

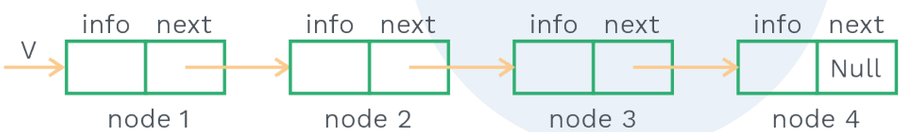


Fig. 4.1

- A node contains two fields:
  1) An information field (Head)
  2) Next address field (Tail)
- Information field contains actual data, and the next address field contains the address of the next node of the list.
- An external pointer 'V', is a pointer variable which contains an address for the first node of the list, and points to the first node.
- We can access the entire list with this external pointer 'V'.
- Linked list is a flexible data structure as we can insert any number of nodes and delete nodes from the given linked list.
- Next field of last node of linked list contains null, thus null pointer is used to signal the end of a list.

Note:

The list with no node is called the empty list or the null list.

Note:

The value of external pointer 'V' to an empty list is the null pointer.

4.2 OPERATIONS ON LINKED LIST

Insertion operation:

1) Insertion of a node to the front of a linked list:

Consider a Linked List:

Fig. 4.2

Now we need to insert a node having info as '4' to the front of the above list.

Steps to insert:

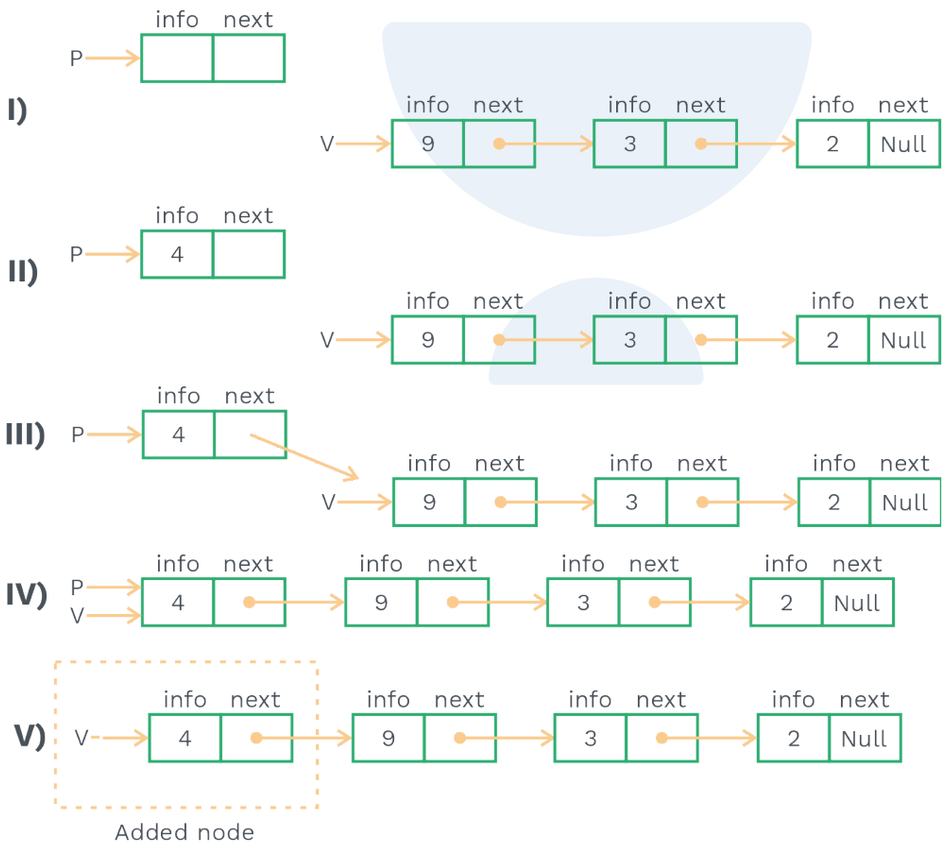


Fig. 4.3

I) We create an empty node ;  , which is pointed by a variable P, means the address of the new node is set to variable P.

II) Integer '4' is inserted into the info field of the newly allocated node.
III) Next field of the new node is updated with the address of the first node of the given list i.e. V.
IV) P–value is assigned to V–value so that V is pointing now to the newly attached node.
V) We get the final list after inserting a node with an info field having integer '4'.

* **Algorithm for insertion of a node to the front of a linked list:**

```
1) P = getnode( ) ;
2) info(P) = 4 ;
3) next(P) = V ;
4) V = P ;
5) return (V) ;
```

Meaning:

1) P = getnode( ) ; // This operation obtains an empty node and sets the content of a variable–'P' to the address of that node.

2) info(P) = 4 ; // By this operation, integer–'4' is inserted into the info field of new node. info(P) implies the info field of a node pointed by P).

3) next(P) = V ; // next(P) implies next field of a node pointed by P, and the given statement of code implies node pointed by P be added to the front of the list.

4) V = P ; // the address contained in P is now assigned to the variable V ; hence previous address residing in V is replaced by the address residing in P.

5) return (V) ; // It will return the linked list, pointed by 'V'.

Note:

```
getnode( )
{
Struct node
{
   int info ;
   Struct node* next ;
    A ;
   return &A ;
   }
```

(We will see detailed code in the **section–4.4 Implementation of Linked List.)**

Generalised algorithm to add any object 'x' to the front of a linked list:

```
1) P = getnode( ) ;
2) info(P) = x ;
3) next(P) = V ;
4) V = P ;
5) return (V) ;
```

2) Insertion of a node to the end of a list:

Consider the previous list only:

Fig. 3.4

Now, we have to insert a node having info as 4 to the end of the above list.

Steps to insert:

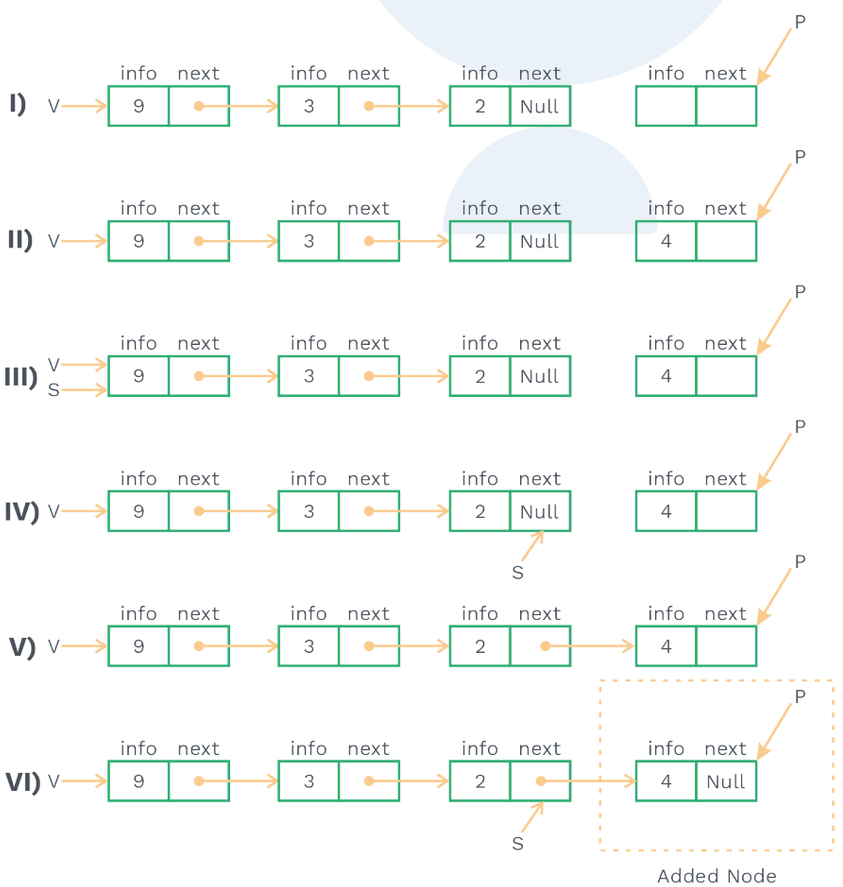


Fig. 4.5

I) An empty node is created, which is pointed by a variable 'P'.
II) Integer '4' is inserted into the info field of the newly allocated node.
III) The value of 'V' is assigned to a new pointer variable 'S'.
IV) 'S' traverses the linked list till the end of the linked list.
V) Last node of the given linked list now points to the newly created node pointed by P.
VI) next(P) is set to 'null', which implies it is last node of the given linked list.

* **Algorithm for insertion of a node to the end of a given linked list:**

```
P = getnode( ) ;
info(P) = 4 ;
S = V ;
while (next(S)! = Null)
S = next(S) ;
next(S) = P ;
next(P) = Null ;
return (V) ;
```

Generalised algorithm to add any object 'x' to the end of a linked list:

```
P = getnode( ) ;
info(P) = 4 ;
S = V ;
while (next(S)! = Null)
S = next(S) ;
next(S) = P ;
next(P) = Null ;
return (V) ;
```

Time complexity:
1) Insertion at beginning = O(1).
2) Insertion at the end when we have only head pointer = O(n).
3) Insertion at the end when we have both head and tail pointer = O(1).

Deletion operation:

1) Deletion of a node from the front of a linked list:

Consider a linked list:

Fig. 4.6

We have to delete the first node of the given linked list.

Steps to delete:

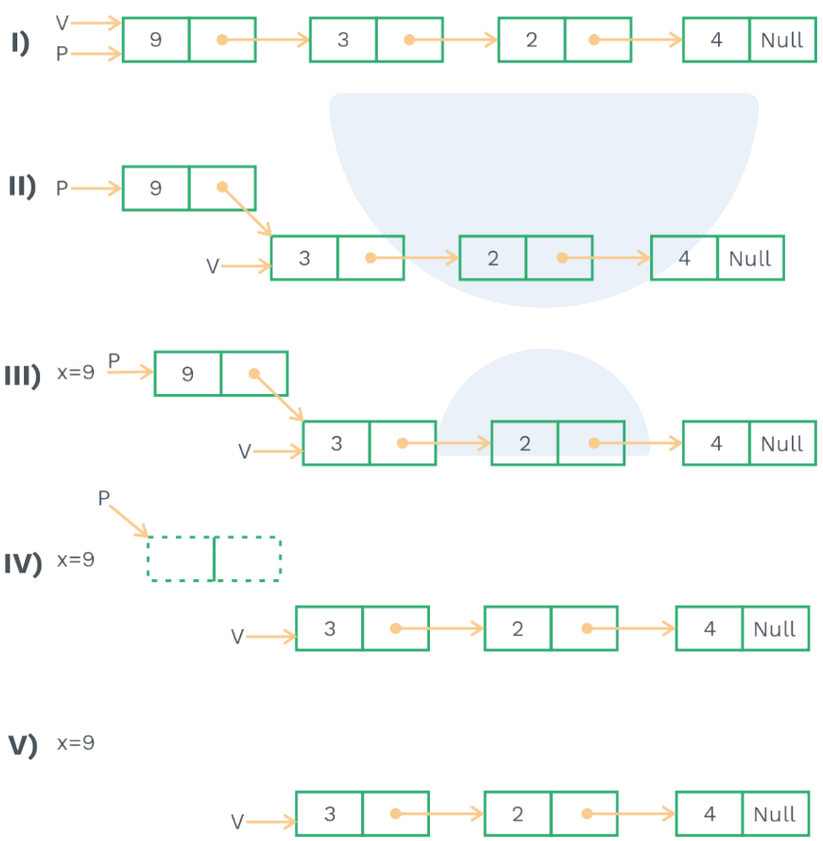


Fig. 4.7

I) The first node of the given linked list, which is pointed by 'V', is pointed by one more variable, 'P'.

II) Next field of 'P' is now assigned to 'V'.

III) Info field is copied to a variable 'x'.

IV) Node pointed by 'P' is now free, which means deleted.

V) Final linked list after deleting the first node is returned.

* **Algorithm for deletion of a node from the front of a linked list:**

```
P = V ;
V = next (P) ;
x = info (P) ;
free node (P) ;
return (V) ;
```

Note:

freenode(P) ; means the node pointed by 'P' is now not allocated. The corresponding memory space is ready for reuse.

2) **Deletion of a node from the end of a linked list:**

Consider the previous linked list:

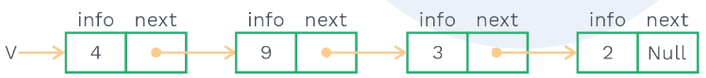


Fig. 4.8

We have to delete the last node from this linked list.

Steps to delete:

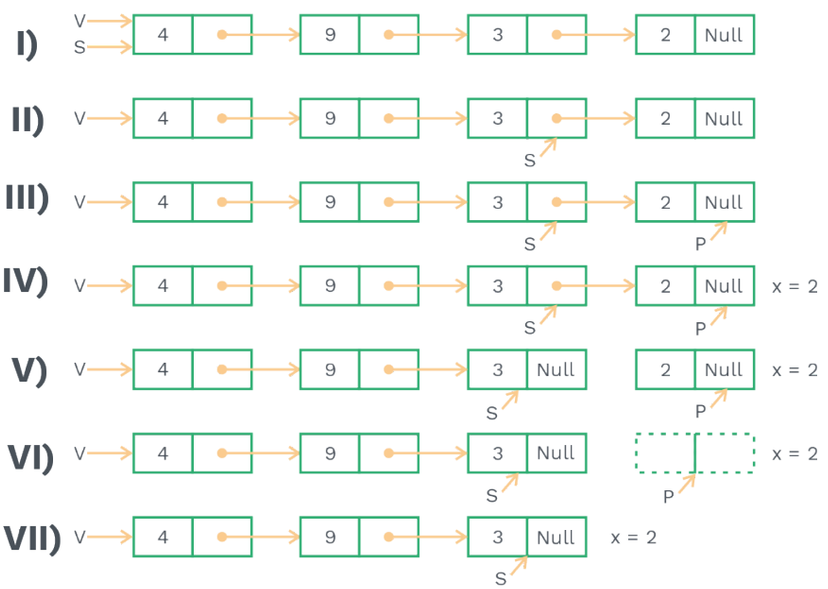


Fig. 4.9

I) First node of the given linked list, which is pointed by 'V', is pointed by one more variable, 'S'.
II) 'S' traverses the given linked list till the second last node, and points it.
III) One variable, 'P', points to last node of the linked list.
IV) info field of the node pointed by P is copied to a variable 'x'.
V) next(S) is set to null, to make it the last node.
VI) node pointed by P is deallocated.
VII) Final linked list after deleting the last node is returned.

* **Algorithm for deletion of a node from the end of a linked list:**

```
S = V ;
while (next(next(S)) ! = Null)
S = next(S) ;
P = next(S) ;
x = info(P) ;
next(S) = Null ;
freenode(P) ;
return(V) ;
```

Time complexity:
1) Deletion from beginning = O(1).
2) Deletion of given node = O(n).
3) Deletion of node at end = O(n).

Rack Your Brain

Write an algorithm to perform each of the following operations:
a) Append an element to the end of a list.
b) Concatenate two lists.
c) Delete the last element from a list.
d) Delete the n^{th} element from a list.

Previous Years' Question

Q. Let P be a singly linked list, Let Q be the pointer to an intermediate node x in the list. What is the worst–case time complexity of the best known algorithm to delete the node x from the list?
a) O(n) **b)** $O(\log^2 n)$
b) O(log n) **d)** O(1)
Sol: d) **(GATE CSE: 2004)**

Previous Years' Question

Q. What is the worst case the time complexity of inserting n elements into an empty linked list, if the linked list needs to be maintained in sorted order?
a) $\Theta(n^2)$ **b)** $\Theta(n)$
c) $\Theta(n\log n)$ **d)** $\Theta(1)$
Sol: a) **(GATE CSE: 2020)**

Searching operation:

Linear search on a linked list:

Consider a linked list, pointed by V, we need to find the address of the node having value '4'.

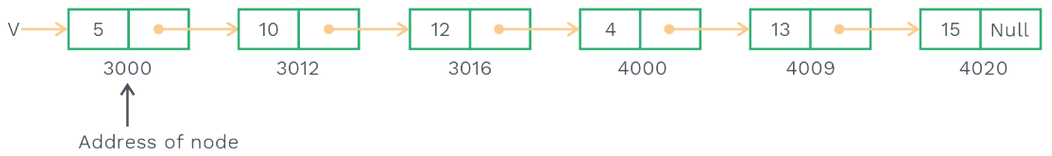


Fig. 4.10

Finding integer '4':

I) Start traversing nodes one by one.

II) Compare the info field of nodes with integer '4'.

III) Return the address of the node containing integer '4'.

Time complexity:

Linear search on a single linked list will take O(n) [worst case] because at the worst case, we need to traverse the whole linked list with unsorted elements of it, to find the desired element.

Note:

Binary search on linked list is possible but not efficient.

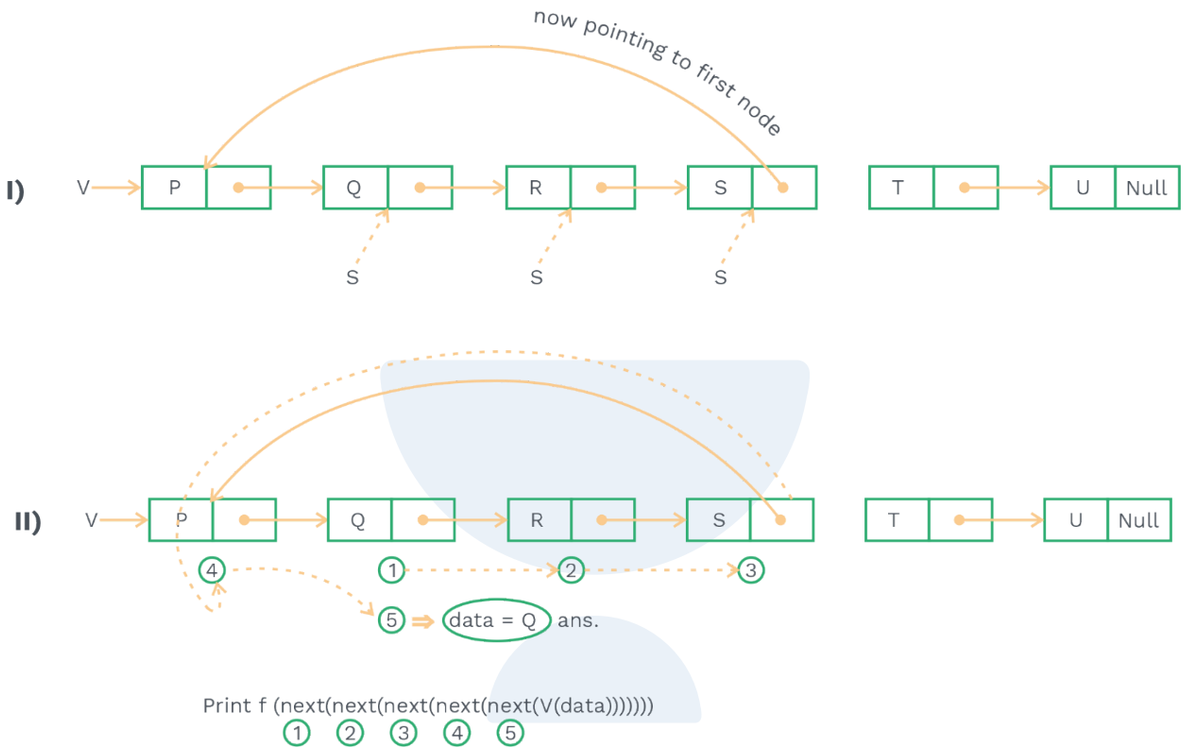
PRACTICE QUESTIONS

Q1 **Consider the given linked list below:**

Find the result of the code given below:

```
Struct node * S ;
S = next(next(next(V)))
next(S) = V
printf(next(next(next(next(next(V)))))(data))
```

Sol:



Previous Years' Question

Q. In the worst case, the number of comparisons needed to search a singly linked list of length n for a given element is

a) $\log_2 n$ **b)** n/2
c) $\log_2 n - 1$ **d)** n

Sol: d) **(GATE CSE: 2002)**

Previous Years' Question

Q. Linked lists are not suitable data structures of which one of the following problems?

a) Insertion sort
b) Binary search
c) Radix sort
d) Polynomial manipulation

Sol: b) **(GATE CSE: 1994)**

4.3 TYPES OF LINKED LIST

Singly linked list:

- Singly linked list is a type of linked list in which there is one next field which points to the next node of the linked list.
- Representation of singly linked list.

Fig. 4.11

- Each node contains two fields
  1) Information field: Contains objects
  2) Next Field: Contains the address of the next node means points to the next node.
- Last node next field contains null, which implies that it is the last node.
- It is flexible in size means, we can insert any number of nodes to the existing linked list, and also can delete nodes.

Doubly linked list:

- Doubly linked list is a type of linked list in which two fields are reserved for addresses other than the information field, i.e. one for the previous node address, and one for the next node address.

Representation of doubly linked list:

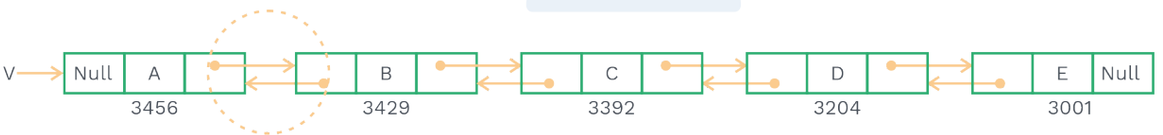


Fig. 4.12

- Each node contains three fields
  1) Previous field: Contains the address of the previous node means points to previous node.
  2) Information field: Contains objects
  3) Next field: Contains the address of the next node means points to the next node.
- The first node, previous address field posses null value. If some node contains the previous field with a null value, it means it is the first node of a doubly linked list.
- Last node, next address field posses a null value. If some node contains the next field with null value, it means it is the last node of a Doubly linked list.
- Doubly linked list is also flexible in size means we can add any number of nodes to the existing linked list and also can remove nodes.

Previous Years' Question

Q. N items are stored in a sorted doubly linked list. For a delete operation, a pointer is provided to the record to be deleted. For a decrease–key operation, a pointer is provided to the record on which the operation is to be performed.
An algorithm performs the following operations on the list in this order :
$\Theta(N)$, delete, $O(\log N)$ insert,
$O(\log N)$ fund, and $\Theta(N)$ decrease–key.
What is the time complexity of all these operations put together?

a) $O(\log^2 N)$ **b)** $O(N)$
c) $O(N^2)$ **d)** $\Theta(N^2 \log N)$

Sol: c) **(GATE CSE: 2016 (Set–2))**

Circular singly linked list:

- Circular single linked list is a type of linked list in which the last node next field contains first node address.
- Representation of circular single linked list.

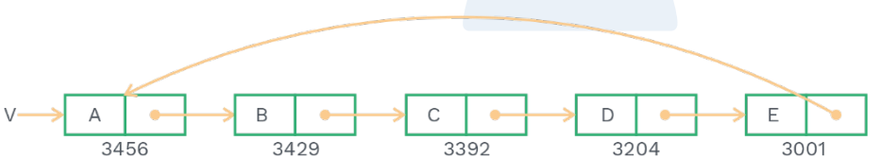


Fig. 4.13

- Each node similar to a single linked list contains two fields
  1) information field
  2) next field
- Last node next field contains the address of the first node of the linked list, which makes it circular.
- Similar to a single linked list, it is also flexible in size means we can add any no. of nodes to the existing linked list, which is circular and also can remove nodes by changing node links.

Previous Years' Question

Q. In a circular linked list organization, insertion of a record involves modification of:

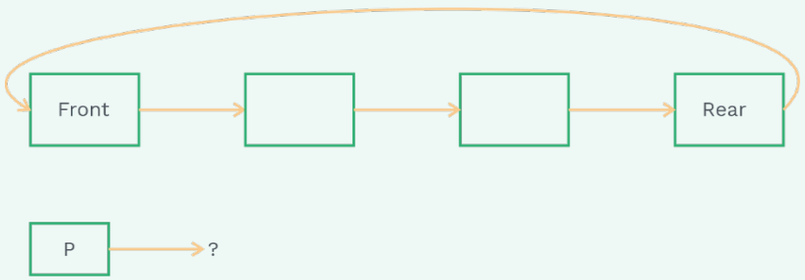
a) One pointer
b) Two pointer
c) Multiple pointers
d) No pointer

Sol: b) **(GATE CSE: 1987)**

- Advantage: We can go back in the list, wherever we want to reach in the circular single linked list.
- We can go back in the circular single linked list which takes O(n) worst case time complexity. 'n' is no. of elements.

Previous Years' Question

Q. A circularly linked list is used to represent a queue. A single variable p is used to access the queue. To which node should point p such that both the operations enqueue and dequeue can be performed in constant time?



a) rear node
b) front node
c) not possible with a single pointer
d) node next to front

Sol: a) **(GATE CSE: 2004)**

Circular doubly linked list:

- Circular doubly linked list follows two conditions on doubly linked list, which are:
  1) Last node next field contains first node address.
  2) First node previous field contains last node address.

Above two conditions on the doubly linked list make it circular doubly linked list.

- Representation of circular doubly linked list:

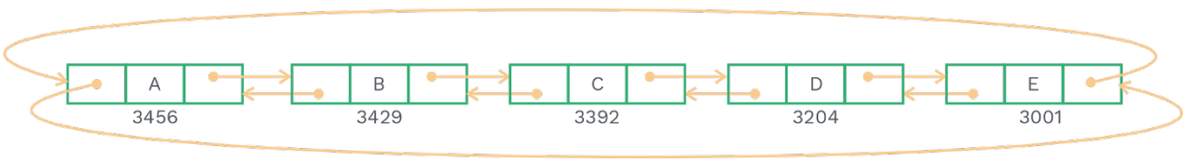


Fig. 4.14

- Each node similar to doubly linked list contains three fields:
  1) Previous field
  2) Information field
  3) Next field

- Similar to doubly linked list, it is also flexible in size means we can add any no. of nodes to the existing circular doubly linked list and also can remove nodes by changing node links.

4.4 IMPLEMENTATION OF LINKED LIST

Implementation of linked list using structures and pointers:

*** A brief note on structures and pointers:**

1) Structure:

It is a group of items in which each item is identified by its own identifier and its datatype.

- Each of the items is known as a member of the structure.
- Consider the following declaration:

```
Struct        {
        int a ;
        char b ;
        float c ;
    } A, B, C ;
```

- 'Struct' it a keyword used to declare a structure.
- There are three members of the structure they are:
  i) a)
  ii) b)
  iii) c)
- A, B, and C are structure variables, each of A, B, and C contains three members a, b, c.
- Using structure, one can define a datatype, which can be referred a user defined data type.

2) Pointer:

In C–language, programmers are allowed to reference the location of objects as well as the objects themselves.

- For example: Suppose there is a variable x of int type or float type or any type. Then, & x refers to the location which contains x. Here x is called a pointer.

Creating linked list:
Consider a linked list that we have to create,

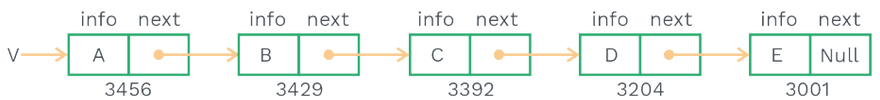


Fig. 4.15

Code:

```
Struct node
{
  Char info ;
  Struct node * next ;
} A1, B1, C1, D1, E1 ;
```

// Creating node structure and defining 5–variables A1, B1, C1, D1 and E1 of it. Datatype of these variables is struct node.

```
A1 = {'A', Null} ;
B1 = {'B', Null} ;
C1 = {'C', Null} ;
D1 = {'D', Null} ;
E1 = {'E', Null} ;
```

// Assigning values to each node.

```
A1. next = & B1 ;
B1. next = & C1 ;
C1. next = & D1 ;
D1. next = & E1 ;
```

// Updating address field i.e. next field of each node with the address of next node.

```
Struct node * V = & A1 ;
```

// Assigning address of the first node to a pointer variable of type 'struct node'.

```
return (V) ;
```

// return the linked list.

Note:

We can create linked lists according to our needs means, whatever type of object, we are required to keep in the info field of a node, we can keep it; for example – int data, char data, float data etc.

Creating doubly linked list:

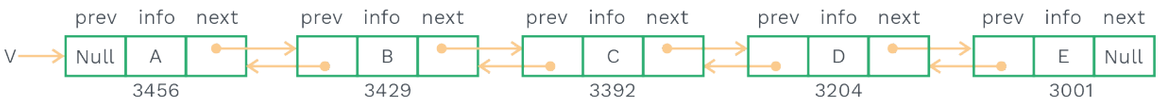


Fig. 4.16

Code:

```
Struct node
{
    Struct node* prev ;
    Char info ;
    Struct node * next ;
}   A1, B1, C1, D1, E1 ;

A1 = { Null, 'A', Null} ;
B1 = { Null, 'B', Null} ;
C1 = { Null, 'C', Null} ;
D1 = { Null, 'D', Null} ;
E1 = { Null, 'E', Null} ;

A1. next = & B1 ;

B1. prev = & A1 ;
B1. next = & C1 ;

C1. prev = & B1 ;
C1. next = & D1 ;

D1. prev = & C1 ;
D1. next = & E1 ;

E1. prev = & D1 ;

Struct node * V = & A1 ;
return (V) ;
```

Creating circular singly linked list:

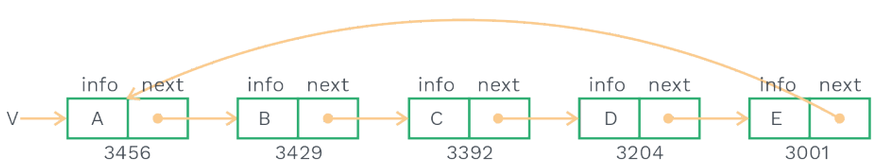


Fig. 4.17

Code:

```
Struct node
{
    Char info ;
    Struct node *next ;
}   A1, B1, C1, D1, E1 ;
A1 = {'A', Null} ;
B1 = {'B', Null} ;
C1 = {'C', Null} ;
D1 = {'D', Null} ;
E1 = {'E', Null} ;
A1. next = & B1 ;
B1. next = & C1 ;
C1. next = & D1 ;
D1. next = & E1 ;
```

```
E1. next = & A1 ;
```

```
Struct node * V = & A1 ;
return (V) ;
```

Creating circular doubly linked list:

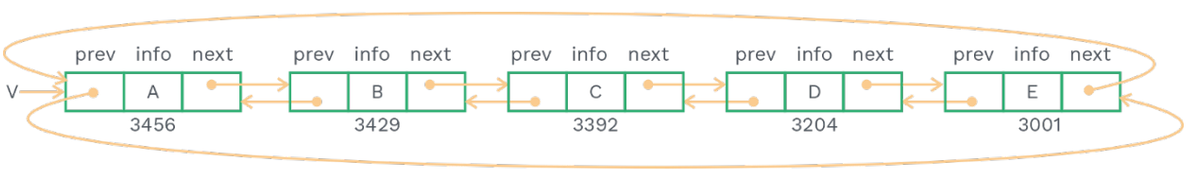


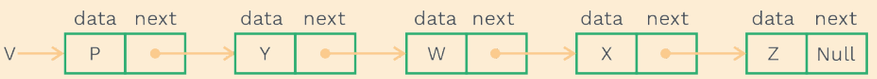
Fig. 4.18

Code:

```
Struct node
{
    Struct node * prev ;
    Char info ;
    Struct node *next ;
}   A1, B1, C1, D1, E1 ;
A1 = {Null, 'A', Null} ;
B1 = {Null, 'B', Null} ;
C1 = {Null, 'C', Null} ;
D1 = {Null, 'D', Null} ;
```

```
E1 = {Null, 'E', Null} ;
A1. prev = & E1 ;
A1. next = & B1 ;
B1. prev = & A1 ;
B1. next = & C1 ;
C1. prev = & B1 ;
C1. next = & D1 ;
D1. prev = & C1 ;
D1. next = & E1 ;
E1. prev = & D1 ;
E1. next = & A1 ;
Struct node *V = & A1 ;
return (V) ;
```

Q2 **Write a function for the given linked list to delete a node which contains the data 'x'.**

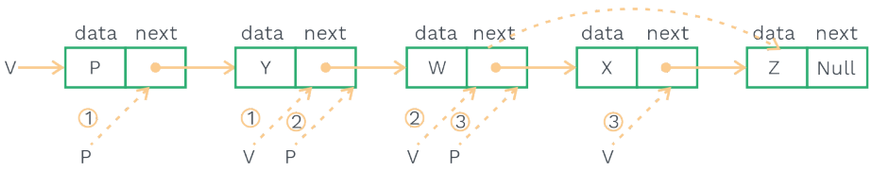


Sol:

```
Void delete_data_x( )
{ while (V → data ! = x && V → next ! = Null)
{
      Q = V ;
      P = V ;        // P is a new pointer variable.
      V = V → next;
}
      if (V → data == x)
{
      P → next = V → next ;

freenode (V);
```

```
V = Null ;
return (Q) ;
}
```



① ② ③ are sequence of execution of conditions in the function.

Resulted linked list is now:

Fig. 4.19

Previous Years' Question

Q. Consider the function f defined below.

```
Struct item {
    int data ;
    Struct item * next ;
} ;
int f(struct item * p) {
return ((p == Null) || (p → next == Null) ||
    ((p → data <= p → next → data) && f(p → next))) ;
}
```

For a given linked list p, the function f returns 1 if and only if

a) the list is empty or has exactly one element

b) the elements in the list are sorted in non–decreasing order of data value

c) the elements in the list are sorted in non–increasing order of data value

d) not all elements in the list have the same data value

Sol: b)

4.4 USES OF LINKED LIST

- Different uses of linked list are as follows:

1) Implementation of stacks:

We can implement stacks by using a linked list, to satisfy the LIFO property by adding and removing nodes.

2) Implementation of queues:

Queues can be implemented by using a linked list to satisfy the FIFO property by adding nodes at last or by removing nodes from the start of the linked list.

3) Representation of graphs:

Graphs are represented by an adjacency list.

For example:

Let's take a graph:

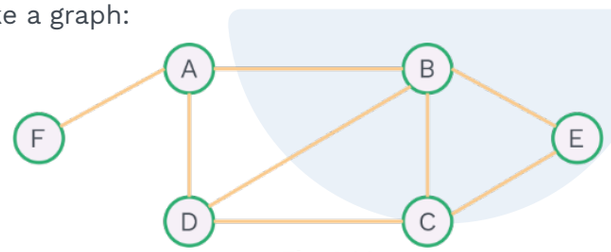


Fig. 4.20

Adjacency list representation of above graph:

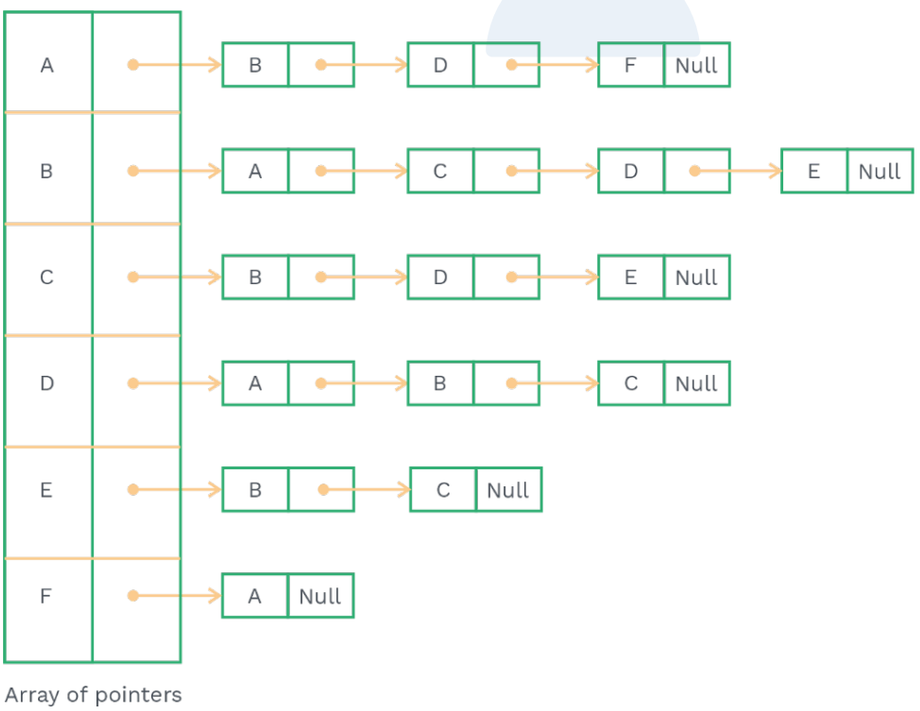


Fig. 4.21

4) Representation of sparse matrices:
To represent or to store sparse matrices, an array is not a good option. We use linked list to save space.

5) Memory allocation:
Dynamically, memory is allocated by using linked list by maintaining list of free blocks.

Chapter Summary

- Linked List is a data structure which is in the form of list of nodes, and contains two fields:

  1) Information field (Head): Information field contains actual data.
  2) Next–address field (Tail): Next–address field contains the address of the next node of the list.

- Linked list is a flexible data structure as we can insert any number of nodes and delete nodes from the given linked list.

- **Operations on linked list:**
  1) Insertion of nodes:
  Steps: **I)**– Create an empty node
  II)– Insert data
  III)– Update the next field of the node
  IV)– Update pointer

  2) Deletion of nodes:
  Steps: **I)** – Add one more pointer to the first node.
  II) – Next field of the new pointer is updated with the pointer of first node previously.
  III)– Take data to some variable.
  IV)– Free the node.

  3) Searching:
  Steps: **I)**– Start traversing nodes one by one.
  II) Compare into the field of nodes with the required data.

III) – Return the address of the node containing the data.

- **Types:** **1)** Singly Linked List
  2) Doubly Linked List
  3) Circular Singly Linked List
  4) Circular Doubly Linked List
- Searching time in linked list is O(n) [Worst case]
- **Usage:** **1)** Implementation of Stacks
  2) Implementation of Queues
  3) Representation of graph by adjacency list.
  4) For sparse matrices
  5) Memory allocation

5 Tree

5.1 TREE

Definition

"Tree is a non-linear data structure. A tree is defined recursively as a collection of nodes."

Terminology:

Root:

It is the topmost node of a tree.

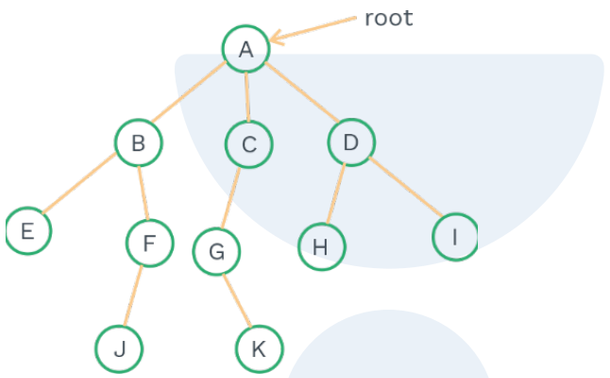


Fig. 5.1 Binary tree

Edge:

An edge refers to the link from parents to children (all links in Fig. 5.1).

Leaf:

A node, which has no children, is known as leaf node.

Example: E, H, J, K and L are all leaf nodes.

Siblings:

Two (or more) children are called to be siblings, if they have same parent node.

Example:

F and E are siblings.

H and I are siblings.

C, B and D are siblings.

Depth:

Depth of a node is the number of links/edges from root to a particular node.

Example: Depth of G = 2.

Level:

- Level = 1+ number of edges between a node and root
  =1+ depth of a node
  where, depth starts from 0
  Example: In Fig. 5.1, B, C and D are on the same level.

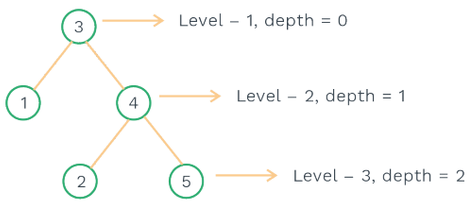


Fig. 5.2

Ancestor and descendant:

- A node 'x' is an ancestor of a node 'y', if there exists a path from the root to 'y', and 'x' appears on the path.
- The node 'y' is called a descendant of 'x'. For example A, C and G are the ancestors for K in Fig. 5.1.

Height:

- Height of a node is the number of edges in the path from that node to its most distant leaf node.
- Height of root node in the below figure = 3.
- In the below example, the height of B is 2 (B–F–J). Height of tree = 3 as shown in below figure.

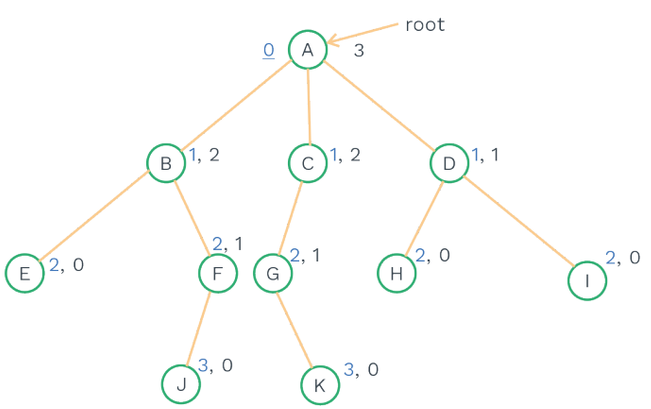


Fig. 5.3

Note:

i) Number in blue represent the depth of the node
ii) Number in black represent the height of the node.

Height of tree:

- Height of tree is the height of root node.
- Height of root node is the number of edges to its the most distant leaf node.

Skew tree:

It is a binary tree, where every node is having one child except for the leaf node.

Fig. 5.4 Skew Tree

Left skew tree:

"It is binary tree, where each node will have only one left child except the leaf node."

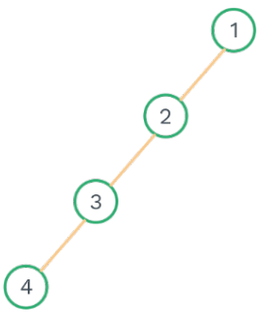


Fig. 5.5 Left Skew Tree

Right skew tree:

"It is binary tree, where each node will have only one right child except the leaf node."

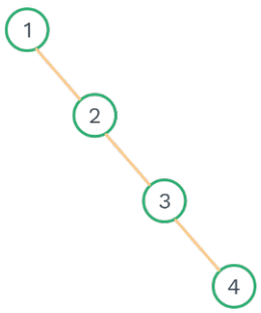


Fig. 5.6 Right Skew Tree

Binary trees:

- Each node in a binary tree can have either 0, 1 or 2 children.
- A binary tree without any node is also a valid binary tree and called as an empty or null binary tree.

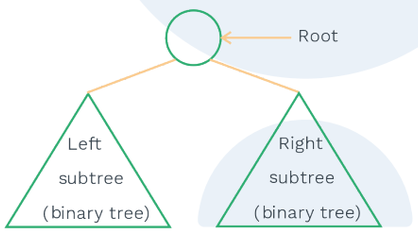


Fig. 5.7 Conceptual Structure of a Binary Tree

Example:

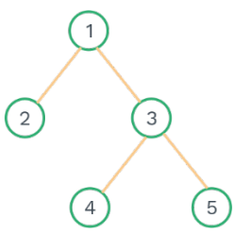


Fig. 5.8

Strict binary tree:

A strict binary tree is a kind of tree, where each node will have either no children or exactly 2 children.

Example:

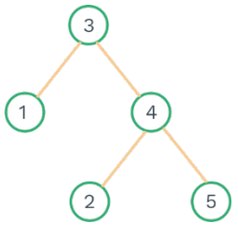


Fig. 5.9 Strict Binary Tree

Full binary tree:

A full binary tree is a binary tree, in which all the internal nodes have exactly two children and all the leaf nodes are present at the last level.

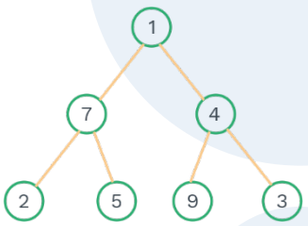
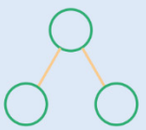
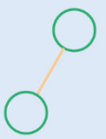
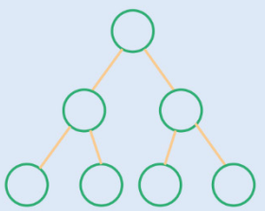
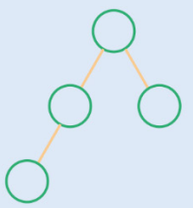


Fig. 5.10 Full Binary Tree

Complete binary tree:

- A complete binary tree is a kind of tree, where except the last level all the other levels are completely filled.

Properties of complete binary trees:

Height of tree	Maximum number of nodes	Minimum number of nodes
h = 0	 1	 1
h = 1	 3	 2
h = 2	 7	 4

In a complete binary tree with n nodes, the number of internal nodes = $\lfloor n / 2 \rfloor$

The maximum number of nodes in a complete binary tree = $2^{(h+1)}-1$.

Properties of full binary tree:

- Number of nodes(n) in a full binary tree of height h = $2^{h+1}-1$.

  at depth = 0, Number of nodes = $2^0 = 1$

  at depth = 1, Number of nodes = $2^1 = 2$

  at depth = 2, Number of nodes = $2^2 = 4$

  at depth h, Number of nodes = 2^h

  $[2^0 + 2^1 + 2^2 + \ldots\ldots\ldots 2^h] = 2^{(h+1)}-1$

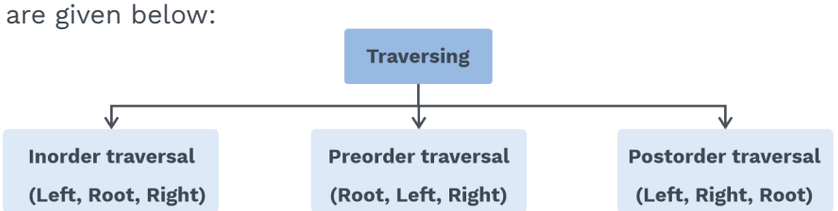
- Number of leaf-nodes in a full binary tree = 2^h.
- Number of non-leaf nodes in a full binary tree = $(2^h)-1$.

Note:

The definition of a Strict binary tree, Full binary tree and complete binary tree varies in different text books but in GATE, the definition will be specified properly in the question.

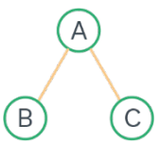
Traversing

- To see the information present in every node, we visit every node, which is called traversing.
- The most popular traversing methods which are applied in binary tree, are given below:



- Generally, they are applicable on the binary tree, but they can be extended to ternary tree or m–way tree.
- Inorder traversal:
  i) Visit left subtree **ii)** Visit root **iii)** Visit right subtree
- Preorder traversal:
  i) Visit root **ii)** Visit left subtree **iii)** Visit right subtree
- Postorder traversal:
  i) Visit left subtree **ii)** Visit right subtree **iii)** Visit root

Example:



Inorder traversal : BAC
Preorder traversal : ABC
Postorder traversal : BCA

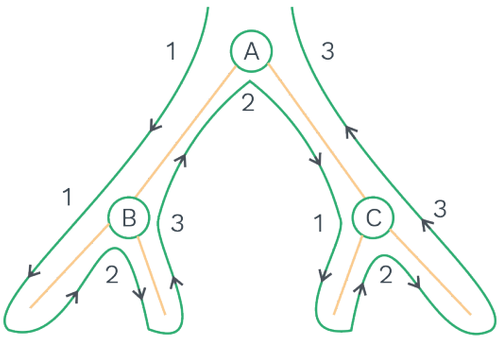
Finding out inorder, preorder and postorder

- For a given tree, if any node (even that node is leaf node) does not have any children, then give them dummy children.
- After that, start traversing on the tree from the top to down, left to right.

Preorder

- To get a preorder of any tree, whenever we visit any node for the 1st time, we print it.

Example:



So, the order, in which we visit the node for 1st time is ABC. So, ABC is preorder of tree.

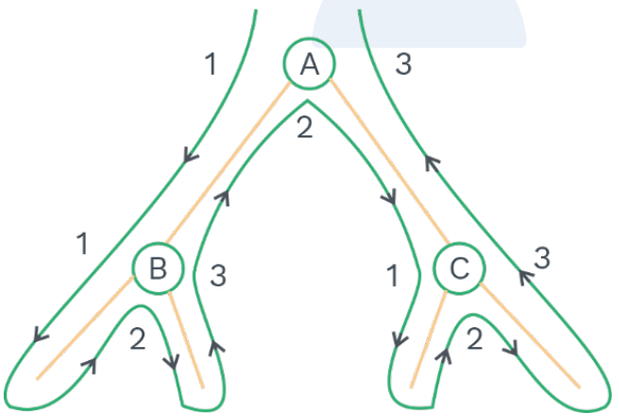
If a node has no child, then its left and right pointers will point to NULL

Note:

Every node will be visited three times before we finish walking entire tree and go back to the root node.

Inorder

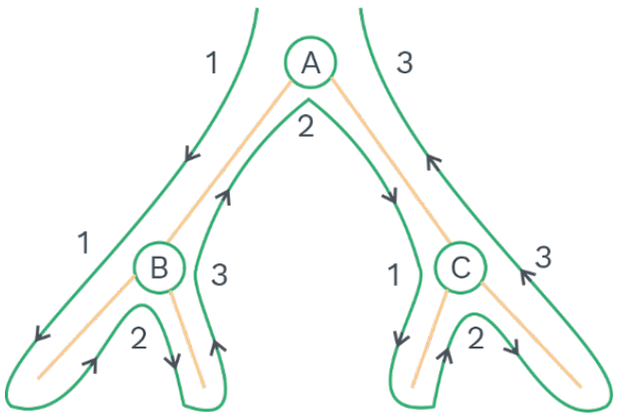
- To get Inorder of any tree whenever we visit any node for the 2nd time, then print it.



- BAC is an order of nodes, in which nodes are visited for the 2nd time. So, BAC is the inorder traversal of tree.

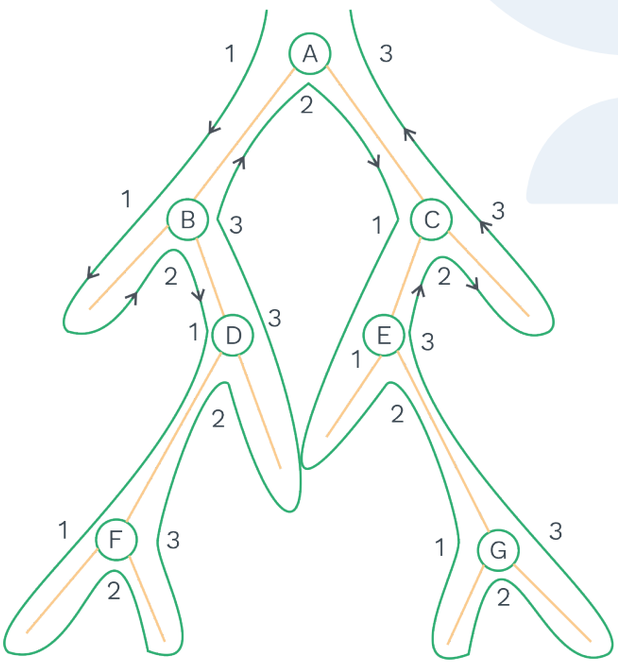
Postorder

- To get postorder of any tree whenever we visit any node for the 3rd time, then print it while walking the tree.



- BCA is an order of nodes, in which nodes are visited for 3rd time. So, BCA → postorder–traversal of tree.

Example:



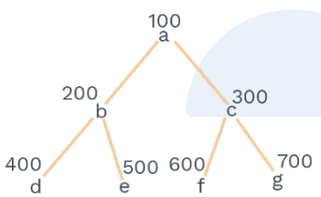
We added dummy nodes for each node, which are not having children.
Preorder traversal: ABDFCEG
Inorder traversal: BFDAEGC
Postorder traversal: FDBGECA

Implementation of inorder traversal

```
Struct node
    {
            char data;
            struct node *left , *right;
    };
    void Inorder (struct node *t)
    {
            if (t)
            {
               Inorder (t → left);
               printf ("%c", t → data);
               Inorder (t → right);
            }
    }
```

Example:

Let the binary tree and address are as shown below:



So, the output is dbeafcg.

Time complexity:

- The best way to analyse the code is to see how many times the codes are visiting the node and time taken by each node at each time.

So, total time complexity = 3 × n × constant time
= O(n)

Space complexity:

- Space complexity depends on the number of levels in the tree, and depending on the level of tree, the stack grows.
- In the worst case, if a tree has 'n' nodes, then n level can be possible, because in worst case, a tree can be a skew–tree.

- So, for every node, there will be a data in the stack.
  Therefore,

Space Complexity : O(n)

Preorder traversal

- Code:

```
                Struct node
                {
                        char data;
                        struct node *left , *right;
                };
void preorder (struct node *t)
    {
        if (t)
            {
                printf ("%c", t → data);
                preorder (t → left);
                preorder (t → right);
            }
        }
```

Similar to Inorder Traversal, in Preorder traversal also, we have to visit each node once. Thus,

Time Complexity = O(n)

In worst case, Tree can be a skewed tree. So, in the worst case, the height of the tree can be n. Therefore,

Space Complexity = O(n)

Postorder traversal

- The code of postorder traversal is shown below:

```
                Struct node
                {
                        char data;
                        struct node *left , *right;
                };
```

```
void postorder (struct node *t)
    {
            if (t)
               {
                   postorder (t → left);
                   postorder (t → right);
                   printf ("%c", t → data);
               }
    }
```

Similarly, in Postorder traversal also, we have to visit each node once. Thus,

Time Complexity = O(n)

In worst case, Tree can be skewed one. So in the worst case, height of the tree can be n. Therefore,

Space Complexity = O(n)

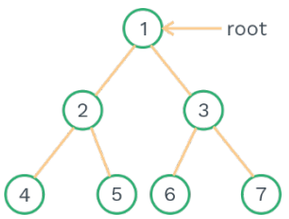
Note:

Space and Time complexity of Inorder, Preorder and Postorder traversals are O(n) only.

Level order traversal:

- The root of the tree as input is given to the algorithm.
- The while loop in the algorithm dequeues the root, prints it and enqueues it's left and right nodes.
- The procedure is repeated until the queue is not empty.

Example:



The order, in which the nodes need to be visited = 1,2,3,4,5,6,7

Note:

Level order traversal of a tree is same as breadth-first traversal for the tree.

Double order

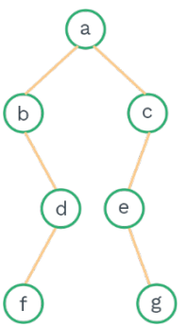
- **1)** In double order traversal, we print each node two times based on code given.

Recursive code:

```
struct node
    {
            char data;
            struct node *left;
            struct node *right;
    };

void DO (struct node *t)
{
  if(t)
  {
      printf ("%c", t → data);
      DO (t → left);
      printf ("%c", t → data);
      DO (t → right);
    }
}
```

Example:



The double order traversal of the above tree is abbdffdaceeggc.

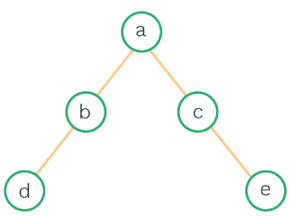
Triple order traversal:

- Triple order traversal means printing every node three times based on below code.

Code:

```
    struct node
    {
            char data;
            struct node *left;
            struct node *right;
    };
void TO(struct node *t)
    {
        if(t)
        {
            printf (" %c", t → data);
            TO(t → left);
            printf ("%c", t → data);
            TO (t → right);
            printf (" %c", t → data);
        }
    }
```

Example:



- Triple order traversal of the above tree is abdddbbacceeeca.

Unlabeled tree:

In an unlabeled tree, the nodes do not have a specific name.

Labeled tree:

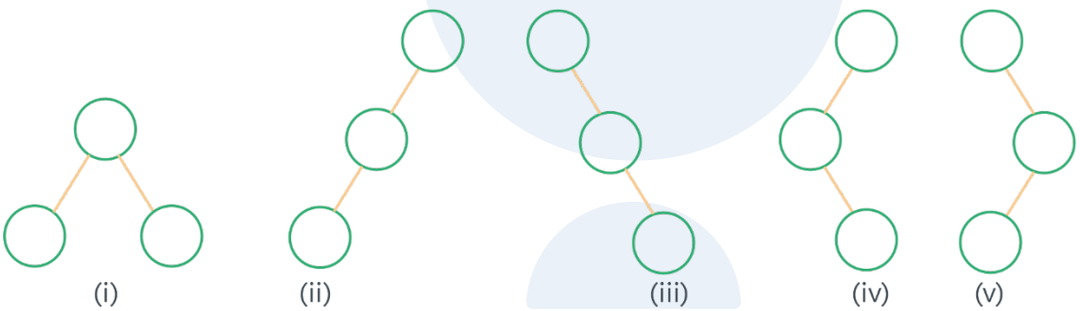
In a labeled tree, the nodes have a specific name.

Number of binary tree:

- With one node, the number of unlabeled binary trees possible is one, and the node itself is a root.

- With two nodes, the number of unlabeled binary trees possible is 2 and are given below:

- With three nodes, the number of unlabeled binary trees possible is 5 and are shown below:

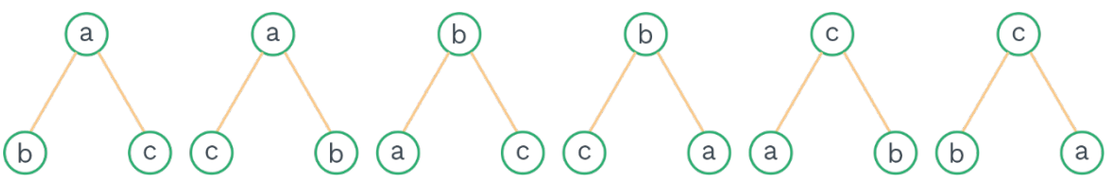


Note:

If we name the node, then the number of trees possible will be even more.

- Number of binary trees possible with 'n' unlabelled nodes is ${}^{2n}C_n / (n+1)$.

  Also known as catlan number.
- Number of binary trees possible with 'n' labelled nodes is $\left[{}^{2n}C_n / (n+1)\right] \times n!$.
- In case of a binary tree having three nodes, each unlabelled tree can be represented in 6 ways in a labelled tree as shown below:
  (Let a, b, c are the names of nodes)



- Every unlabelled tree with three nodes can be labelled in 6 different ways.
- So, the total number of labelled binary trees possible = 5 * 6 = 30.
- Given n nodes, the number of structured (i.e., unlabelled) binary trees possible are ${}^{2n}C_n / (n+1)$, and for any given structure, we can find one tree, which is having the particular preorder or inorder or postorder.

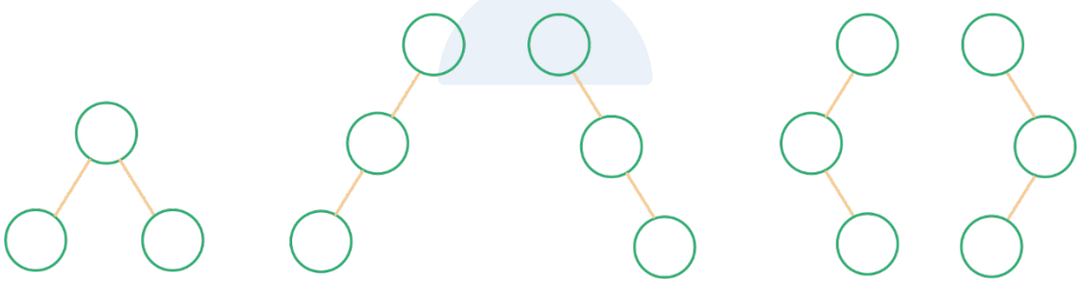
SOLVED EXAMPLES

Q1 **Give all binary trees with three nodes P, Q and R, which have preorder as PQR.**

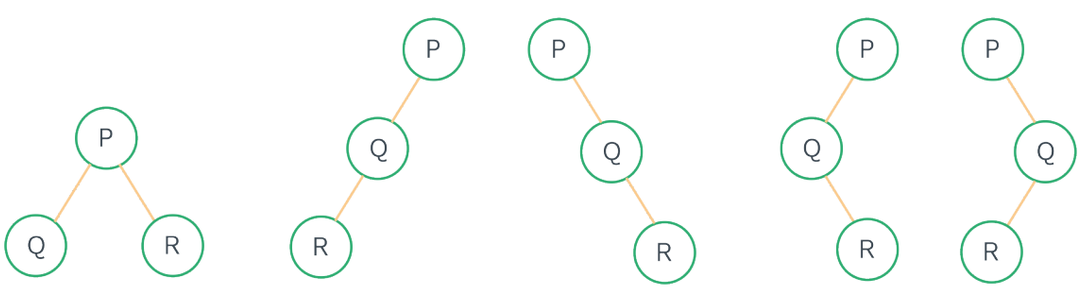
Sol: With three nodes, the number of structures $= \frac{{}^{2n}C_n}{(n+1)}$

$$= \frac{{}^{6}C_3}{4} = \frac{\frac{6!}{3!3!}}{4} = \frac{\frac{6.5.4}{3.2}}{4} = 5.$$

So, five structures are possible, as shown below:



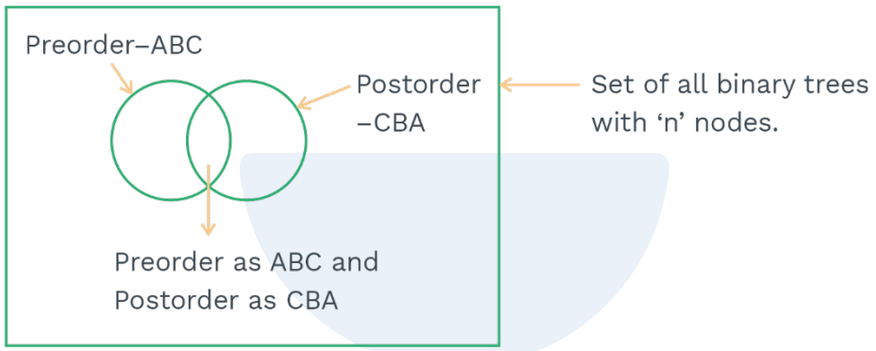
And each structure will represent one particular preorder, i.e., PQR, as shown below:



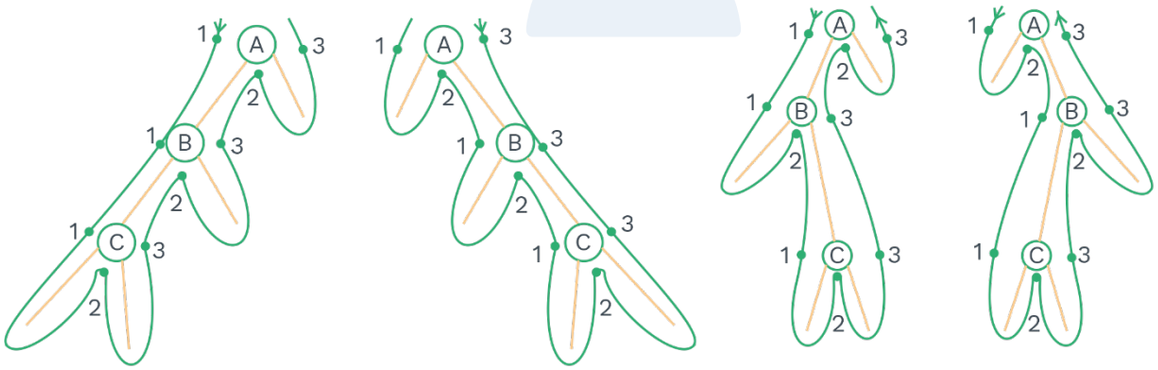
Q2 **What are the number of binary trees possible, if preorder is ABC and postorder is CBA.**

Sol: The number of binary trees possible with three nodes and postorder CBA is also five, because we have five structures possible.

In each structure, there will be exactly one binary tree with CBA as a postorder.



The binary tree with three nodes A, B and C having preorder as ABC and postorder as CBA are given below:



Postorder – CBA
Inorder – CBA
Preorder – ABC

Postorder – CBA
Inorder – ABC
Preorder – ABC

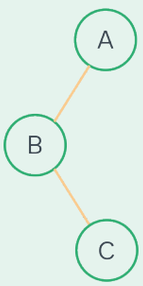
Postorder – CBA
Inorder – BCA
Preorder – ABC

Postorder – CBA
Inorder – ACB
Preorder – ABC

So, the number of a binary tree, which is having preorder ABC and postorder CBA are 4.

Note:

If we apply one more filter as inorder BCA, then we will get only one tree as given below.

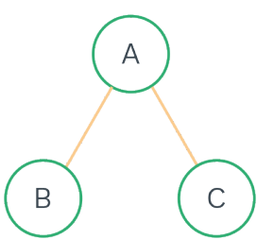


There will always be only one binary tree satisfying all the conditions given specified preorder, postorder and inorder.

Q3 **Let us say, there are three nodes, and three nodes are labelled as A, B and C. The preorder is given as ABC and inorder is given as BAC. Construct the binary tree for the specific order.**

Sol: In preorder, the root will be on the left most side of the given preorder sequence. So, A is the root of the binary tree.

In inorder, the root will be at the middle, and the left subtree will be at the left of the root. Similarly, the right subtree will be at the right of the root.
Hence, the binary tree will look like as shown below:



Preorder : ABC (Root,Left,Right)
Inorder: BAC (Left,Root,Right)

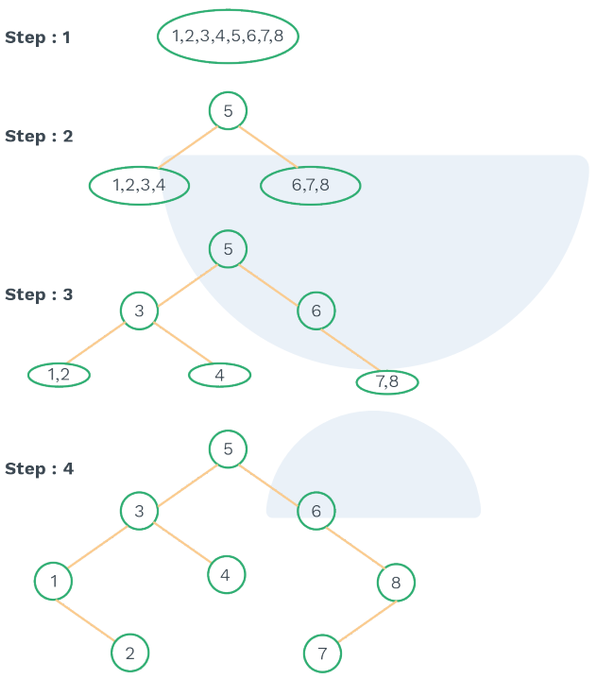
Note:

i) If postorder is given then we get root from right to left.
ii) So, if preorder and inorder are given or postorder and inorder is given, we will be able to construct a unique binary tree.

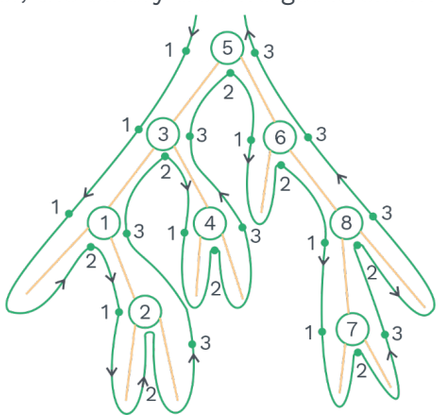
Q4 **Given an inorder as 1, 2, 3, 4, 5, 6, 7, 8 and preorder as 5, 3, 1, 2, 4, 6, 8, 7. Find postorder = ?**

Sol: Preorder: (Root,Left,Right)

Inorder: (Left,Root,Right)
Thus, 5 will be the root.



Using Preorder and Inorder, the binary tree we get is shown below:



The postorder is 2, 1, 4, 3, 7, 8, 6, 5.

Note:

If only one order is given, we will not be able to create a unique binary–tree.

Recursive program to count number of nodes

Suppose 'NN' represents number of nodes.

The recursive equation to count number of nodes is shown below:

NN(T) = 1 + NN(LST) + NN(RST)

Here, LST and RST represent left–subtree and right-subtree, respectively.

Base condition:

NN(T) = 0; when T = 0 (if T is NULL)

Recursive program to count the number of nodes:

```
struct node
{
  int i;
  struct node * left;
  struct node * right;
};
int NN(struct Node *t)
{
  if(t)
    return (1+NN(t → left) + NN(t → right));
  else
    return 0;
}
```

Recursive program to count the number of leaves

Let 'NL' denotes the number of leaf nodes in the tree.

The recursive equation is given below:

NL(T) = 1; if T is leaf

= NL(LST) + NL(RST), otherwise

Program:

```
int NL(struct node *t) {
 if (t == NULL)
    return 0;
 else if (t → left == NULL && t → right == NULL)
    return 1; /* this condition checks whether 't' is leaf or not */
 else
    return (NL(t → left) + NL(t → right));
 }
```

Recursive program to count number of non-leaf

Let 'NNL' denotes the number of non-leaf.
The recursive equation is given below:
NNL(T) = 0, if T is leaf or T is NULL
= 1 + NNL(LST) + NNL(RST), otherwise

Recursive program

```
int NNL(struct node *t) /* Here, root of the tree is passed */
{
  if(t == NULL)
    return 0;
  if (t → left == NULL && t → right == NULL)
    return 0;
  else
    return (1 + NNL(t → left) + NNL(t → right));
}
```

Time complexity:

- Here, every time we visit the node, we are doing constant work.
- Here, three times we are visiting the each node.

So, Time complexity = 3 * C * n = O(n)

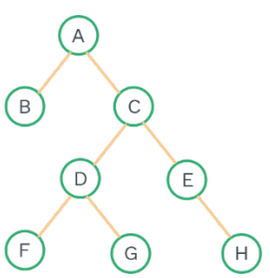
Space complexity

Space complexity depends upon the tree's height, and in the worst-case scenario, tree will be skewed.
Space complexity= O(n)

Full node

- A node having all children is called a full node.
- In a binary tree, a node having exactly two children is called a full node.
- Every full node is non-leaf node, but all non-leaf nodes are not a full node.

Ex:



Leaf nodes are: B, F, G, H
Non–leaf nodes are: A, C, D, E
Full nodes are: A, C, D

Note:

Full nodes will always be a subset of non-leaf nodes.

The recursive equation of the full node is given below:
Let FN denotes full node.
FN(T) = 0; T = NULL
= 0; T is a leaf
= FN(T → LST) + FN(T → RST); if T has only one child.
= FN(T → LST) + FN(T → RST) +1; if T is a full node.
where LST means Left-Subtree and RST means Right-Subtree

Program:

```
int FN(struct node *t)
{
  if (!t)
    return 0;
  if (!t → left && ! t → right)
    return 0;
  return (FN (t → left) + FN (t → right) + (t → left && t → right) ? 1: 0);
}
```

Time Complexity : O(n)
Space Complexity : O(n)

Recursive program to find height of a tree

Let us say, the height of the tree pointed by the pointer T is H(T).

The recursive equation is:

$$H(T) = \begin{cases} 0; \text{if T is empty} \\ 0; \text{if T is leaf} \\ 1 + \max\left(H(LST), H(RST)\right); \text{otherwise} \end{cases}$$

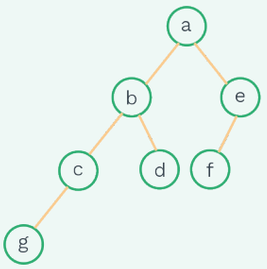
Program:

```
int H(struct node *t)
{
 if (!t && (!t → left && !t → right))
     return 0;
 else
      return(1+ ((H(t → left) > H(t → right)) ?
H(t → left): H (t → right);
}
```

Time Complexity : O(n)
Space Complexity : O(n)

Previous Years' Question

Which of the following sequences denotes the post order traversal sequence of the below tree?



a) fegcdba **b)** gcbdafe
c) gcdbfea **d)** fedgcba
Sol. c) **(GATE: 1996)**

Previous Years' Question

An array X of n distinct integers is interpreted as a complete binary tree. The index of the first element of the array is 0. The index of the parent of element X[i], i ≠ 0, is?

a) $\left\lfloor \frac{i}{2} \right\rfloor$ **b)** $\left\lceil \frac{i-1}{2} \right\rceil$ **c)** $\left\lceil \frac{i}{2} \right\rceil$ **d)** $\left\lceil \frac{i}{2} \right\rceil - 1$

Sol: d) **(GATE: 2006)**

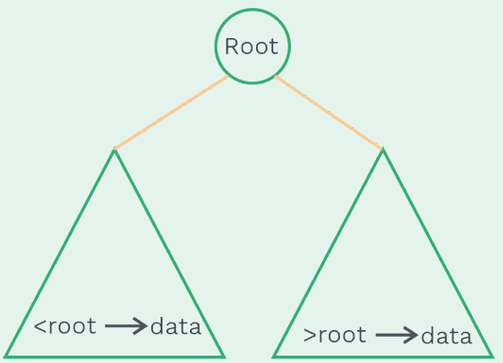
Binary search trees (BSTs)

- BSTs are useful for searching.
- Binary Search Tree Property:- For all nodes, the value of a node should be greater than all nodes in left subtree and smaller than all nodes in right subtree.

Note:

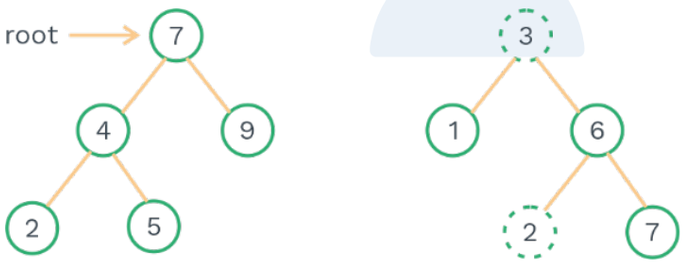
i) The above-mentioned property needs to be satisfied by every node present in the tree.

ii) Both the left-subtree and right-subtree have to be BSTs.



Example:

In the below diagram, the tree on left is a BST, but the right one is not a BST. As node 2 is smaller than 3, but is present in the right subtree of node 3. Thus, it is violating the property of BST.



Declaration of BST

- BST is a binary tree whose in-order traversal gives a sorted order of elements.

```
struct BST_Node
{
    int data;
    struct BST_Node *left;
    struct BST_Node *right;
};
```

Operation on binary search trees

Some of the operations supported by BSTs are

- Find Minimum element
- Find Maximum element
- Insertion in BST
- Deletion in BST
- Searching an element

Note:

- Since root value is always in between left subtree data's and right subtree data's, performing an Inorder traversal on BST produces a sorted list in an increasing order.
- Searching of a key element:
  if key_element < root → data, then search only in the left subtree.
  if key_element > root → data, then search only in the right subtree.
  if key_element == root → data then key_element is found
  In worst case, searching time in Binary Search Tree will be O(n).

Finding an element in binary search trees

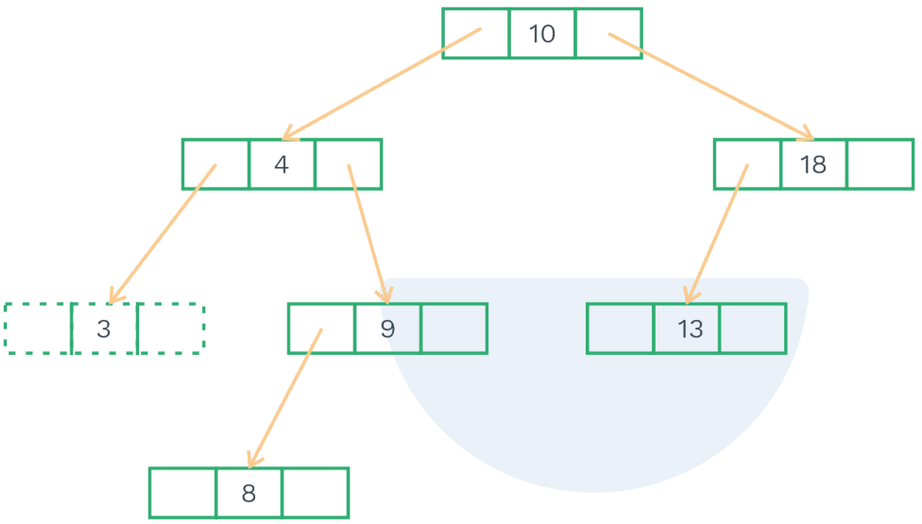
- First, start traversing from root node, BST-property is used to find any element in BST.
- If the value of the element, which we want to search is less than the value of root-node, then search only in the left sub-tree.
- If the value of the element, which we want to search is greater than the value of root-node, then search only in the right sub-tree.

```
struct BST_Node *Find(struct BST_Node * root, int data)
{
  if(root == NULL)
     return NULL;
  if(data < root → data)
     return (Find(root → left, data));
  else if(data > root → data)
     return (Find (root → right, data));
  else if(root->data ==data)
     return root;
}
```

Time Complexity : O(n), in the worst case (when BST is a skewtree).
Space Complexity : O(n), for a recursive stack, in the worst case.

Finding a minimum element in binary search trees

- In BSTs, the minimum element is the leftmost node, which is either a leaf node or a node with a right child but not with a left child.
- In the below example, the minimum element is 3.



Time Complexity : O(n), worst case (when BST is a left skew tree).
Space Complexity : O(n) for a recursive function stack.

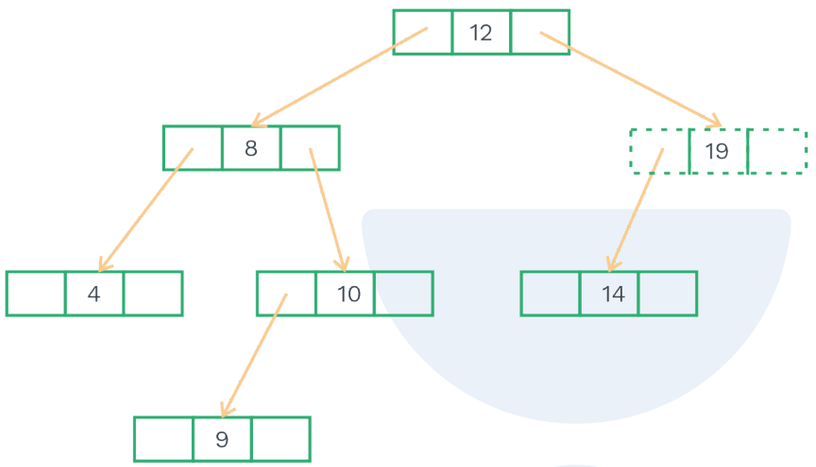
- Non-recursive way to explain the above algorithm:

```
struct Node *Minimum(struct Node *Root)
{
    if(Root==NULL)
       return NULL;
    while(Root → l_child !=NULL)
      Root=Root → l_child;
    return Root; }
```

Time Complexity : O(n)
Space Complexity : O(1)

Finding a maximum element in binary search trees

- The maximum element in a BST is the rightmost node, i.e., the right most leaf node in right subtree.
  (OR)
  A node in the right subtree having a left child but not having a right child.
- The maximum element in the below shown BST = 19.




```
struct Node *Maximum(struct Node *Root)
{
  if(Root==NULL)
     return NULL;
  else if(Root → r_child==NULL)
     return root;
  else
     return Maximum(root → r_child);
}
```

Time Complexity : O(n), worst case (when BST is a right skew tree).
Space Complexity : O(n) for a recursive stack.

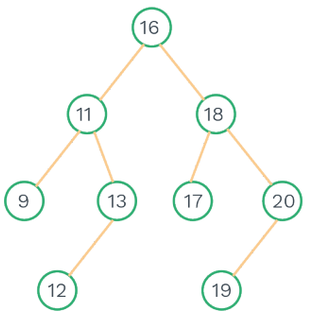
Inorder predecessor:

In BST, Inorder predecessor is the greatest element in the left subtree of a node.

Inorder successor:

In BST, Inorder successor is the smallest element in the right subtree of a node.

eg:

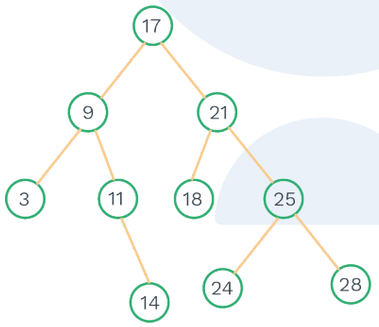


Inorder predecessor (16) = 13

Inorder successor (16) = 17

Insertion in BST

- We have to first search for the location to insert an element (node) into a Binary-Search-Tree.
- Find operation is used to find location of data to insert.
- **eg:** Insert an element 12 into the BST.



Search for element y = 12
12 < 17, go to the left subtree of 17
12 > 9, go to the right subtree of 9
12 > 11, go to the right subtree of 11
12 < 14, go to the left side of 14 and insert 12.

Time Complexity : O(n) ; when Tree is skewed Tree
Space Complexity : O(n) for recursive stack.
Space Complexity = O(1) ; for the iterative code.

Deletion in BST

- If a node to be deleted is a leaf node, then simply delete it. Deleting a non-leaf node is a complex task as other nodes may need to be shuffled after deletion. The process of deletion can be seen below:
- First search the location of the node to be deleted.

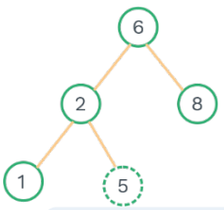
- Following are the cases to delete a node in BST:

For leaf node:

- To delete the node having no child, simply deallocate the memory and add the NULL pointer to its parent.

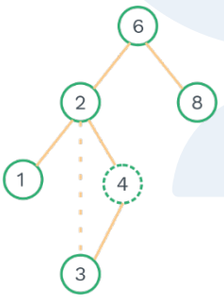
Example:

- If we have to delete a node 5, just delete it.



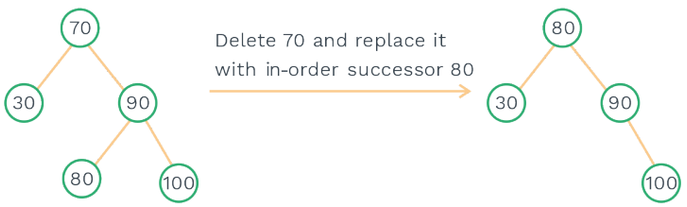
Node having one child:

- When we have to delete the node having only one child.
- Delete that node and point the node's parent-pointer to its children.



Node having two children:

When we have to delete a node having two children, delete that node and replace that with either Inorder successor (OR) Inorder predecessor.



Time Complexity : O(n), in worst case, when Tree is skewed one.
Time Complexity: (logn), in best case.

Previous Years' Question

A program takes as input a balanced binary search tree with n leaf nodes and computes the value of a function g(x) for each node x. If the cost of computing g(x) is:

$$\min\begin{pmatrix}\text{number of leaf – nodes}\\ \text{in left – subtree of x}\end{pmatrix}\begin{pmatrix}\text{number of leaf – nodes}\\ \text{, in right – subtree of x}\end{pmatrix}$$

Then the worst–case time complexity of the program is?

a) $\theta(n)$ **b)** $\theta(n\log n)$ **c)** $\theta(n^2)$ **d)** $\theta(n^2\log n)$

Sol: b) **(GATE: 2004)**

Previous Years' Question

Suppose that we have numbers between 1 and 100 in a binary search tree and want to search for the number 55. Which of the following sequences CANNOT be the sequence of nodes examined?

a) {10, 75, 64, 43, 60, 57, 55}
b) {90, 12, 68, 34, 62, 45, 55}
c) {9, 85, 47, 68, 43, 57, 55}
d) {79, 14, 72, 56, 16, 53, 55}

Sol: c) **(GATE: 2006)**

Previous Years' Question

A data structure is required for storing a set of integers such that each of the following operations can be done in O(logn) time, where n is the number of elements in the set.
i) Deletion of the smallest element
II) Insertion of an element if it is not already present in the set

Which of the following data structures can be used for this purpose?
a) A heap can be used but not a balanced binary search tree.
b) A balanced binary search tree can be used but not a heap.
c) Both balanced binary search tree and heap can be used.
d) Neither balanced search tree nor heap can be used.
Sol: b) **(GATE: 2003)**

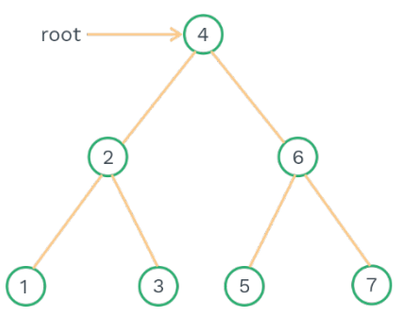
Balanced binary search trees

- Worst case time required for searching in a BST in O(n).
- It occurs when BST is either left-skewed or right-skewed.
- The worst-case time complexity can be reduced to O(logn) using the concept of balanced binary search tree.
- Suppose H(K) represents the height of a balanced-tree.
  Here, K=balance factor
  where Balance factor=Height of left subtree-Height of right subtree.

Full balanced binary search trees

- A binary search tree is known to be full balanced BSTs, if balance factor of each node = 0.

For example:



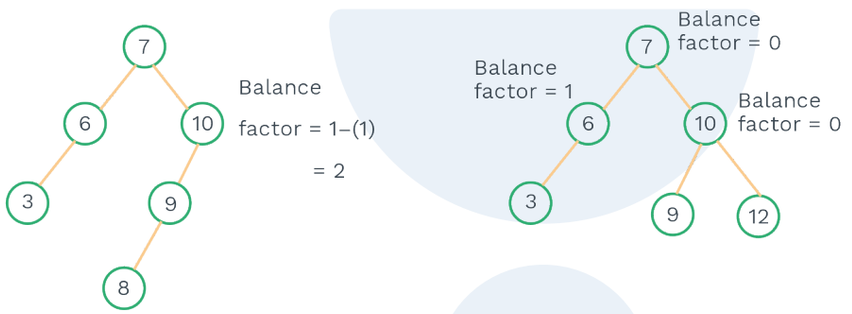
AVL (Adelson–Velskil and Landis) trees

Definition

It is a height balanced binary search tree.
For each node K, the height of the left and right subtrees of K differ by at most 1. It means the balance factor of an AVL tree should be (-1,0,1).

What is balance factor?

It is the difference between heights of left sub tree and right sub tree. It means Balance factor(K) = Height of LST(K)-Height of RST(K).



NOT an AVL Tree as balance factor of each node is not following AVL – property.

An AVL Tree each node is following AVL–property. Balance factor of each node is either –1, 0 or 1.

Note:

Height of a Null Tree is –1.

Minimum/maximum number of nodes in AVL tree

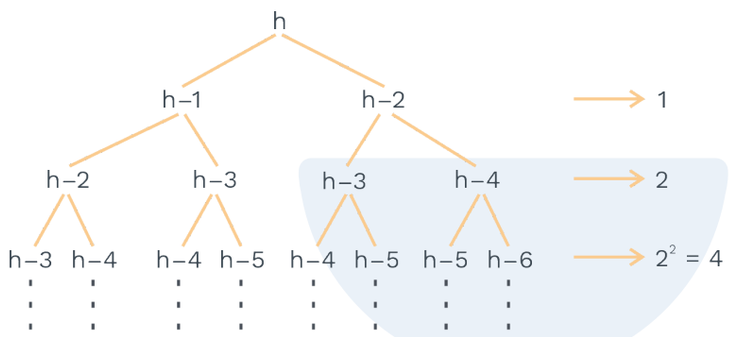
- Suppose N(h) denotes the number of nodes present in the AVL tree, where h denotes the height of an AVL tree.
- For minimum number of nodes having height=h, left subtree of an AVL tree should be of height (h-1) and right subtree should be of height (h-2). (OR)
- The left subtree of an AVL tree should be of height (h-2) and right subtree should be of height (h-1).

- Therefore, the recursive equation for minimum number of nodes of height h:

  $N(h) = N(h-1) + N(h-2)+1$

- Where N(h-1)= minimum number of nodes of height (h-1)
- N(h-2)= minimum number of nodes of height (h-2)

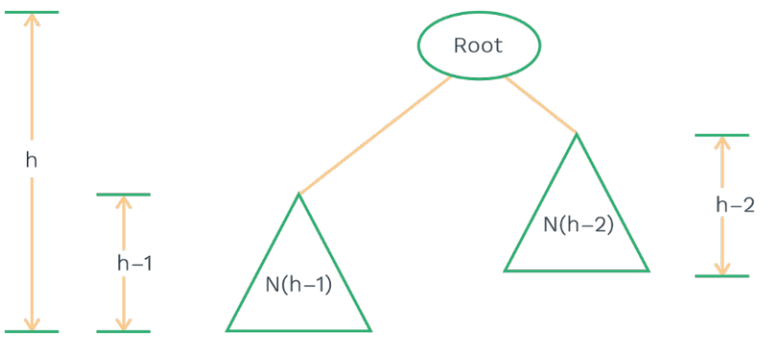
From the above equation:



$$n(h) = 2^0 + 2^1 + 2^2 + \ldots\ldots\ldots\ldots\ldots\ldots\ldots + 2^{h-1}$$

$$n(h) = 2^h - 1$$

$$h = \theta(\log_2 n)$$



So, the maximum height of an AVL tree = O(logn)
For maximum number of nodes, the recursive equation, we get:
N(h)=N(h-1)+N(h-1) +1
=2N(h-1) +1
It is a case of full binary tree.
After solving the above recurrence relation, we will get:
height h = O(log n)

Note:

All above cases will lead to height of an AVL = O(logn), where n denotes the number of nodes.

AVL tree declaration

```
struct AT
{
  int d;
  struct AT *Lt ,*Rt;
  int h;
};
```

In the above AVL Tree Declaration, AT represents AVL Tree, d represents data, *Lt represents left pointer, *Rt represents right pointer and h represents height.

Height of an AVL tree

```
int h(struct AT *root)
{
  if(root==Null)
    return -1;
  else
    return root → h;
}
```

Time Complexity : O(1).

Rotation:

- When we insert an element to an AVL tree or delete an element from an AVL tree, the structure of tree will likely to get changed.
- As insertion/deletion of an element can result in an increase/decrease of the height of subtree by 1.
- To correct it, we need to check the balance factor of each node after every insertion/deletion of an element in an AVL-Tree and then rotation of tree is required.
- There are mainly two types of rotations to restore the properties of AVL-Tree:
  Single Rotation and Double Rotation.

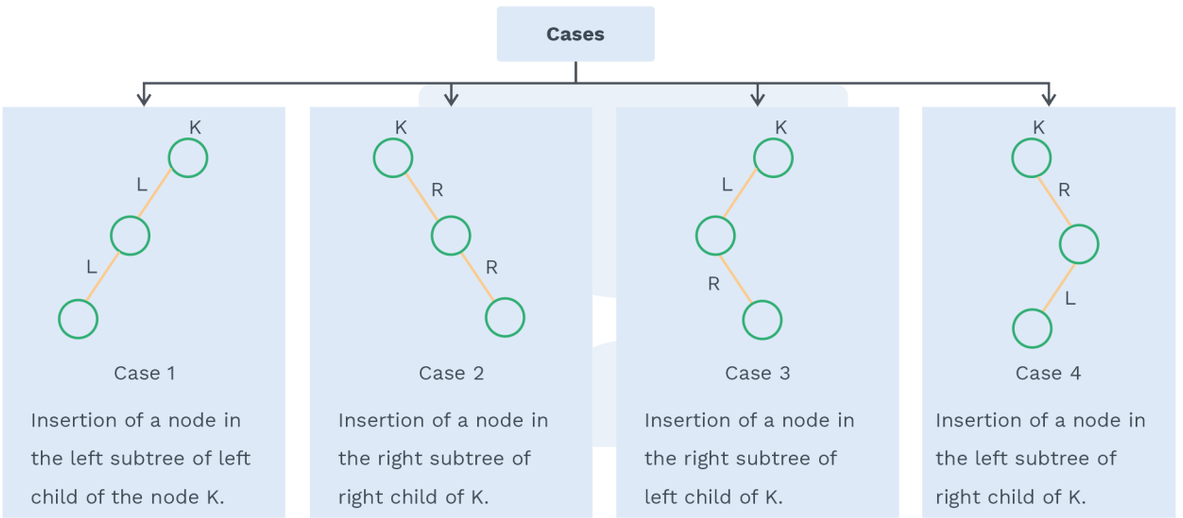
Observation

- When we insert a new node in an AVL tree, balance factor of those nodes get affected, which are present in the path from root to the insertion-point.

- So, after insertion of a new node, we need to check the balance factor of 1st node present in the path from insertion-point to root and so on.
- We need to restore the AVL-Tree property according to the type of violation.

Types of violation

- What type of violation can occur?
- Suppose K is the node, where the imbalance occurs due to different type of insertion.
- There are four cases possible:

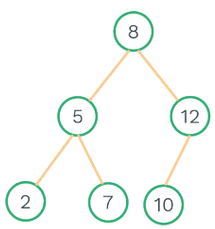


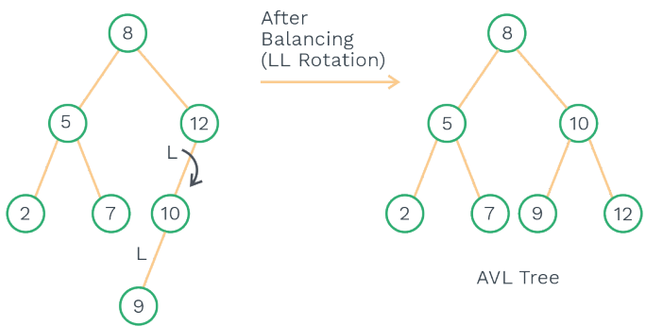
Balancing of a tree using rotation

Single rotations

Left left rotation (LL rotation):

- Suppose a new node 9 is inserted into the given AVL tree, an imbalance occurs as balance factor of some nodes get changed. We need to perform LL-Rotation in this case.
- Given AVL tree

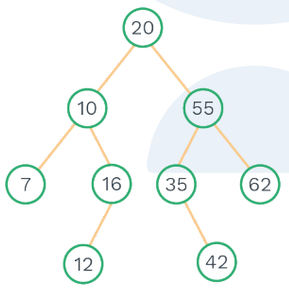




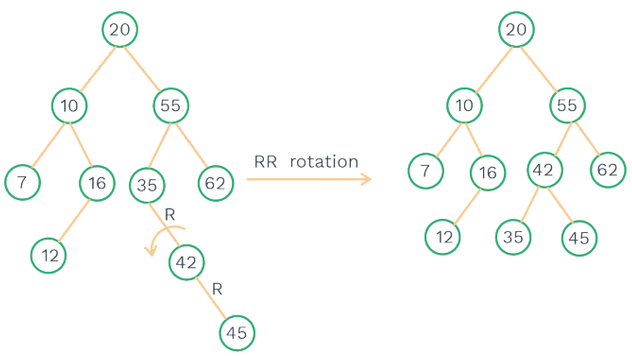
Time complexity: O(1)
Space complexity: O(1)

Right right rotation (RR rotation)

- When a new node 45 is inserted into the given AVL tree, an imbalance occurs, as balance factor of some nodes get changed.
  Given AVL Tree



Example:



Time complexity: O(1)
Space complexity: O(1)

Double rotation

Sometimes, Double rotation is required to restore the AVL-tree properties as single rotation is not enough to solve it.

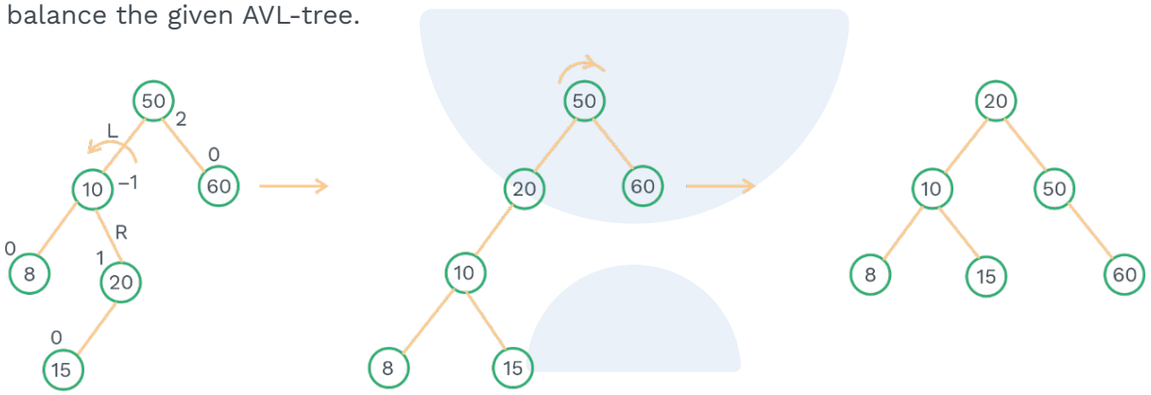
Left right rotation (LR rotation)

- LR Rotation is required when insertion occurs at the right subtree of the left child of a node.

Example

When a new node 15 is inserted, Imbalancing of AVL-tree occurs.

Here, we have to perform LR Rotation(double rotation) to balance the given AVL-tree.

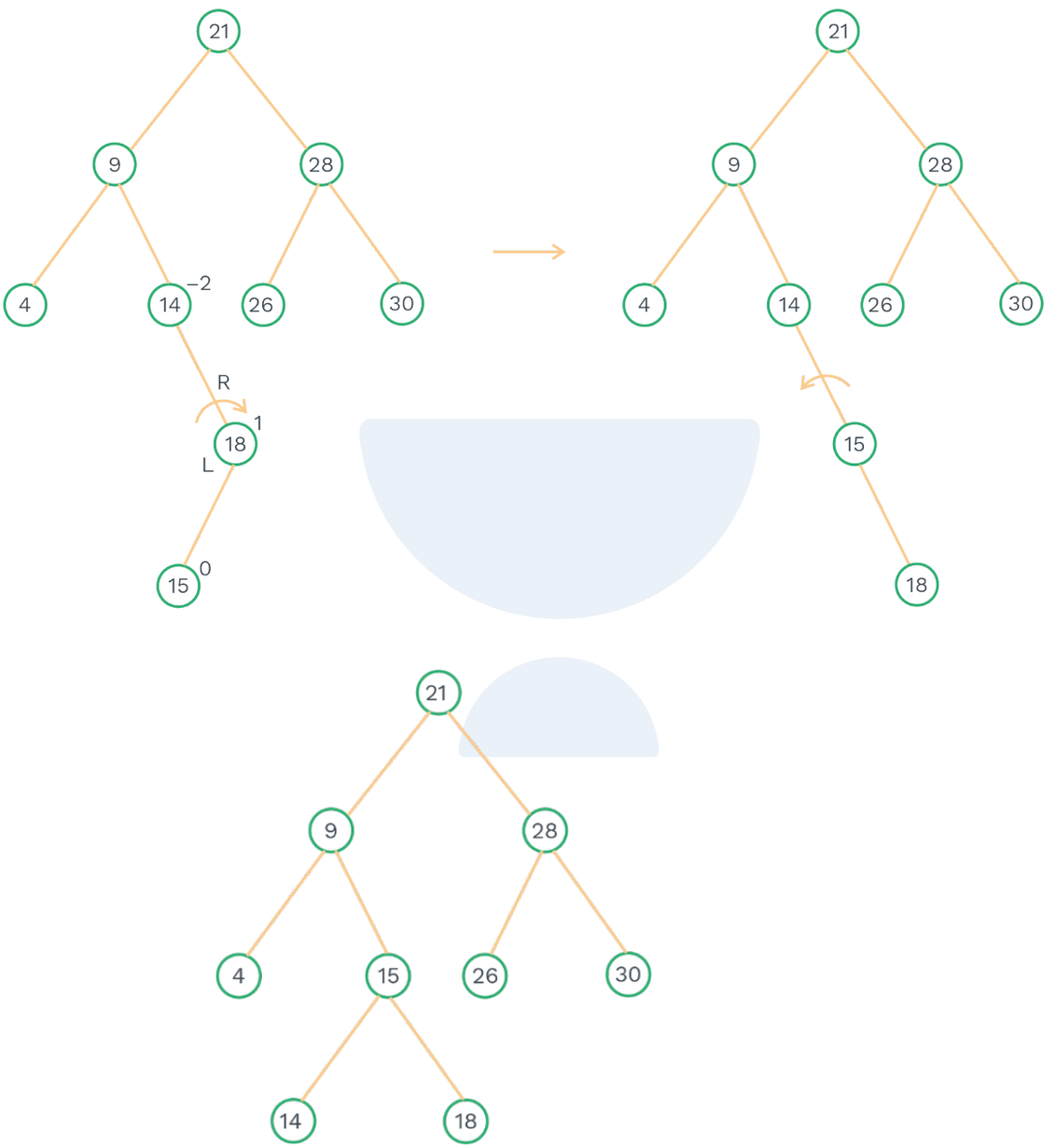


Right left rotation (RL rotation)

- RL Rotation is required when insertion occurs at the left subtree of the right child of a node.

Example:

- When a new node 15 is inserted, Imbalancing of AVL-tree occurs.
- Here, we have to perform RL Rotation(double rotation) to balance the given AVL-tree



Time and space complexity of insertion in an AVL-tree:

Time Complexity = O(log n)

Note:

i) In red-black tree and 2-3 tree an element can be searched in O (log n) time

ii) If "I" is the number of internal nodes of a complete n-ary tree, Then, the number of leaves present in it can be represented by I(n–1) + 1.

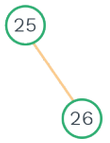
iii) So, if we have 'n' internal nodes of degree 2 in a binary tree, then we have (n+1) leaf nodes.

SOLVED EXAMPLES

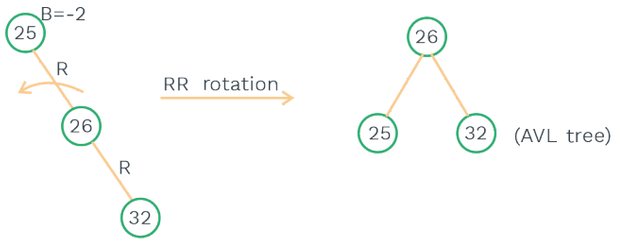
Q1 **Construct an AVL tree for the given sequence**
25, 26, 32, 8, 5, 13, 29, 17, 15

Sol: Step 1: Insert 25

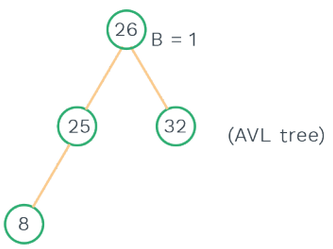
Step 2: Insert 26



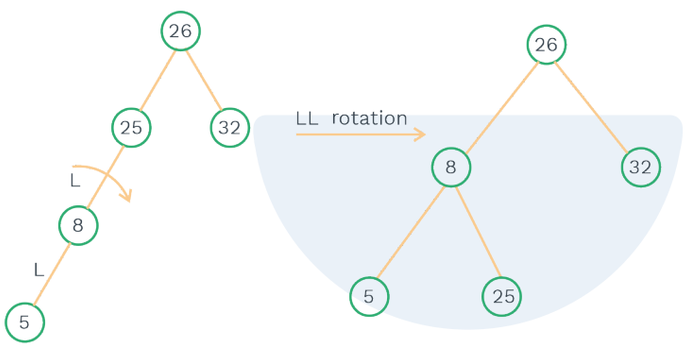
Step 3: Insert 32



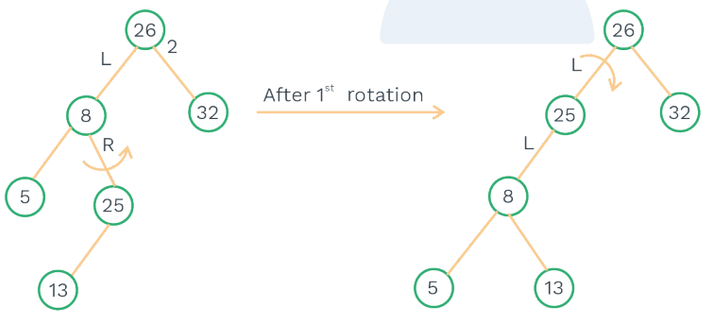
Step 4: Insert 8

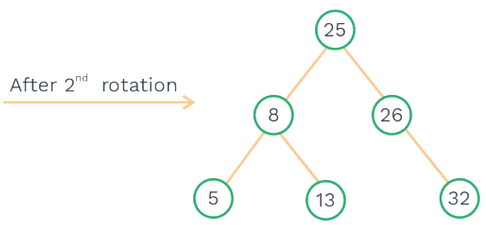


Step 5: Insert 5

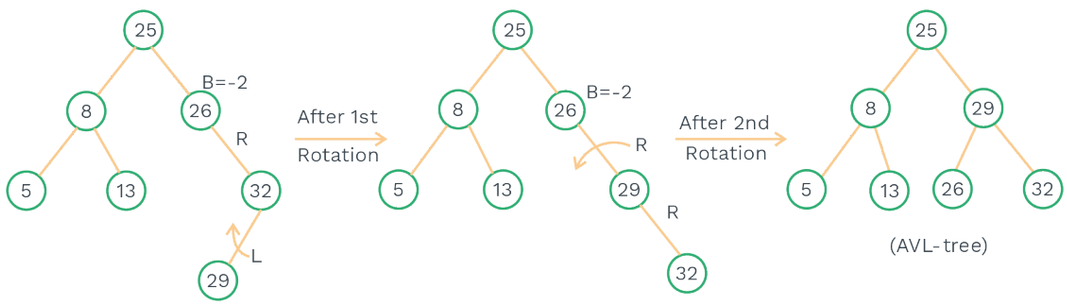


Step 6: Insert 13

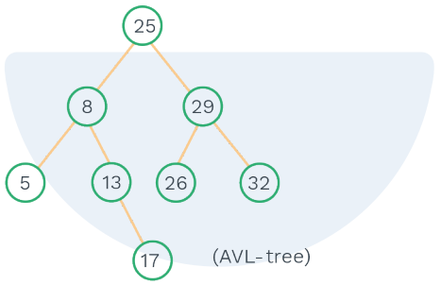




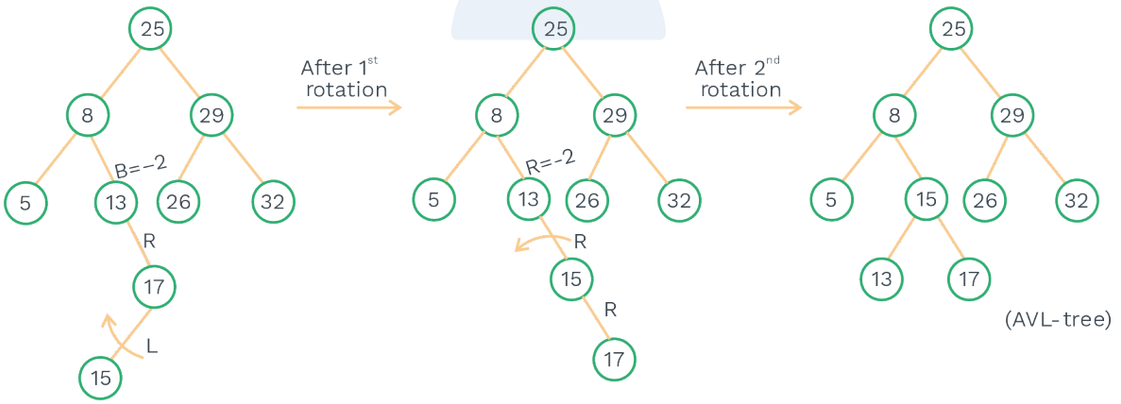
Step 7: Insert 29



Step 8 : Insert 17



Step 9 : Insert 15



Previous Years' Question

Which of the following is TRUE?

a) The cost of searching an AVL tree is θ(logn), but that of a binary search tree is O(n).

b) The cost of searching an AVL tree is θ(logn), but that of a complete binary tree is θ(nlogn).

c) The cost of searching a binary search tree is O(logn), but that of an AVL tree is θ(n).

d) The cost of searching an AVL tree is θ(nlogn), but that of a binary search tree is O(n).

Sol: a) **(GATE: 2008)**

Heap

- Heap is a nearly complete binary tree.
- It means, all the leaf nodes must be either at level h or (h-1).
- The main property of a heap is that the value of each parent node should be either: less than equal to its children(min-heap)
  OR
  greater than equal to the value of its children(max-heap).
- Heap takes less time for inserting an element, for finding minimum in case of min-heap, for finding maximum element in case of max-heap, for deleting an element etc.
- The time complexity details of different data-structures for different operations are as shown below:

Data structure	Insertion	Search	Find Min.	Delete Min.
Unsorted array	O(1)	O(n)	O(n)	O(n)
Sorted array (increasing order)	O(n)	O(log n)	O(1)	O(n)
Unsorted linked list	O(1)	O(n)	O(n)	O(n)
Min-heap	O(log n)	O(n)	O(1)	O(log n)

Unsorted array:

- **Insertion:** In an Unsorted array, insertion can happen at the end of the array, so it will take O(1) as time complexity.
- **Seaching:** In an unsorted array, to search for an element, we need to visit all the elements in the array, i.e., linear search. Hence, its time complexity is O(n).
- **Finding minimum element:** Similarly, to find the minimum element in an unsorted array, we might need to visit all the elements in the array in worst case, i.e., linear search. Hence, its time complexity is O(n).
- **Deletion of minimum element:** To delete a minimum element in an unsorted array, we first have to find the minimum element, which we want to delete and then after deleting that element, remaining (n-1) elements need to be moved in worst case given n number of elements in an unsorted array.

  Total complexity = O(n) +O(n) =O(n).

  Sorted array (Let 'n' be number of elements in increasing order)
- **Insertion:** Insertion takes O(n) time. Since it is a sorted array, we need to place the element in its correct position. After that, in the worst case, we need to move the remaining (n–1) elements, so, the overall time complexity is O(n).
- **Seaching:** Since it is a sorted array, we can apply binary search, and it will take time complexity as O(logn).
- **Finding minimum element:** Since the sorted array is in an increasing order the 1st element is the minimum element. Hence, O(1) is the time complexity to find a minimum element.
- **Deletion of minimum element:** Now, if we want to delete the minimum element, we need to move remaining (n–1) element ahead. Hence the time complexity is O(n).

Unsorted linked list (having 'n' element)

- Insertion: Insertion can happen in O(1), because it is an unsorted linked list.
- Searching: To search an element, we need to visit every element of a list. So, In worst case its time complexity is O(n).
- Finding minimum element: To find the minimum element, we need to visit every element of the list in worst case, as linked list is unsorted. Hence, its time complexity is O(n).
- Deletion of minimum element: To delete minimum, finding minimum takes O(n), and deleting it will take O(1) time.

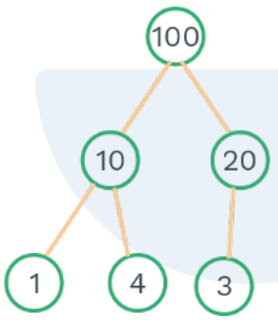
Min-heap, (having 'n' element)

- The time taken by min heap to insert and delete an element in min heap =O(log n).

- In min heap, first element (root) is the smallest element; thus, the time taken by min-heap to find a minimum element = O(1)
- In the worst case, searching an element in a heap requires to visit all the nodes once.
  So, time complexity= O(n)

Max-heap:

Definition: Max-heap is a complete binary tree, where the root value is maximum. It means, the root element must be greater than all the elements present in the left subtree and right subtree.

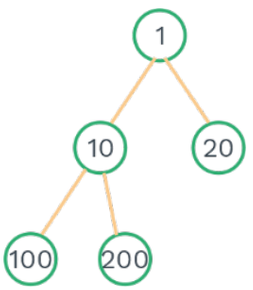


Min-heap:

Definition: Min–heap is a complete binary tree, where the root value must be smallest.

- It means the root element should be smaller than all the elements present in the left subtree and right subtree.

Example:



Array representation of a complete binary tree

- Heap can be represented using the arrays.
- As we know, Heap is nearly a complete binary tree. This method to represent the heap does not waste any space(location).

- Suppose all the elements are stored in array starting from index 0.
- In this case, the representation of the previous given heap can be given as:

1	10	20	100	200
0	1	2	3	4

- The index of a parent for a child at index 'i' is $\lceil i / 2 \rceil - 1$.
- Left child of the ith element is at 2*i + 1.
- Right child of the ith element is at 2(i+1).
- Suppose the index of the array starts at 1, then
- The index of a parent for a child at index 'i' is $\left\lfloor \frac{i}{2} \right\rfloor$.
- Left child of the ith element is 2*i.
- Right child of the ith element is 2*i+1.

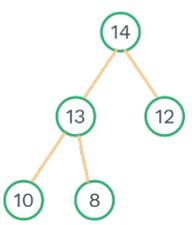
Note:

Here, multiplication with 2 indicates the left shift, and division with 2 implies the right shift. It is easy to implement like this.

SOLVED EXAMPLES

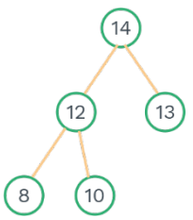
Q1 **Given the array of element check whether it is binary max–heap or not?**
a) 14, 13, 12, 10, 8
b) 14, 12, 13, 8, 10
c) 14, 13, 8, 12, 10
d) 14, 13, 12, 8, 10
e) 89, 19, 40, 17, 12, 10, 2, 5, 7, 11, 6, 9, 70

Sol: **a)**



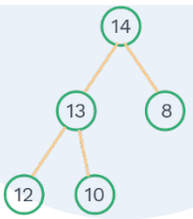
It is a binary max-heap, because all the nodes are satisfying the heap property. Here, A.heapsize is 5.

b)



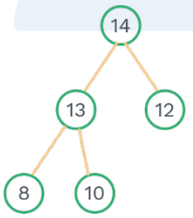
This is also a binary max-heap, because all the nodes are satisfying the heap property. Here, A.heapsize is 5.

c)



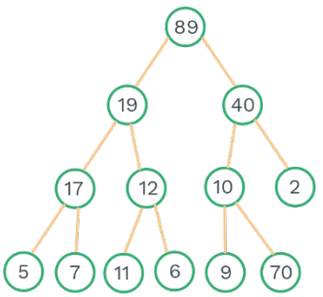
This is also a binary max-heap, because all the nodes are satisfying the heap property. Here, A.heapsize is 5.

d)



This is also a binary max-heap, because all the nodes are satisfying the heap property. Here, A.heapsize is 5.

e)



This is not a max-heap, because 70 is larger than the parent node, which is 10.

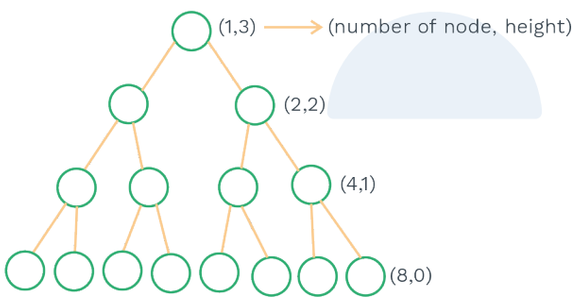
Note:

- Every leaf is a heap.
- For a given set of numbers, there may be more than one heap.
- If an array is in decreasing order, then it is always a max–heap.
- If the array is in ascending order, then it is always a min–heap.
- The starting index of leaf nodes of a complete binary tree having 'n' nodes is, $\left(\left\lfloor \frac{n}{2} \right\rfloor + 1 \text{ to } n\right)$ and the number of non–leaf is $\left(1 \text{ to } \left\lfloor \frac{n}{2} \right\rfloor\right)$.

Note:

Maximum number of nodes at a given level of height 'h' of complete binary tree (having n number of nodes) is $\left\lceil \frac{n}{2^{h+1}} \right\rceil$.

Example:



Maximum number of nodes at height = 0

$$= \left\lceil \frac{n}{2^{h+1}} \right\rceil$$

$$= \left\lceil \frac{15}{2^{0+1}} \right\rceil == \left\lceil \frac{15}{2} \right\rceil = 8$$

So, at most eight nodes will be at height '0' in this particular tree.

Heapifying an element

- Sometimes, when we insert an element into a heap, it might violate the property of heap.

- Heapifying is the process of maintaining the heap property (i.e., every node satisfies the property of the heap according to the max(min) heap).
- For example, to heapify an element in a max-heap:
- First, find its child with maximum value and swap it with the current node.
- Continue this process as long as all the nodes satisfy the heap-property.
- MAX-HEAPIFY(A, i), where A is the heap and 'i' is the index of element, where insertion happens.
- This function can be applied, only if the left subtree and the right subtree tree are to be max-heap.

```
Max-Heapify (A, i)
{
    l = 2i; // 'l' indicates left child index
    r = 2i + 1; // 'r' indicates right child index
    if (l ≤ A.heapsize and A[l] > A[i])
            largest = l;
    else
            largest = i;
    if (r <= A.heapsize and A[r] > A[largest])
            largest = r;
    if (largest ≠ i)
            exchange A[i] with A[largest]
            Max-Heapify (A, largest)
}
```

- At each step, the largest of the elements A[i], A[LEFT(i)] and A[RIGHT(i)] are determined, and its index is stored in largest.
- If A[i] is largest, then the subtree rooted at node i is already a max-heap, and the procedure terminates.
- Otherwise, one of the two children has the largest element, and A[i] is swapped with A[largest], which causes node i and its children to satisfy the max–heap property.
- The node indexed by largest, however, now has the original value A[i], and thus the subtree rooted at largest might violate the max-heap property.
- Consequently, we call MAX-HEAPIFY recursively on that subtree.
- The running time of MAX-HEAPIFY on a subtree of size n rooted at a given node i is the q(1) time to fix up the relationships among the elements A[i], A[LEFT(i)], and A[RIGHT(i)], plus the time to run MAX-HEAPIFY on a subtree rooted at one of the children of node i (assuming that the recursive call occurs).

- The children's subtrees have a size at most 2n/3. The worst case occurs when the bottom level of the tree is exactly half full, and therefore, we can describe the running time of MAX-HEAPIFY by the recurrence

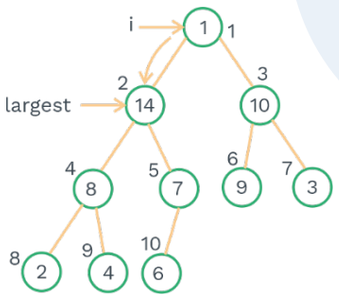
$$T(n) \leq T(2n/3) + \theta(1)$$

- The solution to this recurrence, by case 2 of the master theorem, is T(n) = O(logn).
- Alternatively, we can characterise the running time of MAX-HEAPIFY on a node of height "h" as O(h).

Example:

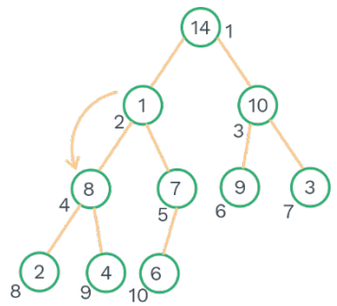
Let us assume that node 1 is at the root as shown below:

Step: 1



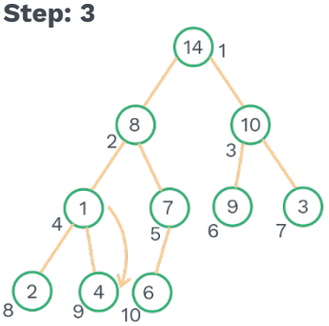
⇒ Swap (14, 1). Since, 14 is the largest value.

Step: 2



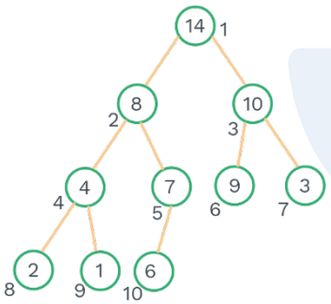
⇒ Swap (8, 1). Since, 8 is the largest element.

Step: 3



⇒ Swap (4, 1). Since, 4 is the largest number.

Step: 4



Previous Years' Question

What is the maximum height of any AVL tree with 7 nodes? Assume that the height of a tree with a single node is 0.

a) 2 **b)** 3
c) 4 **d)** 5
Sol: b) **(GATE: 2017)**

Space complexity:

- Since, it is a recursive function and for worst case (log n), a recursive call can be made.
- So, the function is called the number of level times.
- Number of levels will be logn in worst case.
- So, space complexity = O(log n)

Note:

i) In worst case, the number of recursive call = number of levels.
ii) Average space complexity = θ(log n)
iii) Best case space complexity = Ω(1)

Build max-heap

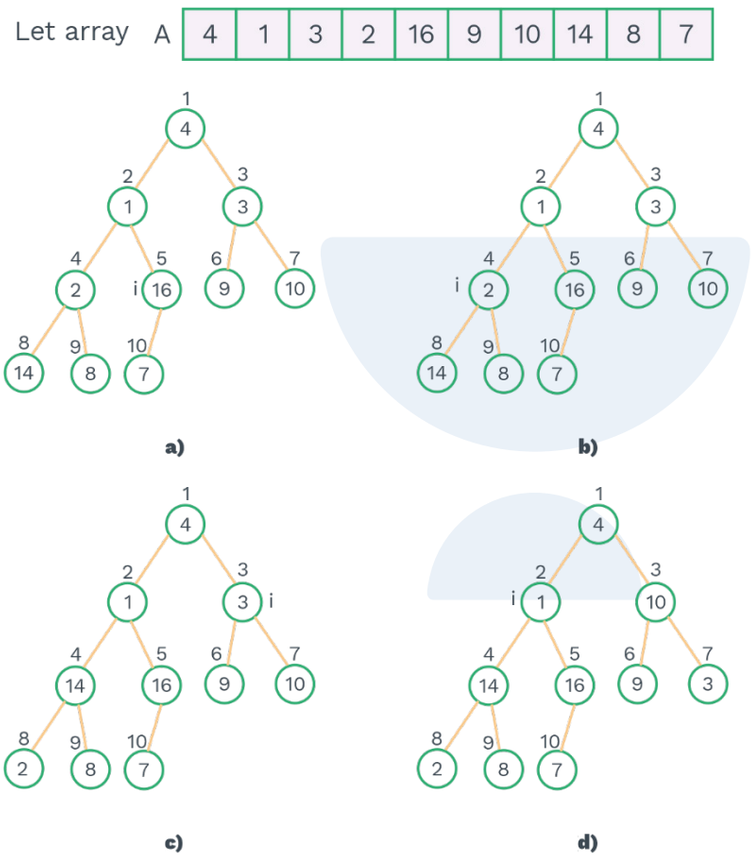
- To convert a given array A[1....n] into a max heap, Max-Heapify procedure is used in bottom to up fashion.

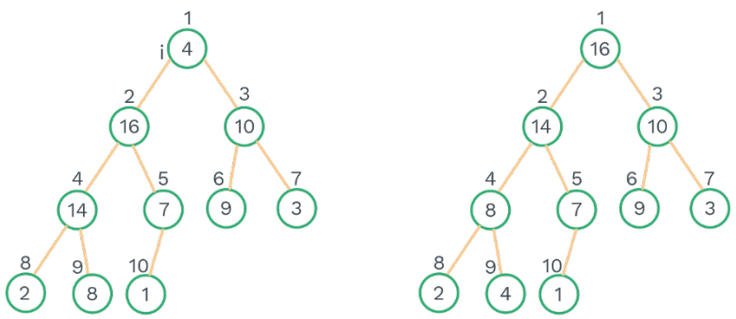
BUILD-MAX-HEAP(A)

1) A.heap–size = A.length
2) for i = ⌊A.length / 2⌋ downto 1
3) MAX-HEAPIFY (A, i)

Note:

As all the leaf nodes are by default heap.
So, we have to start the iteration from last non-leaf node to node index 1(initial).





e) f)

- As we can see in the above figures that index i points to node 5 first and then call Max-Heapify.
- In the next iteration, index i will point to 4 and so on.
- Final result we get is Figure (f) which is final max-heap.

Time complexity:

- Time taken by the Max-Heapify procedure when called on a node having height 'h' = O(h)
- Therefore, the total time taken by Build-Max-Heap is as:

$$\sum_{h=0}^{\log n}\left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) \quad \left(\begin{array}{l}\text{Here, 'n' is the total number of nodes in}\\ \text{a tree, where h is height of node.}\end{array}\right)$$

$$\sum_{h=0}^{\log n}\left\lceil \frac{n}{2^{h}.2} \right\rceil (ch)$$

$$= \frac{cn}{2}\sum_{h=0}^{\log n}\left(\frac{h}{2^{h}}\right) \ldots\ldots\ldots (1)$$

$$= O\left(\frac{cn}{2}\sum_{h=0}^{\infty}\frac{h}{2^{h}}\right) \ldots\ldots\ldots (2) \quad \left(\begin{array}{l}\because \text{eq.}(2) \text{ is greater than eq. } (1),\\ \text{that's why we are putting Big Oh}\end{array}\right)$$

$$= O(n) \quad \left[\because \sum_{h=0}^{\infty}\frac{h}{2^{h}} = 2\right] (\text{this is a harmonic propagation}).$$

So, time complexity is O(n).

Space complexity:

- Build max heap space complexity will be same as space complexity taken by max-heapify (A,i).
- It will take maximum, when it is called from root.
- That's why the space complexity is O(log n).

Deletion of Maximum element from max-heap

```
Heap_Delete_Max(A)
{
  if(A.heap–size < 1)
     printf ("Heap Underflow");
  max = A[1];
  A[1] = A[A.heap-size];
  A.heap–size = A.heap–size – 1;
  MAX–HEAPIFY (A, 1);
  return max;
}
```

- Here, space complexity and time complexity will be for MAX-HEAPIFY (A,1) only.
- So, time and space complexity are O(log n).

Increase Key in max-heap

Let 'A' be the array, and 'i' be the index, whose value we want to increase to key.

```
Heap_Increase_Key (A, i, key)
{
  if (key < A[i])
     printf ("error");
  A[i] = key;
  while (i > 1 && A[i/2] < A[i])
  {
      exchange A[i/2] and A[i]
      i=i/2;
   }
}
```

Time complexity:

- In worst case, the leaf node will get increased, and then Heap_Increase_Key (A, i, Key) will get called from leaf.
- That's why it has to go till the root node. So, it will take O(log n).

Space complexity:

- Heap_Increase_Key(A, i, key) does not require any extra space. Hence, its space complexity is O(1).

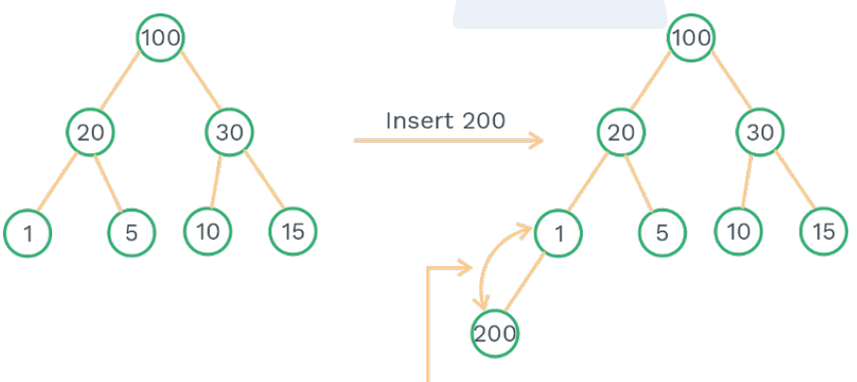
Note:

- For Heap_decrease_Key( ) on max–heap, we have to just call the max–heapify at that index, where decrease key happen, and after that we will get a heap.
- Space Complexity of Heap_decrease_Key( ) is O(1).
- Time Complexity of Heap_decrease_Key( ) is O(log n).

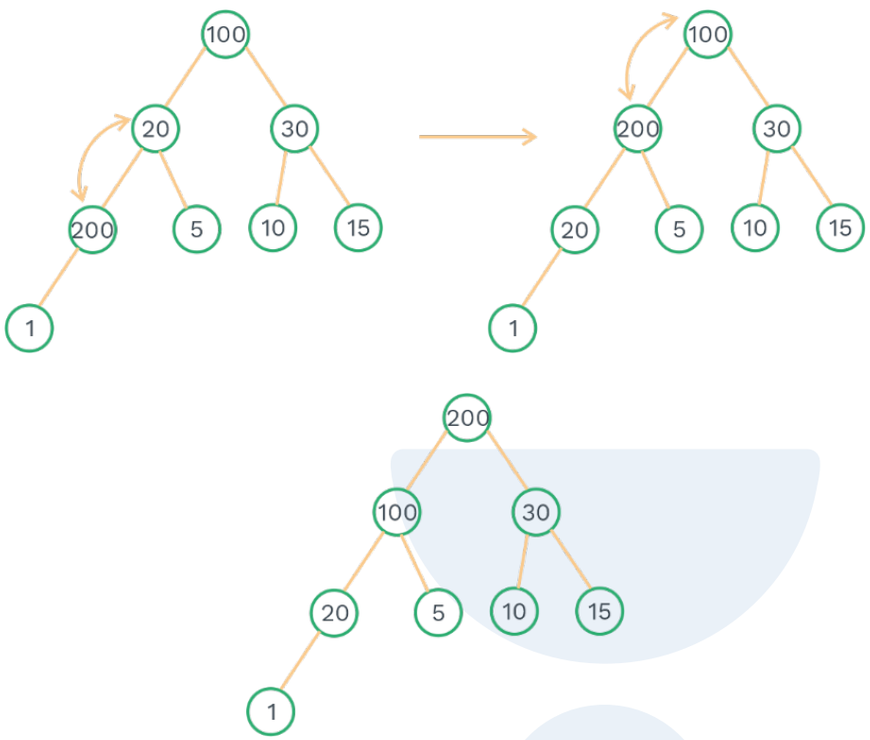
Insert an element in max-heap:

- Whenever we want to insert an element in max–heap, then insert the element at last index (position).
- After insertion, compare it with parent, and if parent is lesser than new element, then swap it.
- We will continue this procedure, until either we reach at root, or if parent is greater than the child.

Example:



Since, newly inserted element is greater. So, swap (200, 1).



- So, in worst case, the newly inserted element will have to traverse from the leaf node to the root.
- Hence, time complexity is O(log n).
- We can even use the max_heap_increase_key (A, i, Key) to implement the insertion into max-heap.

Procedure:

i) At new node we will insert an element $-\infty$. ($-\infty$ is some element, which could be very very smaller than all the elements).

ii) Since-infinity is there, then increasing that node from -infinity to newly inserted node and to do that we called increased key and we got the max-heap.

Finding minimum element in max-heap (having 'n' elements)

- Finding a minimum element will take O(n), because minimum will be present at leaf.
- The starting index of leaf node of heap having n nodes is

$$\left(\left(\left\lfloor \frac{n}{2} \right\rfloor + 1\right) \text{to } n\right).$$

- So, minimum element will be in from $\left(\left\lfloor \frac{n}{2} \right\rfloor + 1\right)$ to n.
- So, at most we need to search $\frac{n}{2}$ element to find minimum. Hence, the time complexity is $O\left(\frac{n}{2}\right) = O(n)$.

Search an element in Heap

- To search an element in Heap, we may need to visit every element.
- Let us assume number of elements in heap are n. Then, time complexity to search an element is O(n).

Delete an element in Heap

- To delete an element, first we need to search an element.
- To search an element, we require O(n) as time complexity.
- After deletion of an element, the tree may violate the Heap condition. So, again we need to call heapify function.
- Hence, its time complexity is O(n).

Time complexity of max-heap with different operations

Operations in max-heap	Time complexity
Find Maximum	O(1)
Delete Maximum	O(log n)
Insert an Element	O(log n)
Key Increase	O(log n)
Decrease Key	O(log n)
Search a random element	O(n)
Find minimum	O(n)
Delete an element	O(n)

- Similarly, we can have all these operations on min-heap.

Heapsort

- In this algorithm, we are swapping the root element with the last element in the heap, and then reducing the heapsize by 1. After that, we are applying the Max-heapify in the root.
- In this way, we can sort the Heap.

Pseudocode:

```
Heap_Sort (A) /* A is the array having 'n' elements.
{
  Build-Max-Heap(A)
  for (i = A.length down to 2)
  {
     exchange A[1] with A[i]
     A.heapsize = A.heapsize – 1
     Max-Heapify (A, 1)
   }
}
```

Time complexity:

In Heapsort,

Step 1: Building max heap takes O(n) time.

Step 2:

Swapping of A[1] with A[n]

A.heapsize = A.heapsize – 1

Max-Heapify(A,1)

All these lines will take time as O(log n) in one iteration due to Max-Heapify.

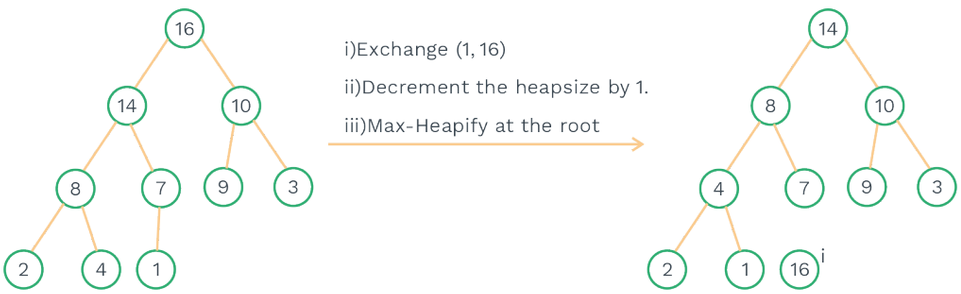
For n nodes(elements), it will take overall O(n log n) time.

Overall time complexity

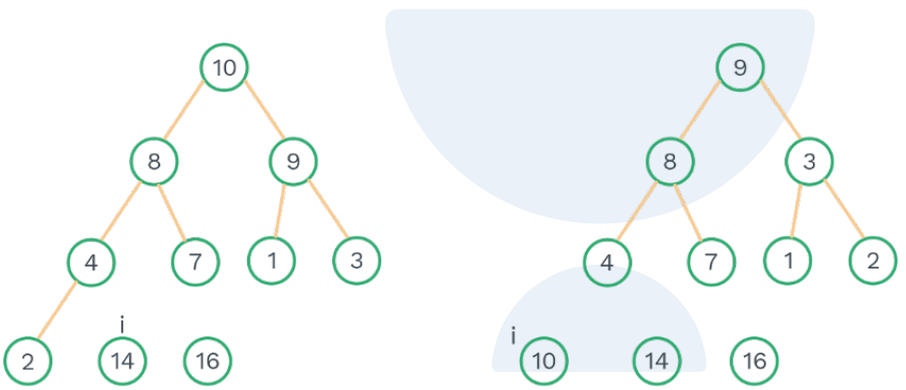
=O(n) + O(n log n)

=O(n log n)

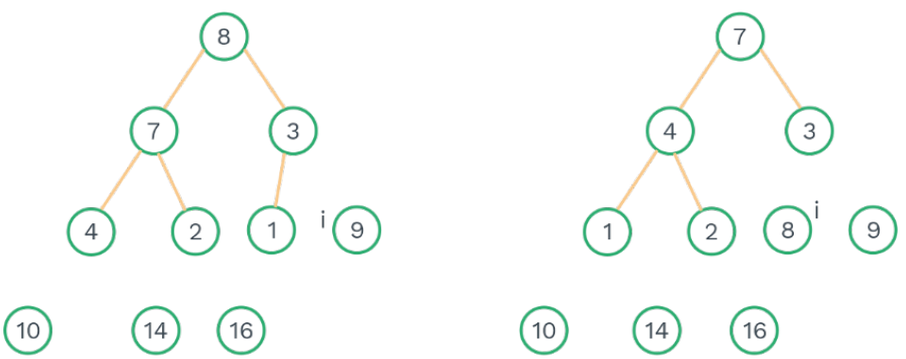
Note: In the below example of heapsort, we have already built the heap.


16
14
10
8
7
9
3
2
4
1
i)Exchange (1, 16)
ii)Decrement the heapsize by 1.
iii)Max-Heapify at the root
14
8
10
4
7
9
3
2
1
16
i

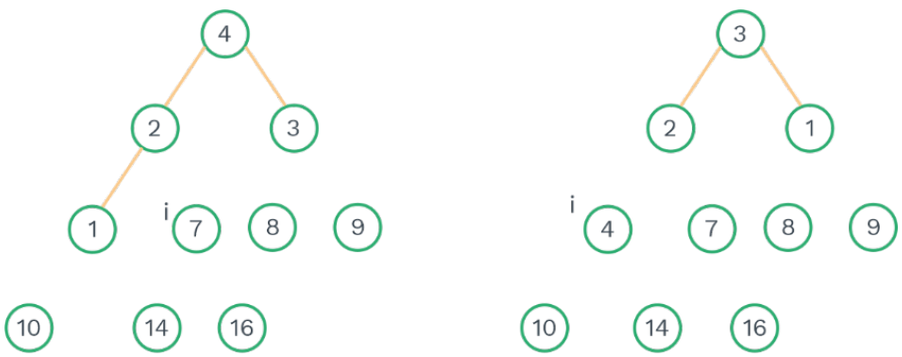
a) **b)**


10
8
9
4
7
1
3
i
2
14
16
9
8
3
4
7
1
2
i
10
14
16

c) **d)**

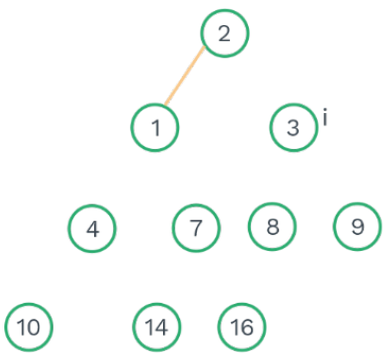

8
7
3
4
2
1
i
9
10
14
16
7
4
3
1
2
8
i
9
10
14
16

e) **f)**



g)

h)



i)

j)

(K)

- In the above figures, Figure(a) shows the max-heap after the call of Build-Max-Heap.
- Figures (b) to (j) show swapping of A[1] with A[i] and decrement of heapsize each time and the position of i on the max-heap after each call of Max-Heapify.

Previous Years' Question

Consider the process of inserting an element into a Max Heap, where the Max Heap is represented by an array. Suppose we perform a binary search on the path from the new leaf to the root to find the position for the newly inserted element, the number of comparisons performed is:

a) $\theta(\log_2 n)$ **b)** $\theta(\log_2 \log_2 n)$

c) $\theta(n)$ **d)** $\theta(n \log_2 n)$

Sol: b) **(GATE: 2007)**

Previous Years' Question

We have a binary heap on n elements and wish to insert n more elements (not necessarily one after another) into this heap. The total time required for this is

a) $\theta(\log n)$ **b)** $\theta(n)$

c) $\theta(n \log n)$ **d)** $\theta(n^2)$

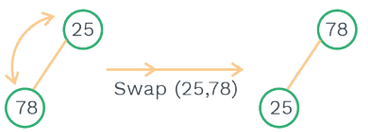
Sol: b)

SOLVED EXAMPLES

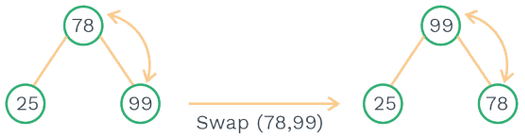
Q1 **Draw binary max-heap for the following given sequence:**
25, 78, 99, 98, 82, 100, 102

Sol: Step 1: Insert 25

Step 2: Insert 78

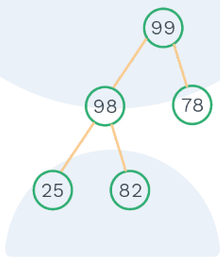


Step 3: Insert 99

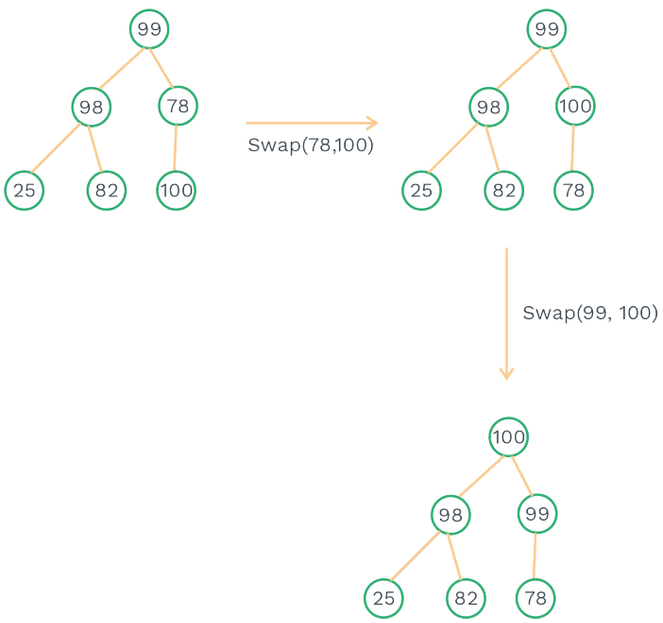


Step 4: Insert 98

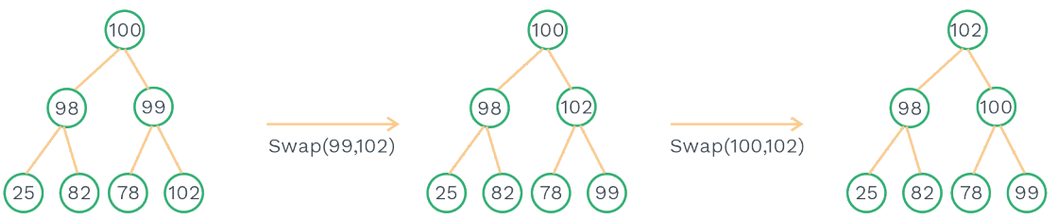
Step 5: Insert 82



Step 6: insert 100



Step 7: Insert 102

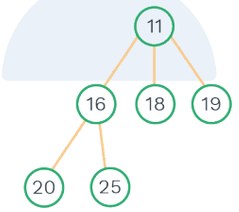


Q2 **Which of the following sequence represents ternary max-heap:**
a) 11,16,18,19,20,25
b) 15,17,25,30,35,5
c) 20,4,2,5,7,1
d) 40,10,25,30,9,2

Sol: **d)**

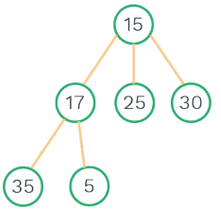
Explanation: Since, it is a ternary max-heap. So, it will have 3 children.

Option a):



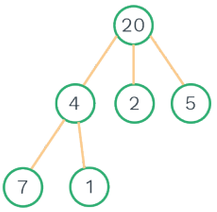
It is not a max-heap. It is a min-heap.

Option b):



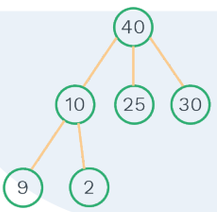
Here, node 25 is greater than node 15. Therefore, it is not a max-heap.

Option c):



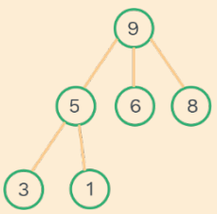
Here, node 7 is greater than node 4.
∴ It is not a max-heap.

Option d):

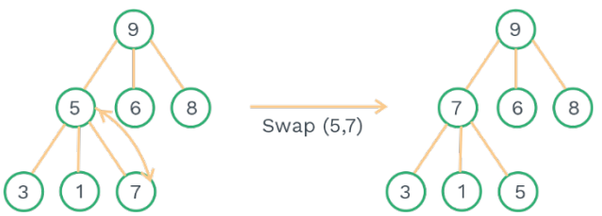


It is a max-heap.

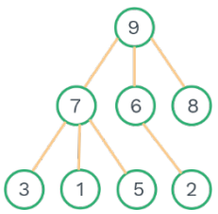
Q3 **Insert 7, 2, 10, 4 in that order in the ternary max heap as shown below:**



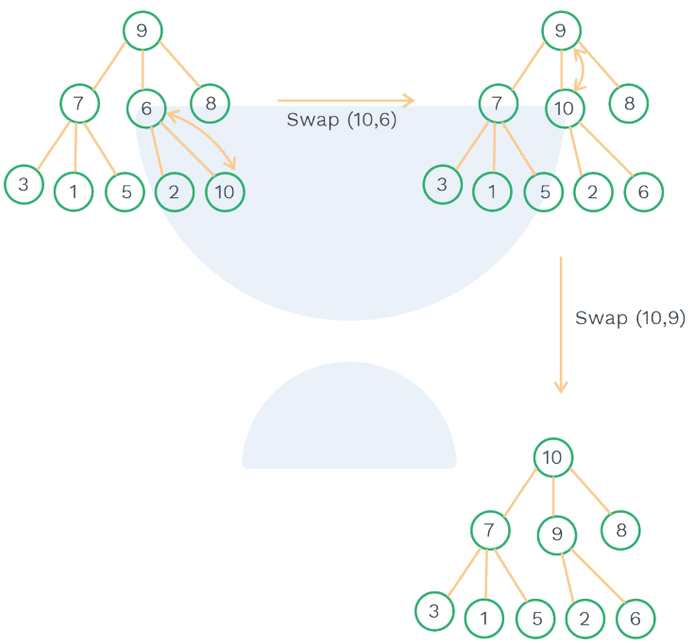
Sol. Step 1: Insert 7



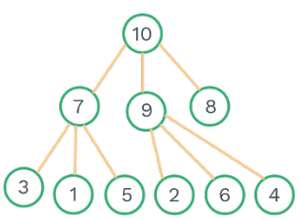
Step 2: Insert 2



Step 3: Insert 10



Step 4: Insert 4



Q4 **Number of binary tree with 4 nodes A, B, C and D which is having preorder as ABCD.**

Sol: Number of structure with 'n' nodes = $\frac{^{2n}C_n}{(n+1)}$

So, with 4 nodes, number of structure = $\frac{^{8}C_4}{5}$

$$= \frac{\frac{8!}{4!4!}}{5}$$

$$= \frac{8 \times 7 \times 6 \times 5 \times 4!}{4! \times 4! \times 5}$$

$$= \frac{8 \times 7 \times 6 \times 5}{4 \times 3 \times 2 \times 1 \times 5} = 14$$

Since, each structure will represent one particular preorder, i.e, ABCD.
So, 14 is the number of binary tree with 4 nodes A, B, C and D which is having preorder as ABCD.

Q5 **Given an inorder as 4, 12, 16, 27, 29, 34, 44, 50, 52, 65, 77, 88, 92, 93 and pre-order as 50, 27, 16, 4, 12, 34, 29, 44, 88, 65, 52, 77, 93, 92 of binary tree. What is postorder?**
a) 12, 4, 16, 29, 44, 34, 27, 52, 77, 65, 92, 93, 88, 50
b) 29, 44, 16, 4, 12, 34, 27, 52, 77, 65, 92, 93, 50, 88
c) 12, 4, 16, 29, 34, 44, 27, 52, 77, 65, 92, 93, 88, 50
d) 12, 4, 16, 29, 44, 34, 27, 52, 77, 65, 92, 93, 50, 88

Sol: **a)**
Explanation:
Preorder: (Root,Left,Right)
Visit Root, Visit Left subtree, Visit Right subtree

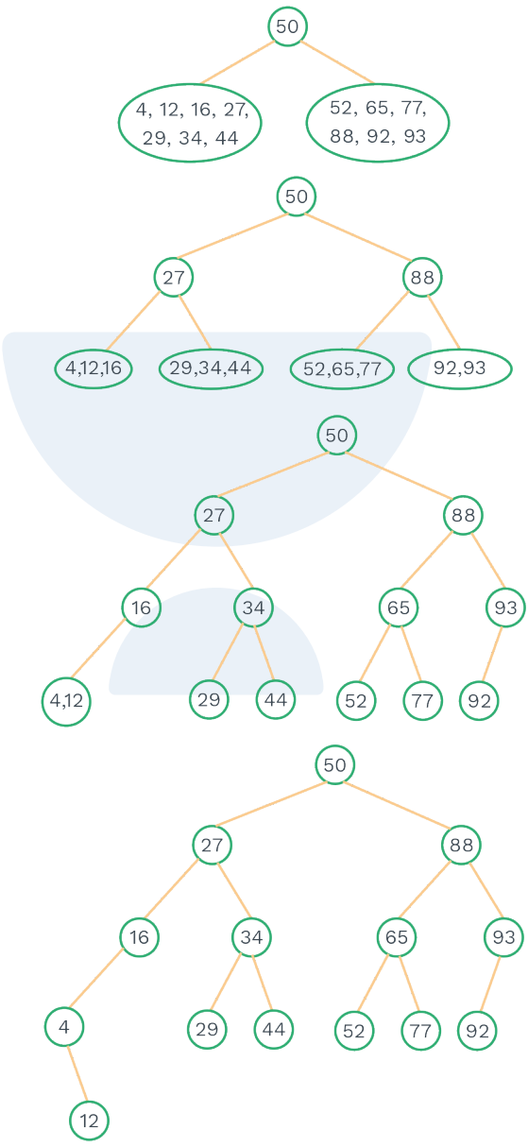
Inorder: (Left,Root,Right)
Visit Left subtree, Visit Root, Visit Right subtree
So, according to preorder, 50 is the root node.

Inorder: (Left, Root, Right)

4, 12, 16, 27, 29, 34, 44, 50, 52, 65, 77, 88, 92, 93

Preorder: (Root, Left, Right)

50, 27, 16, 4, 12, 34, 29, 44, 88, 65, 52, 77, 93, 92



So, the postorder is 12, 4, 16, 29, 44, 34, 27, 52, 77, 65, 92, 93, 88, 50.
We just traversed the tree from top to down, left to right.
Whenever we visit a node for the last time, we print it.

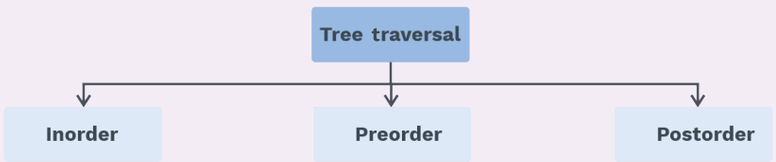
Q6 **Suppose the nodes in BST are given as:**
50,30,80,20,48,60,90,10,15
What will be the Inorder traversal of BST:
a) 10,15,20,30,48,50,60,80,90
b) 15,20,30,10,50,48,80,90,60
c) 10,15,20,30,50,48,60,80,90
d) None of above

Sol: **a)**

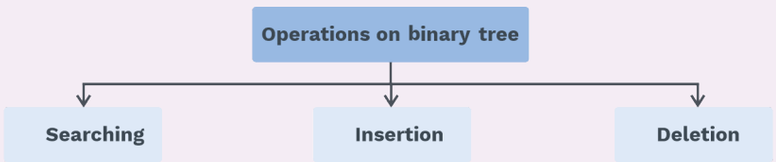
Explanation:
Inorder traversal of a binary search tree is in ascending order (Sorted order) of key value of nodes.
So, a) is the correct option.

Chapter Summary

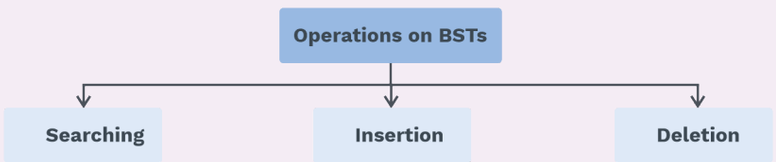
- **Tree:** It is an example of non-linear data structures.
- **Binary trees:** A tree in every node is having atmost 2 children (i.e. either 0 child or 1 child or 2 children).
- **Tree traversal:**



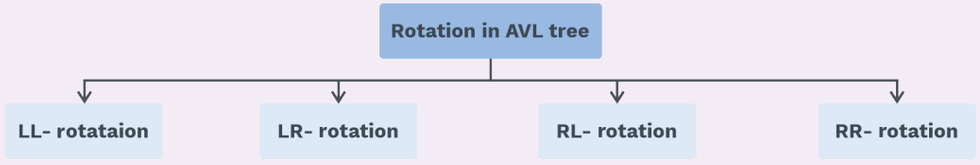
- **Operations on binary tree:**



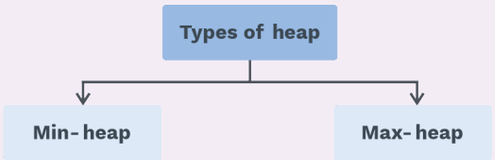
- **Binary seach tree:** It is used for searching.
- In BSTs, the element present in the left subtree should be less than the root element, and the element present in the right subtree should be greater than the root element.
- Operations on BSTs:



- **AVL-tree:** It is a height balanced-tree.
- Balance factor of each node in AVL tree must be either -1 or 0 or 1.
- Rotation in AVL tree:



- **Heap:** It is a complete binary-tree, which takes O(log n) time to insert an element.



- **HeapSort:**
- First swap the root element with the last element in the heap.
- After that decrease the heapsize by 1
- Then perform max-heapify on the root.

6 Hashing

HASHING TECHNIQUES

Introduction:

- Hashing is a searching technique.
- Hashing is an efficient searching technique in which the number of keys that must be inspected before obtaining the desired one is minimised.
- Worst case, time complexity in case of hashing is expected as O(1).
- Basically, the motive behind hashing is to retrieve each key stored in the table in a single access.
- In hashing, the location of the record within the table depends only on the key resulting in retrieving each record in a single access. It does not depend on the locations of other keys, as in a tree.

Hash table and hash function:

1) Hash table:

- Hash table is a table used to maintain keys in it.
- The most efficient way to organize such a table is as array.
- Each key is stored in the hash table at a specific offset from the base address of the table.
- If the record keys are integers, the keys themselves can serve as indices to the array, which is used as a table.
- An example of a hash table using an array:

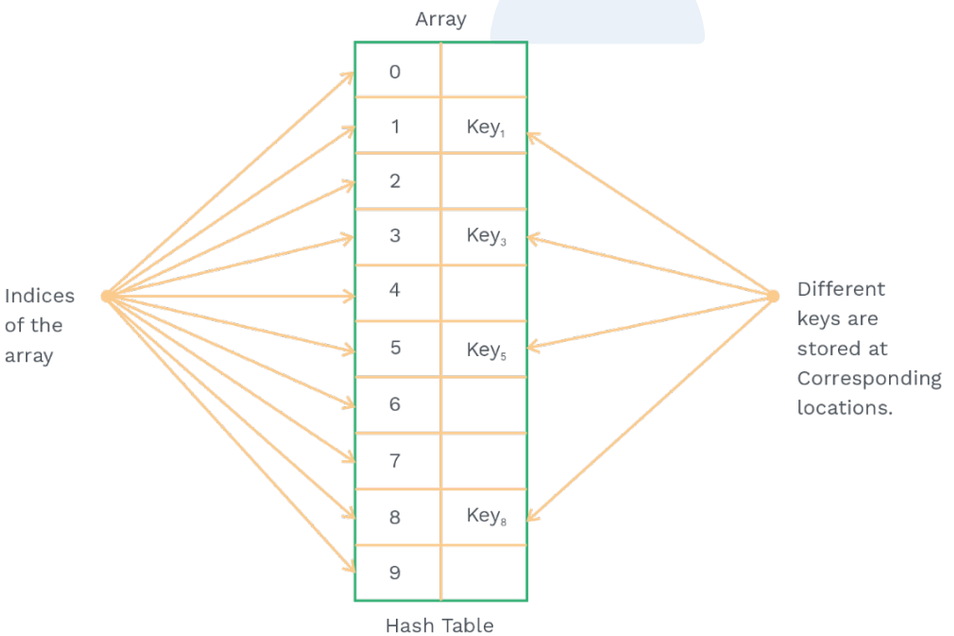


Table 6.1

2) Hash function:

- A function which transforms a key into a table index is called as a hash function.
- Consider the situation where keys are supposed to be stored in a hash table directly by their values, i.e., key 'k' will be stored in the k^{th} position only.

For example:

⇒ keys are 1, 3, 5, and 8, so we need a table of size at least 8. let's take it of size 10.

⇒ Each key will go to its corresponding location as shown below:

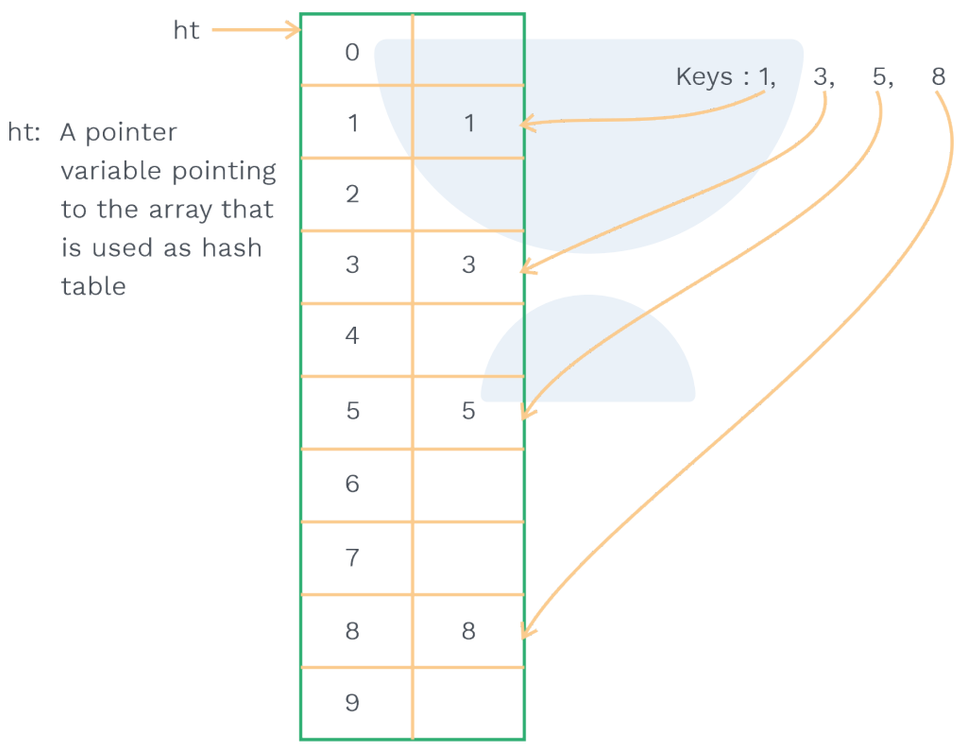


Table 6.2

The above hash table is called the direct address table.

- But suppose, if one key value is large enough,

For example:

keys are 1, 3, 5, and 100058.

So, the minimum table size required is 1,00,058. Thus, for storing only 4 keys

(1, 3, 5, and 100058), a table of size 1,00,058 is needed, by doing so, a lot of space is wasted.

- To overcome the above situation, the hash function is used.
- Basically, the hash function is used to compress the keys which are to be stored in the hash table.
- If 'h' is a hash function and the key is 'k', then h(k) is called the 'hash value of key'.
- Let's understand the hash function by taking h(k) = k%10, where '%' is the modulo operator.

Example:

Keys are 105, 110, 202, 403, 509

Since the hash function used is "key modulo 10", so table size should be 10.

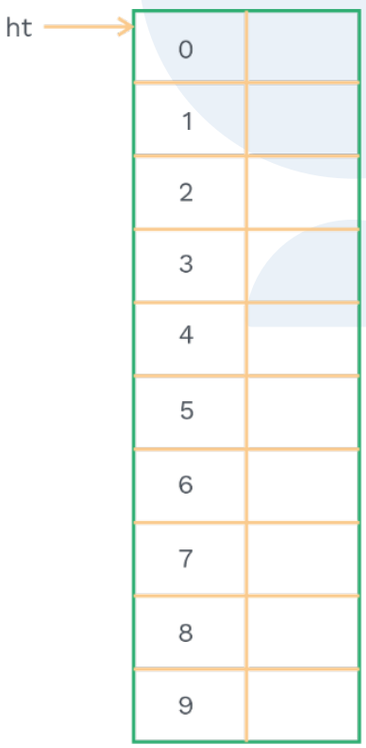


Table 6.3

Table size is 10 because after applying hash function h(k) = k%10, the last digit of the key will decide the location/position where the key is to be stored. Thus, 0 to 9 digits are sufficient to store all the keys.

So, h(105) = 105 % 10 = 5

h(110) = 110 % 10 = 0

h(202) = 202 % 10 = 2

h(403) = 403 % 10 = 3

h(509) = 509 % 10 = 9

Now, keys can be stored as :

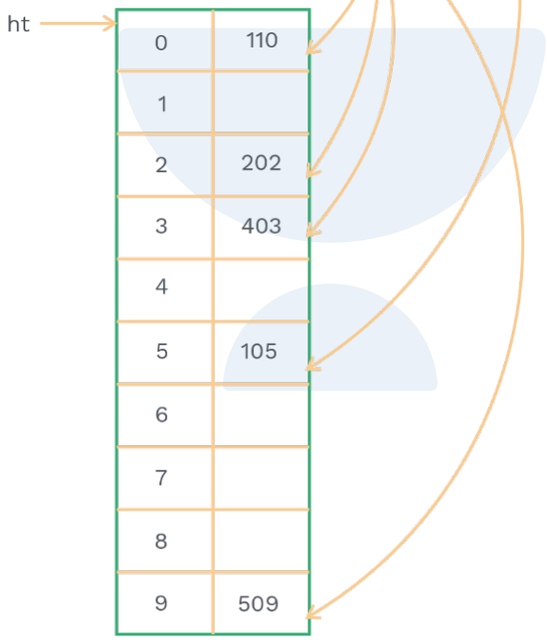


Table 6.4

So, this table size is minimized by compressing the given keys.

Types of hash functions:

Different types of hash functions are:

1) Division Modulo Method
2) Digit Extraction Method
3) Mid Square Method
4) Folding Method

Rack Your Brain

A hash function is given as h(k) = K%69, where K indicates a key and h(k) indicates hash function of the key. What should be the minimum size of hash table and indices of a hash table?

Detailed discussion on each of the hash function types:

1) Division modulo method:

- Division Modulo Method is used to compress the key.
- It uses function as, h(k) = k mod (hash_table_size).
- Obtained mod value of key with hash_table_size gives the position for the particular key to be stored at.

- For example:

Hash table size is S = 1000 and hash index varies from 0 to 999.
Key = 5231119265
Now, the key is to be stored at?
Mod value for the key:
h(k) = k mod (hash_table_size)
= (5231119265) mod 1000
= 265; It is the position where the key will be stored.

Previous Years' Question

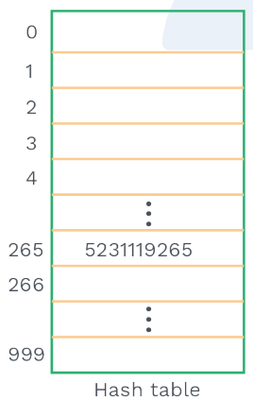
Given the following input (4322, 1334, 1471, 9679, 1989, 6171, 6173, 4199) and the hash function x mod 10, which of the following statements are true?

i) 9679, 1989, 4199 hash to the same value
ii) 1471, 6171 hash to the same value
iii) All elements hash to the same value
iv) Each element hashes to a different value

a) i only **b)** ii only
c) i and ii only **d)** iii or iv

Sol: c) **(GATE CSE: 2004)**

like,



Hash table

Table 6.5

- It is the easiest method to create a hash function.

2) Digit extraction method:

- Digit Extraction Method is also used to compress the key.
- In this method, selected digits are extracted from the key to be stored and used as the position where key has to be stored.
- For example:

Suppose hash table addresses are from 0 to 999, means of size 1000

if keys are of 7 digits and are as follows: 5231119,

4290386,

1296043,

5930532,

6080239,

According to the digit extraction method, we could extract digits from positions 2, 3 and 7 from the keys.

like,

5 ②③ 1 1 1 ⑨ ⟶ 239; 5231119 will be stored at 239

4 ②⑨ 0 3 8 ⑥ ⟶ 296; 4290386 will be stored at 296 and so on.

1 ②⑨ 6 0 4 ③ ⟶ 293

5 ⑨③ 0 5 3 ② ⟶ 932

6 ⓪⑧ 0 2 3 ⑨ ⟶ 089

The hash table looks like this,

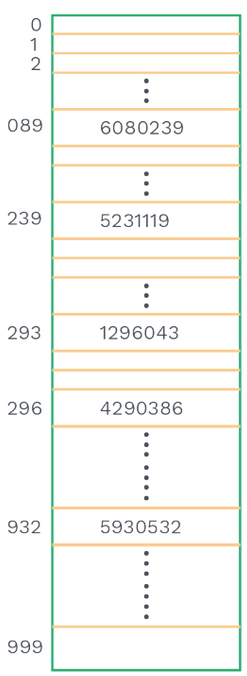


Table 6.6

Note:

Digits can be extracted from any desired position in the keys.

3) Mid square method:

- It is used to compress the key, which is supposed to be stored in the hash table.
- In this method, we perform a square operation on the key and take the middle digits as addresses for keys.

Grey Matter Alert!

The Digit extraction method is also called Truncation Method.

- For example:

Suppose the key is 32165 and the size of the hash table is 1000.
So, h(k) = $(32165)^2$ plus take middle digits.

= 1 0 3 4 5 8 7 2 2 5

position for the key (we consider 3 digits by referring the size of the hash table).

Hash table looks like,

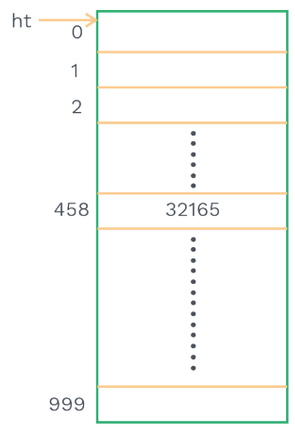


Table 6.7

4) Folding method:

- In this method, we select a number of digits from the key and perform addition on digits and make it concise.
- The value resulting from doing the above changes, acts as the address/position for the key to be stored in the hash table.

- For example:

Suppose the key to be stored is 321653533 and the size of the hash table is 1000.

Since table indices vary from 0 to 999, we will take the number of digits equal to 3 to fold from the key.

So, h(k) = 321 653 533

⇒ 321 + 653 + 533 = 1507

⇒ 150 + 007 = 157 (This location will act as the address to store the key.)

Hash table looks like,

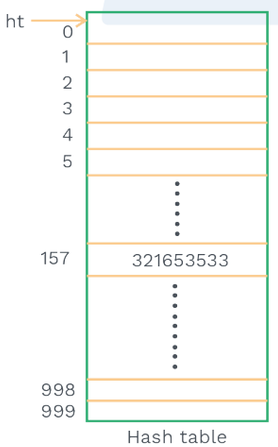


Table 6.8

Note:

The folding method and mid square method are better among all the methods. Since every digit is participating, it is difficult to find other counter keys which generate the same address.

Retrieval of keys from hash table:
Retrieval of keys means we need to find the location of the key exactly where it is stored in the hash table.
So, depending on the type of hash functions, we search the keys. Let's check one by one:

Previous Years' Question

Which one of the following hash functions on integers will distribute keys most uniformly over 10 buckets numbered 0 to 9 for i ranging from 0 to 2020?

a) $h(i) = i^2 \bmod 10$
b) $h(i) = i^3 \bmod 10$
c) $h(i) = (11*i^2) \bmod 10$
d) $h(i) = (12*i) \bmod 10$

Sol: b)

(GATE CSE: 2015 (Set–2))

1) Division modulo method:
The hash table size is 1000 (0 to 999).
The key is 5231119265 and we need to find its location. So, applying the division modulo method,
h (5231119265) = (5231119265) % (1000)
= 265; 265 is the address, for the key.
Thus, it just takes constant time to search a key.
means O(1) in Every Case.

2) Digit extraction method:
Hash table size is 1000 (0 to 999). Key is 5231119, and we need to find its location.
So, applying the digit extraction method,
Extract the digits from positions exactly which are used to store.

h(523119) = 5 ② ③ 1 1 1 ⑨ ⟶ 239; 239 is the address for the key.

3) Mid square method:
Hash table size is 1000 (0 to 999)
The key is 32165, and we need to find its location.
So, applying mid square method,
$h(32165) = (32165)^2$ plus taking middle digits

= 1 0 3 4 5 8 7 2 2 5

458 is the address for the key.

4) Folding method:
Hash table size is 1000 (0 to 999)
The key is 321653533, and we need to find its location.
So, applying the folding method in the same way,

h(321653533) = 321 653 533

$\Rightarrow$ 321 + 653 + 533 = 1507

$\Rightarrow$ 150 + 007 = (157) ; is the address for the key.

Note:

For retrieval of keys by using any of the above methods, constant time is needed i.e. O(1) Every Case.

Load factor and collisions:

Load factor:

- The number of keys per slot of the hash table is called the load factor.
- Load factor is denoted by 'μ'.
- Suppose the number of slots in the hash table equals to N and the number of keys to be stored in the hash table is x.
- $\therefore$ Number of keys per slot = $\frac{x}{N}$

This means the average number of keys per slot = $\frac{x}{N}$

$\therefore \quad \boxed{\mu = \frac{x}{N}}$

Previous Years' Question

Consider the following two statements:

i) A hash function (these are often used for computing digital signatures) is an injective function.

ii) encryption technique such as DES performs a permutation on the elements of its input alphabet.

Which one of the following options is valid for the above two statements?

a) Both are false

b) Statement (i) is true, and the other is false

c) Statement (ii) is true, and the other is false

d) Both are true

Sol: c) **(GATE CSE: 2007)**

SOLVED EXAMPLES

Q1 **Given a hash table with 39 slots that store 250 keys. Calculate the load factor 'μ'. (up to two decimal places)**

Sol: **6.41**

Given,
Number of slots = 39
Number of keys = 250

$$\therefore \mu = \frac{\text{Number of keys}}{\text{Number of slots}} = \frac{250}{39} = 6.41 \text{ Ans.}$$

Collision:

- After applying the hash function on keys to map them to some address of the hash table, if more than one key are mapped to the same address of the hash table, then this phenomenon results in a collision.
- In simple words, if one slot of hash table is getting more than one key to be stored at it, it is called a collision.
- Let's understand it with an example:

Keys are given as: 49, 26, 32, 59, 60, 29, 41
Hash table size is given as: 10 (0 to 9)
Hash function is given as: h(K) = K mod 10

Previous Years' Question

Consider a hash function that distributes keys uniformly. The hash table size is 20. After hashing of how many keys will the probability that any new key hashed collides with an existing one exceed 0.5.

a) 5 **b)** 6
c) 7 **d)** 10
Sol: d) **(GATE CSE: 2007)**

Calculating hash indices for each key,

1) h(49) = 49 mod 10 = 9; this key will go to hash index 9 and so on.
2) h(26) = 26 mod 10 = 6
3) h(32) = 32 mod 10 = 2
4) h(59) = 59 mod 10 = 9
5) h(60) = 60 mod 10 = 0
6) h(29) = 29 mod 10 = 9
7) h(41) = 41 mod 10 = 1

Mapping each key to its location in the hash table,

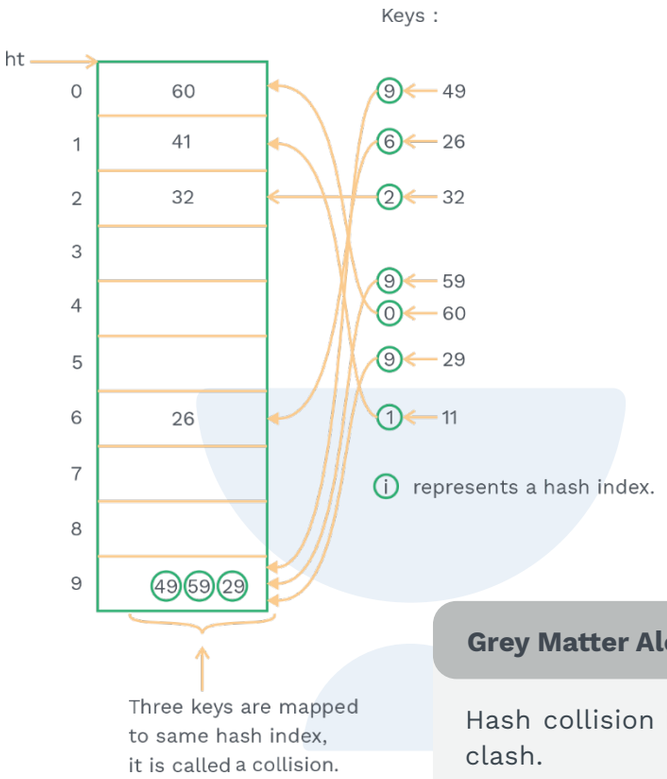


Table 6.9

Grey Matter Alert!

Hash collision is also termed a hash clash.

Q2 **There is a hash table of size 10. Keys are given as 539, 510, 201, 202, 503, 405, 406, 911, 922, 906. Find if there is any collision present in the system and if yes, find the number of collisions encountered. The division modulo method is used as the hash function.**

Sol: **Yes, 3 collisions.**

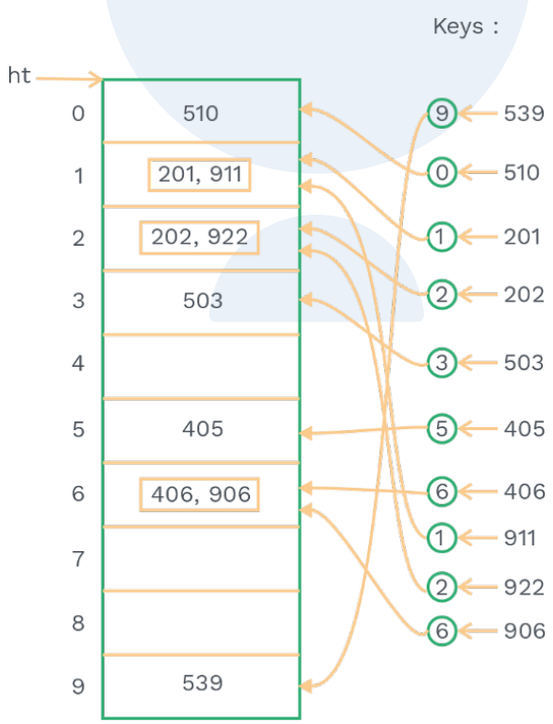
Calculating hash indices for each key:
Given the hash function used is the division modulo method.

So, $h(k) = k \bmod 10$

($h(k)$: hash of key; k: key; 10: table size)

h(539) = 539 mod 10 = 9
h(510) = 510 mod 10 = 0
h(201) = 201 mod 10 = 1
h(202) = 202 mod 10 = 2
h(503) = 503 mod 10 = 3
h(405) = 405 mod 10 = 5
h(406) = 406 mod 10 = 6
h(911) = 911 mod 10 = 1
h(922) = 922 mod 10 = 2
h(906) = 906 mod 10 = 6

Mapping each key to its location in hash table,



i) Yes, At index no. 1, 2, 6, collisions are present.
ii) Calculating no. of collisions:
Index 1 → One collision
Index 2 → One collision
Index 6 → One collision
∴ Total no. of collisions = 3 Ans.

COLLISION RESOLUTION TECHNIQUES

- Ideally, no two keys should be compressed into the same integer. Unfortunately, such an ideal technique doesn't exist.
  So, here we will discuss some techniques which will resolve when collisions are resulted, and leads us close to ideal.

- Basically, there are two techniques of dealing with hash collisions.
  1) Chaining
  2) Open addressing

1) Chaining:

- Chaining is a collision resolution technique.
- This technique resolves collision by building a linked list of all keys hashed to the same hash index.
- Hash table is in the form of the array only, but it is the array of pointers in this case.
- Keys are stored in nodes of a linked list.
- At a specific hash index, one address is stored that is of the first node of the chain in the form of linked list.
- Applying chaining for some keys like:

Example: Keys are given as 49, 26, 32, 59, 69, 79, 60, 50, 30, 41, 61, 29

Table size is given as 10 (0 to 9).

Hash function, h(k) = k mod 10.

The collision resolution technique is chaining.

Calculating hash indices for each key,
h(49) = 49 mod 10 = 9
h(26) = 26 mod 10 = 6
h(32) = 32 mod 10 = 2
h(59) = 59 mod 10 = 9
h(69) = 69 mod 10 = 9
h(79) = 79 mod 10 = 9
h(60) = 60 mod 10 = 0
h(50) = 50 mod 10 = 0
h(30) = 30 mod 10 = 0
h(41) = 41 mod 10 = 1
h(61) = 61 mod 10 = 1
h(29) = 29 mod 10 = 9

Mapping each key to its location in the hash table,

49 → ⑨, 26 → ⑥, 32 → ②, 59 → ⑨, 69 → ⑨, 79 → ⑨, 60 → ⓪, 50 → ⓪, 30 → ⓪, 41 → ①, 61 → ①, 29 → ⑨

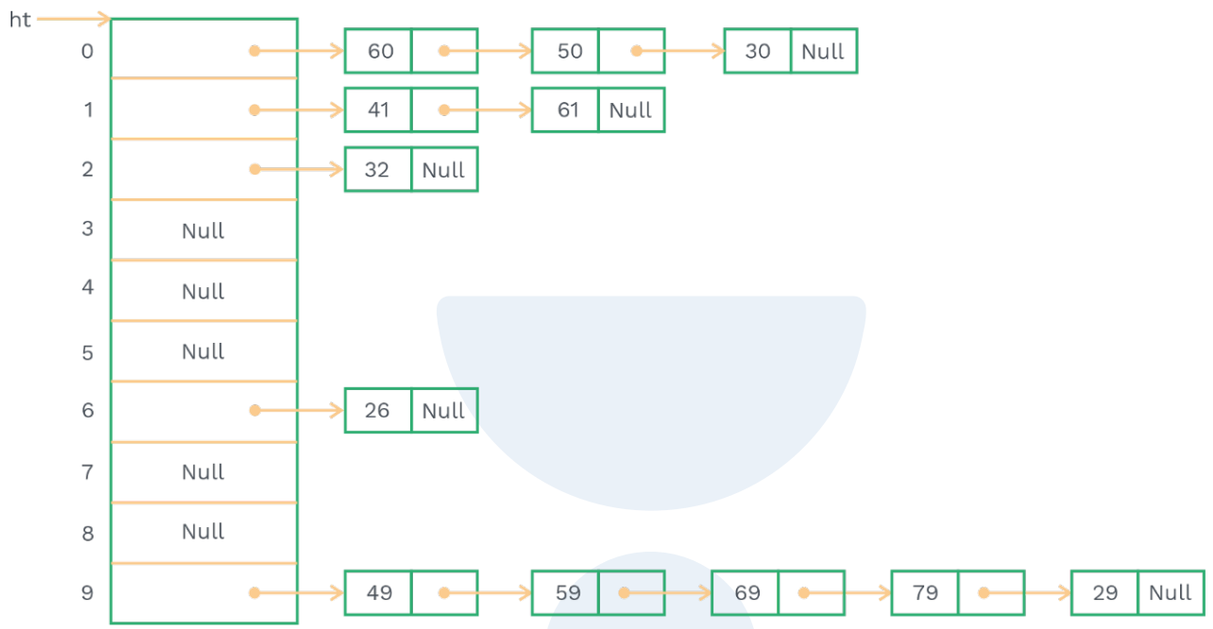


Table 6.10

- Array positions which don't have any node to point to are set to Null pointer.

Advantages of chaining:

- By applying chaining, a hash index can handle many keys in it. We can deal with the infinite number of collisions.
- Any insertion of a key in hash table will take constant time, i.e. O(1) every case.
- Deletion in chaining is easy because we store keys in an unsorted manner, so deletion of one key doesn't create trouble for other keys.

Disadvantages of chaining:

- Deletion and searching in chaining takes O(n) time complexity.
- Consider the previous instance,

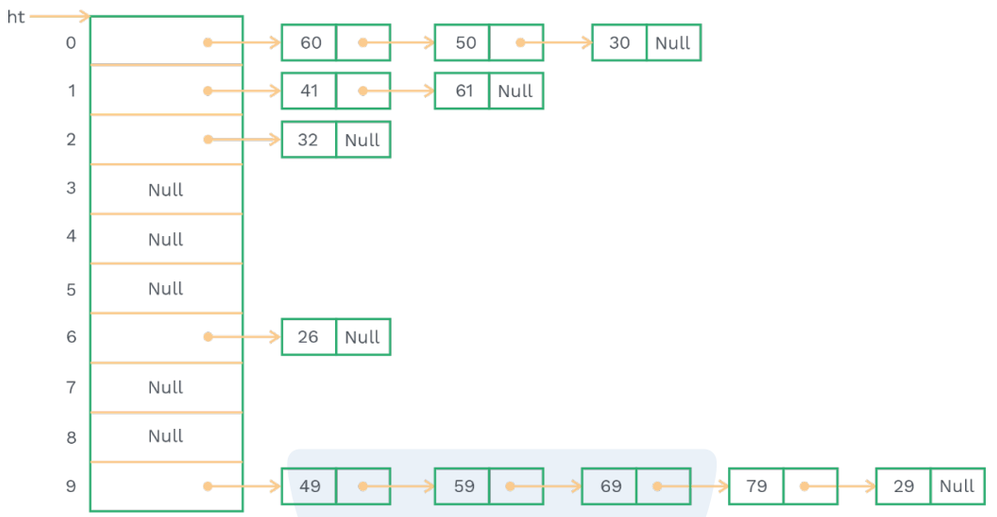


Table 6.11

we can analyse clearly that even though many slots are empty but we can't store keys in them, and forcefully, we need to take space outside the array. Chaining has a drawback in terms of space consumption.

Load factor:
Load factor for chaining; $\mu \geq 0$

It means keys per slot is 0 or any positive integer.
One slot can point to 0–key, 1–key or more than one key.

SOLVED EXAMPLES

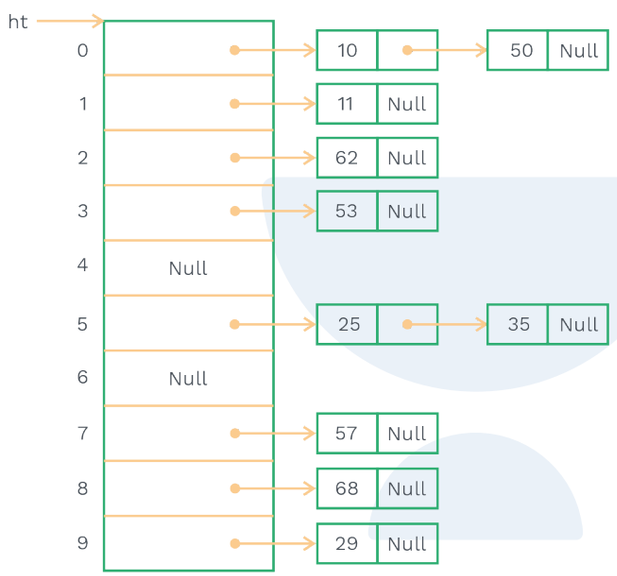
Q3 **Consider a hash table with 10 slots. The hash function used is h(k) = k mod 10. The collisions are resolved by using chaining. Keys 25, 68, 29, 35, 10, 53, 62, 57, 50, 11 are inserted in the given order. The maximum, minimum and average chain lengths for the hash table, respectively are: _____**

Sol: **2, 0, 1.**

Calculating hash indices for each key.
h(25) = 25 mod 10 = 5
h(68) = 68 mod 10 = 8
h(29) = 29 mod 10 = 9
h(35) = 35 mod 10 = 5

h(10) = 10 mod 10 = 0
h(53) = 53 mod 10 = 3
h(62) = 62 mod 10 = 2
h(57) = 57 mod 10 = 7
h(50) = 50 mod 10 = 0
h(11) = 11 mod 10 = 1

Applying Chaining,



Index no.	Chain length
0	2
1	1
2	1
3	1
4	0
5	2
6	0
7	1
8	1
9	1

∴ Maximum chain length = 2
Minimum chain length = 0
Average chain length = $\frac{2+1+1+1+0+2+0+1+1+1}{10} = \frac{10}{10} = 1$
So, 2, 0, 1 Ans.

Q4 **Consider a hash table with 10 slots which uses chaining for collision resolution. Assuming the table is initially empty and considering simple uniform hashing. What would be the probability that after 3 keys are inserted, a chain of size at least 2 results?**

Sol: **0.28**

- We have given Table size = 10
- Collision resolution technique – chaining
- Table is empty initially
- Simple uniform hashing
- 3 keys are inserted
- Simple uniform hashing – It evenly distributes elements into the slots of a hash table.

Probability of a chain of size atleast 2
= [Probability of a chain of size exactly 2]
+ [Probability of a chain of size exactly 3]

$$= 10 \times \left[{}^{3}C_{2} \times \frac{1}{10} \times \frac{1}{10} \times \frac{9}{10} \right] + 10 \times \left[{}^{3}C_{3} \times \frac{1}{10} \times \frac{1}{10} \times \frac{1}{10} \right]$$

$$= \left[{}^{3}C_{2} \times \frac{1}{10} \times \frac{1}{10} \times \frac{9}{10} \right] \times 10 + \left[{}^{3}C_{3} \times \frac{1}{10} \times \frac{1}{10} \times \frac{1}{10} \right] \times 10$$

Selecting 2 elements from 3 inserted elements

Probability of going an element to a slot

Probability of going 3rd element to any of other 9 slots.

for every 10 slots

Selecting all 3 inserted elements

$$= 3 \times \frac{1}{10} \times \frac{1}{10} \times \frac{9}{10} \times 10 + 1 \times \frac{1}{10} \times \frac{1}{10} \times \frac{1}{10} \times 10$$

$$= \frac{3 \times 9}{10 \times 10} + \frac{1}{10 \times 10}$$

$$= \frac{27}{100} + \frac{1}{100}$$

$$= \frac{28}{100}$$

= 0.28 Ans.

Previous Years' Question

An advantage of a chained hash table (external hashing) over the open addressing scheme is

a) Worst-case complexity of search operations are less
b) Space used is less
c) Deletion is easier
d) None of the above

Sol: c) **(GATE CSE: 1996)**

2) Open addressing:

- Open addressing is also a collision resolution technique like chaining.
- Only a hash table is used to store keys, unlike chaining.
- In open addressing, at any point, the size of the table must be greater than or equal to the total number of keys.

- There are three types of open addressing, considering the way the collisions are resolved:
  i) Linear Probing
  ii) Quadratic Probing
  iii) Double Hashing

Let's understand each of the above three types of open addressing one by one:

i) Linear probing:

- In this method, if some collision arises, then to resolve it, we probe next slot to store the key linear.
- Linear_Probing(k, i) = (h(k) + i) mod S

$$\forall\ i \in \{0, 1, 2, 3, \ldots\ldots, S-1\}$$

Where k is a key
h(k) is the hash of key 'k'.
i is iteration number or collision number.
S is the size of the hash table.

- When any collision is encountered for some key, the above definition is applied to resolve the collision according to Linear Probing.
- Let's understand linear probing with an example:

Keys are given as: 49, 26, 36, 32, 59, 60, 70, 80.
Hash table size given as: 10 (0 to 9)
Hash function used is: h(k) = k mod 10
Linear probing (k, i): (h(k) + i) mod S

Insert keys one by one into the hash table.
Initially, table is empty:

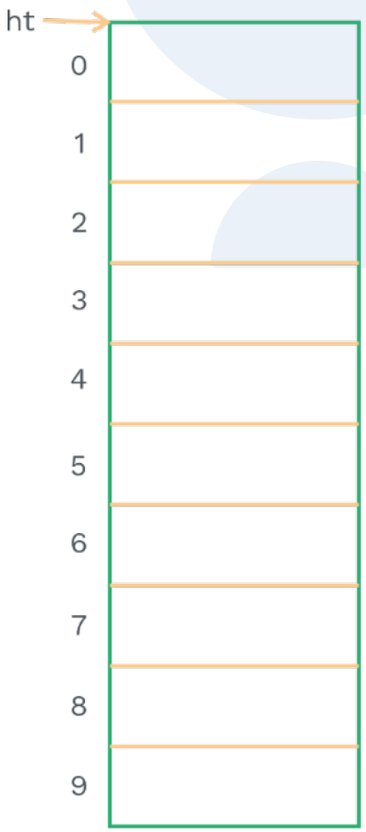


Table 6.12

1) h(49) = 49 mod 10 = 9 ✓

2) h(26) = 26 mod 10 = 6 ✓

3) h(36) = 36 mod 10 = 6 ⟶ 1–Collision

As we can see, key–36 is hashed to position '6', but we can't store key–36 to position '6' because one key is already present to position '6'.

ht	
0	59
1	60
2	32
3	70
4	80
5	
6	26
7	36
8	
9	49

Table 6.13

So, applying linear probing,
h(k) = 36 mod 10 = 6
Linear probing (k, i) = (h(k) + i) mod S

= (6 + 1) mod S
↑
i = 1 in first iteration and so on.
= 7 mod 10 = 7 ✓

Now, key–36 will go to position '7'.

4) h(32) = 32 mod 10 = 2 ✓

5) h(59) = 59 mod 10 = 9 ⟶ 1–Collision

Linear probing (k, 1) = (9 + 1) mod 10
= 10 mod 10 = 0 ✓

6) h(60) = 60 mod 10 = 0 ⟶ 1–Collision

Linear probing (k, 1) = (0 + 1) mod 10
= 1 mod 10 = 1 ✓

7) h(70) = 70 mod 10 = 0 → 1 – Collision
Linear Probing (k, 1) = (0 + 1) mod 10

= 1 mod 10 = 1 → 1 – Collision again
Linear probing (k, 2) = (0 + 2) mod 10
= 2 mod 10 = 2 → 1 – Collision again
Linear probing (k, 3) = (0 + 3) mod 10

= 3 mod 10 = 3 ✓

8) h(80) = 80 mod 10 = 0 → 1 – Collision
Linear probing (k, 1) = (0 + 1) mod 10
= 1 mod 10 = 1 → 1 – Collision again
Linear probing (k, 2) = (0 + 2) mod 10
= 2 mod 10 = 2 → 1 – Collision again
Linear probing (k, 3) = (0 + 3) mod 10
= 3 mod 10 = 3 → 1 – Collision again
Linear probing (k, 4) = (0 + 4) mod 10

= 4 mod 10 = 4 ✓

Now, all keys are stored by following linear probing.

Advantages of linear probing:
The main advantage of linear probing is, the each position/the hash index of the hash table can be used to store keys.

Disadvantages of linear probing:

i) While probing, each slot is being checked to get to store some key at it, so it will take more time by probing slowly.

ii) In linear probing, we can probe only in the manner of one slot after another.

iii) Deletion is difficult because once we delete some key, it creates trouble for other keys while any search is performed.

Time complexity:

i)	Searching	:	O(1) [Best case]
			O(S) [Worst case], [Average Case]
ii)	Insertion	:	O(1) [Best case]
			O(S) [Worst case], [Average Case]
iii)	Deletion	:	O(1) [Best case]
			O(S) [Worst case], [Average Case]

Primary clustering:

- In case two keys have the same start of the hash index, then while performing linear probing to resolve the collision, both will follow the same path in a linear manner, which is unnecessary. It is called primary clustering.

- Because of primary clustering, the average searching time increases.
- Linear probing is suffering from primary clustering.

Grey Matter Alert!

Linear probing is also called closed hashing.

Q5 **Consider a hash table with 8 buckets which use linear probing to resolve collisions. The key values are integers, and hash function used is key mod 8. If the keys 55, 133, 50, 123, 62 are inserted in the hash table in order then what is the index of the key 62?**

Sol: 0^{th} position of the hash table.

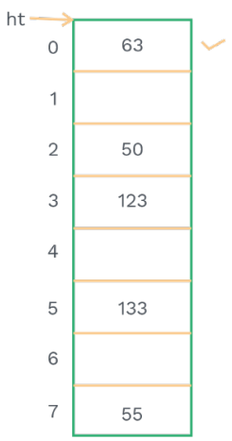
Given, Number of buckets/slots = 8
Means hash table size is 8. (0 to 7)
Hash function = key% 8, (% = mod)
Collision resolution technique = Linear probing.

Mapping keys to their location one by one:
1) h(55) = 55% 8 = 7
2) h(133) = 133% 8 = 5
3) h(50) = 50% 8 = 2
4) h(123) = 123% 8 = 3
5) h(63) = 63% 8 = 7 – 1 collision

Applying linear probing,

$$h(63) = 63\% 8 = 7$$

Linear probing (k, i) = (h(k) + i) mod 8
$\Rightarrow$ Linear probing (k, 1) = (7 + 1) mod 8 = 8 mod 8 = 0
Resulting hash table is:



So, key–63 would be inserted at location '0' of the hash table.

2) Quadratic probing:

- In this method, if some collision arises, then to resolve the collision, we probe the next slot to store the key in a quadratic manner.
- Quadratic_probing(k, i) = (h(k) + i^2) mod S

  $\forall$ i $\in$ {0, 1, 2, 3, ..., S–1}

Where k is a key.
h(k) is the hash of key 'k'.
i is iteration number or collision number.
S is the size of the hash table.

- When any collision is encountered for some key, the above definition is applied to resolve the collision according to quadratic probing.
- In quadratic probing, we probe one slot after another in quadratic manner.

Time complexity:

i) Searching : O(1) [Best case]
O(S) [Worst case], [Average Case]
ii) Insertion : O(1) [Best case]
O(S) [Worst case], [Average Case]
iii) Deletion : O(1) [Best case]
O(S) [Worst case], [Average Case]

- Quadratic probing faces the problem of secondary clustering.

Secondary clustering:

- In case two keys have the same start of the hash index of the hash table while performing quadratic probing to resolve the collision, both will follow the same path in quadratic manner, which is unnecessary. It is called secondary clustering.
- Because of secondary clustering average searching time for a key increases.

Previous Years' Question

Consider a hash table of size seven, with starting index zero, and a hash function (3x + 4) mod 7. Assuming the hash table is initially empty, which of the following are the contents of the table when the sequence 1, 3, 8, 10 is inserted into the table using closed hashing? Note that '_' denotes an empty location in the table.

a) 8, _, _, _, _, _, 10
b) 1, 8, 10, _, _, _, 3
c) 1, _, _, _, _, _, 3
d) 1, 10, 8, _, _, _, 3

Sol: b) **(GATE CSE: 2007)**

Previous Years' Question

Consider a hash table of size 11 that uses open addressing with linear probing. Let h(k) = k mod 11 be the hash function used. A sequence of records with keys

43 36 92 87 11 4 71 13 14 is inserted into an initially empty hash table, the bins of which are indexed from zero to ten. What is the index of the bin into which the last record is inserted?

a) 2 **b)** 4
c) 6 **d)** 7

Sol: d) **(GATE CSE: 2008)**

3) Double hashing:

- In this method, if some collision arises then to resolve the collision, a second hash function to key is applied.
- Double_hashing(k, i) = (hash_1(k) + i × hash_2(k)) mod S
  $\forall$ i $\in$ {0, 1, 2, 3, ..., S–1}

  Where k is a key.
  hash 1 (k) is first hash function which is applied to the key.
  hash 2 (k) is second hash function which is applied to the key.
  i is the iteration number or collision number.
  S is the size of the hash table.
- First hash function is: hash1 (k) = k mod S.
- Second hash function can be taken in many ways by keeping in mind the goodness of the hash function.
- Good second hash function must never evaluate to zero and must make sure all cells should be covered while probing.
- When any collision is encountered for some key, the above definition is applied to resolve the collisions according to the double hashing definition.
- In double hashing, we can probe every slot of the hash table for some key randomly, but every slot should be covered.
- Double_hashing doesn't have the problem of primary clustering or secondary clustering.

Comparison between open addressing and chaining:

i)	Open addressing scheme is complex in consideration of computational work.	**i)**	Chaining is simple.
ii)	All the keys are stored only in the table.	**ii)**	We need extra space outside the hash table to store the keys.
iii)	In open addressing, a key can be sent to a slot which is not mapped for the key.	**iii)**	When keys are not mapped to a slot, then the slot space is wasted.
iv)	Open addressing schemes have problems like primary clustering and secondary clustering.	**iv)**	There is nothing like clustering in chaining.

v)	Hash table may become full.	**v)**	In chaining, hash table never fill up, we can only add keys to chain.
vi)	It provides better cache performance.	**vi)**	Cache performing in chaining is not good.

- Hashing is a searching technique.
- Basically, the motive behind hashing is to retrieve each key stored in hash table in single access.
- Worst case searching time in case of hashing is expected O(1) [Every Case].
- Hash table is used to maintain keys in it.

Chapter Summary

- A function which transforms a key into a table index is called as hash function.
- **Types of hash functions:**
  1) Division modulo method
  2) Digit extraction method
  3) Mid square method
  4) Folding method
- **Load Factor:** Number of keys per slot of the hash table is called load factor. $(\mu); \mu = \frac{x}{N}$, x: Number of keys, N: Number of slots.
- **Collision:** In simple words, if one slot of hash table is getting more than one key to be stored in it, is called as collision.
- **Collision Resolution Techniques:**
  Techniques which can resolve collision, basically are of two kinds.
  i) Chaining:
  This technique resolves collision by building a linked list of all keys hashed to the same hash index.
  ii) Open addressing:
  Open addressing scheme works in three ways.
  a) Linear Probing: Probes the slots lineary to store the keys.
  b) Quadratic Probing: Probes the slots quadratically to store the keys.
  c) Double Hashing: Covers all the slots randomly to store a key and so on.