

UNIT-1 Introduction to C Language - C language elements, variable declarations and data types, operators and expressions, decision statements - If and switch statements, loop control statements - while, for, do-while statements, arrays.

UNIT – 2 Functions, types of functions, Recursion and argument passing, pointers, storage allocation, pointers to functions, expressions involving pointers, Storage classes – auto, register, static, extern, Structures, Unions, Strings, string handling functions, and Command line arguments.

UNIT-3 Data Structures, Overview of data structures, stacks and queues, representation of a stack, stack related terms, operations on a stack, implementation of a stack, evaluation of arithmetic expressions, infix, prefix, and postfix notations, evaluation of postfix expression, conversion of expression from infix to postfix, recursion, queues - various positions of queue, representation of queue, insertion, deletion, searching operations.

UNIT – 4 Linked Lists – Singly linked list, dynamically linked stacks and queues, polynomials using singly linked lists, using circularly linked lists, insertion, deletion and searching operations, doubly linked lists and its operations, circular linked lists and its operations.

UNIT-5 Trees - Tree terminology, representation, Binary trees, representation, binary tree traversals. binary tree operations, Graphs - graph terminology, graph representation, elementary graph operations, Breadth First Search (BFS) and Depth First Search (DFS), connected components, spanning trees. Searching and Sorting – sequential search, binary search, exchange (bubble) sort, selection sort, insertion sort.

Text Books:

1. The C Programming Language, Brian W Kernighan and Dennis M Ritchie, Second Edition, Prentice Hall Publication.
2. Fundamentals of Data Structures in C, Ellis Horowitz, SartajSahni, Susan Anderson-Freed, Computer Science Press.
3. Programming in C and Data Structures, J.R.Hanly, Ashok N. Kamthane and A. AnandaRao, Pearson Education.
4. B.A. Forouzon and R.F. Gilberg, "COMPUTER SCIENCE: A Structured Programming Approach Using C", Third edition, CENGAGE Learning, 2016.
5. Richard F. Gilberg & Behrouz A. Forouzan, "Data Structures: A Pseudocode Approach with C", Second Edition, CENGAGE Learning, 2011.

Reference Books:

1. Pradip Dey and Manas Ghosh, Programming in C, Oxford University Press, 2nd Edition 2011.
2. E. Balaguruswamy, "C and Data Structures", 4th Edition, Tata Mc Graw Hill.
3. A.K. Sharma, Computer Fundamentals and Programming in C, 2nd Edition, University Press.
4. M.T. Somashekara, "Problem Solving Using C", PHI, 2nd Edition 2009.

Introduction to Computers and Problem Solving

Q1. Define Computer.

Answer:

The Computer is an electronic device which operates under the control of instructions stored in its memory and it takes the data from the user (input), a process that data (Processing), gives the result (Output) and stores the result for future use.

Q2. Define input and output.

Answer:

Input: Input is a data that is given by the user to computer.

Output: Output is a data that is given by the computer to user.

Q3. What does a computer consist of?

Answer:

- Every computer mainly consists of three things and those are...
 1. Hardware
 2. Software
 3. User
- Here the user interacts with the software, and the software makes the computer hardware parts to work for the user.

Q4. What is Computer Hardware? Explain different computer hardware components.

Answer:

- The computer hardware is the physical part of a computer.
- The computer hardware components are as follows...

Input Devices - These are the physical components of a computer through which a user can give the data to the computer.

Example: Keyboard, Mouse, Mic, etc...

Output Devices - These are the physical components of a computer through which the computer gives the result to the user.

Example: Monitor, Projector, Speakers, etc...

Storage Devices - These are the physical components of a computer in which the data can be stored.

Example: Hard disk, RAM, ROM, CD, DVD, Pen drive etc...

Devices Drives - Using drives, user can read and write data on to the storage devices like CD/DVD Drive, Floppy Drive, etc...

Cables - Various cables (Wires) are used to make connections in a computer

Other Devices - Other than the above hardware components, a computer also contains components like Motherboard, CPU (Processor), SMPS, Fans, etc.,

Q5. How does Computer work?

Answer:

- The working process of a computer is shown in the following figure.



- When a user wants to communicate with the computer, the user interacts with an application.
- The application interacts with the operating system, and the operating system makes hardware components to work according to the user given instructions.

- The hardware components send the result back to the operating system, then the operating system forwards the same to the application and the application shows the result to the user.
- By using input devices, the user interacts with the application and the application uses output devices to show the result.
- All input and output devices work according to the instructions given by the operating system.

Q6. Define Programming Languages.

Answer:

- Computer Programming languages can communicate and provide instructions to the computers.
- Example: C, Python, C++, Java etc...,

Q7. Explain different types of programming languages.

Answer:

- There are mainly three different languages with the help of which we can develop computer programs. And they are –
 1. Low Level Language (Machine Level language)
 2. Middle Level Language (Assembly Level Language)
 3. High Level Language

1. Low Level Language (Machine Level Language):

- Low-Level language is the only language which can be understood by the computer.
- **Binary Language** is an example of a low-level language.
- Low-level language is also known as **Machine Language**.
- The binary language contains only two symbols 1 & 0. All the instructions of binary language are written in the form of binary numbers 1's & 0's.
- A computer can directly understand the binary language.
- Machine language is also known as the **Machine Code**.
- These are also called as First Generation Languages.

Advantages:

- High speed execution
- The computer can understand instructions immediately
- No translation is needed.

Disadvantages:

- Machine dependent
- Programming is very difficult
- Difficult to understand
- Difficult to write bug free programs
- Difficult to isolate an error

2. Middle Level Language (Assembly Level Language):

- Middle-level language is a computer language in which the instructions are created using symbols such as letters, digits and special characters.
- Assembly language is an example of middle-level language.
- In assembly language, we use predefined words called **mnemonics**.
- But the computer cannot understand mnemonics, so we use a translator called **Assembler** to translate mnemonics into binary language.
- Assembler is a translator which takes assembly code as input and produces machine code as output.
- Assembler is used to translate middle-level language into low-level language.
- These are also called as Second Generation Languages.

Advantages:

- Easy to understand and use
- Easy to modify and isolate error
- High efficiency
- More control on hardware

Disadvantages:

- Machine Dependent Language
- Requires translator
- Difficult to learn and write programs

- Slow development time
- Less efficient

3. High-Level Languages

- A high-level language is a computer language which can be understood by the users.
- The high-level language is very similar to human languages and has a set of grammar rules that are used to make instructions more easily.
- The high-level language is easier to understand for the users but the computer cannot understand it.
- We use **Compiler** or **interpreter** to convert high-level language to low-level language.
- These are also called as Third Generation Languages.
- Example: COBOL, FORTRAN, BASIC, C, C++, JAVA, PYTHON, etc...,

Advantages:

- Easy to write and understand
- Easy to isolate an error
- Machine independent language
- Easy to maintain
- Better readability
- Low Development cost
- Easier to document
- Portable

Disadvantages:

- Needs translator
- Requires high execution time
- Poor control on hardware
- Less efficient

Q8. Define Program

Answer:

A program is a set of instructions.

Q9. Define Instruction.

Answer:

An instruction is a statement that can be formed using language character set.

Q 10. What is C?

Answer:

C is a computer programming language used to design computer software and applications.

Q11. Why do we use C?

Answer:

We use the C programming language to design computer software and applications.

Q12. Who invented C?

Answer:

C Programming Language was invented in the year 1972 by Dennis Ritchie (Dennis MacAlistair Ritchie).

He was an American Computer Scientist who worked at Bell Labs as a researcher along with Ken Thompson.

He was born on 9th September 1941 and lived till 12th October 2011. He is said to be the Father of C.

Q13. Which software is used to create C programs.

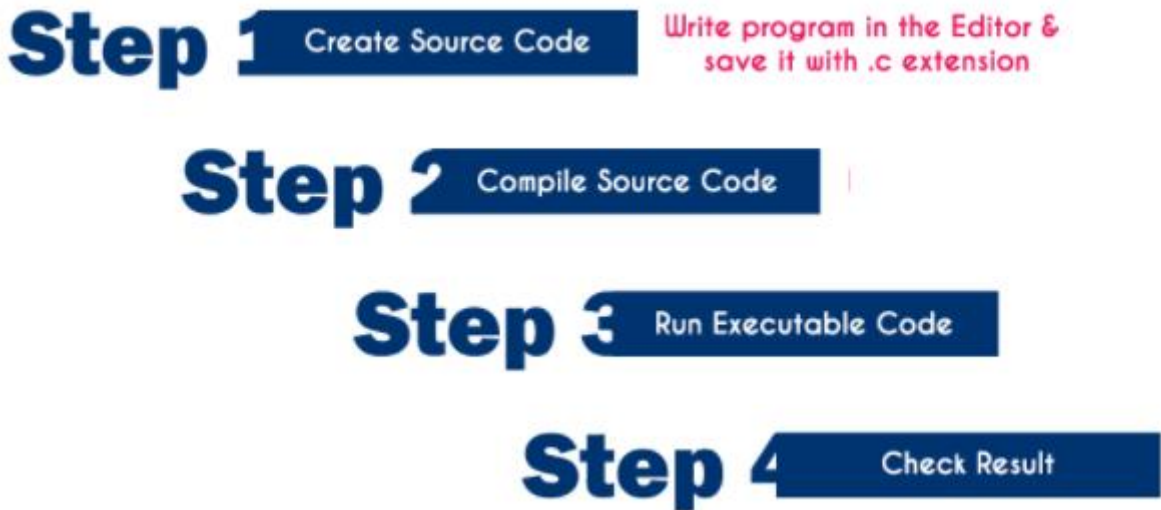
Answer:

Following are the applications and software used to create and execute C programs.

- Turbo C
- Turbo C++
- GNU C
- Code Blocks
- Net Beans
- Dev C++ etc...

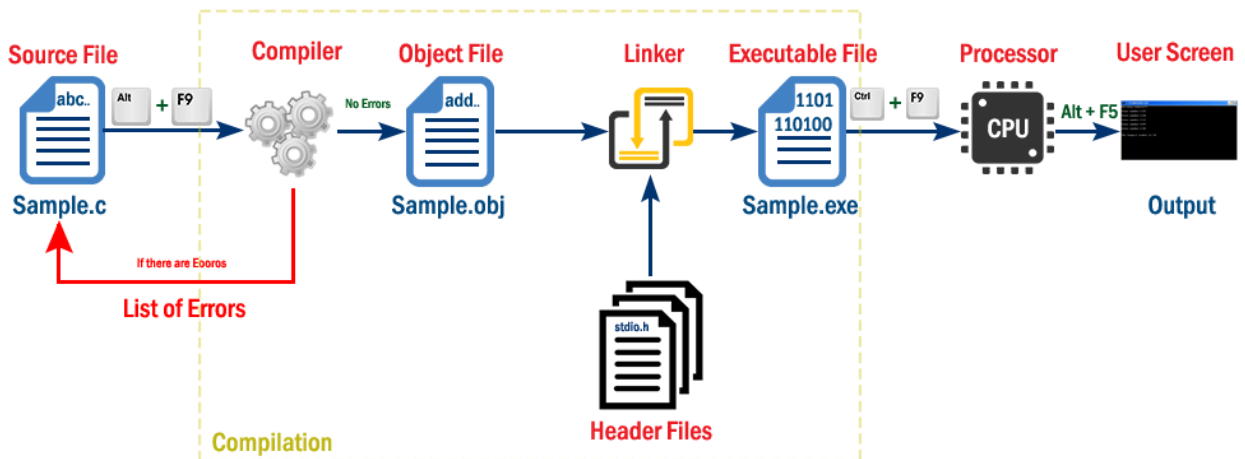
Q14. How to Create and Run C Program?

Answer:



Q15. Explain execution process of a c program.

Answer:



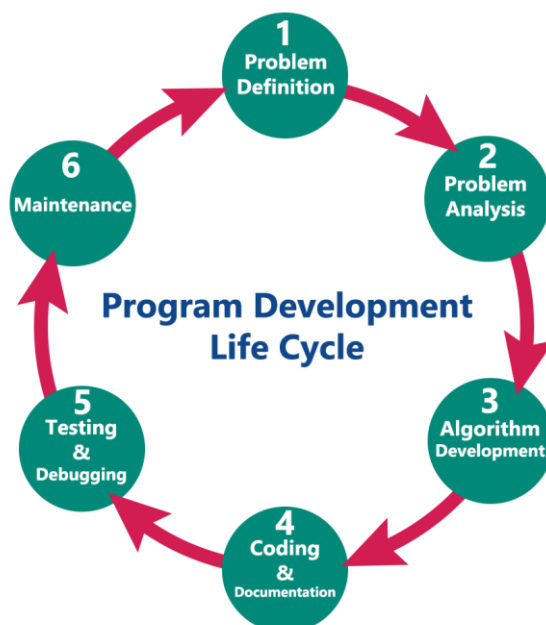
- The file which contains c program instructions in a high-level language is said to be source code.
- Every c program source file is saved with .c extension, for example, Sample.c.
- Whenever we press compilation button the source file is submitted to the compiler.
- Compiler checks for the errors, if there are any errors, it returns a list of errors, otherwise generates object code in a file with name Sample.obj and submit it to the linker.

- The linker combines the code from specified header file into an object file and generates executable file as Sample.exe.
- With this compilation process completes. Now, we need to run the executable file (Sample.exe).
- To run a program, we press run button.
- When we press run button the executable file is submitted to the CPU.
- Then CPU performs the task according to the instructions written in that program and place the result into User Screen.

Q16. Explain Program Development Life Cycle.

Answer:

- Generally, the program development life cycle contains 6 phases, they are as follows....
 1. Problem Definition
 2. Problem Analysis
 3. Algorithm Development
 4. Coding & Documentation
 5. Testing & Debugging
 6. Maintenance



Q17. Write short note on history of C.

Answer:

- C programming language was developed in the year of 1972 by Dennis Ritchie at Bell Laboratories in the USA (AT & T).
- In the year of 1968, research was started by Dennis Ritchie on programming languages like BCPL, CPL.
- The main aim of his research was to develop a new language to create an OS called UNIX.
- After four years of research, a new programming language was created with solutions for drawbacks in languages like BCPL & CPL.
- In the year of 1972, the new language was introduced with the name “Traditional C”.
- The name 'C' was selected from the sequence of previous language 'B' (BCPL) because most of the features of 'c' were derived from BCPL (B language).
- The first outcome of the c language was the UNIX operating system. The initial UNIX OS was completely developed using 'c' programming language.
- The founder of the 'C' language, Dennis Ritchie is known as “Father of C” and also “Father of UNIX”.

Q18. Why C called as a Structured Programming Language.

Answer:

C is a Structured programming language in which program is divided into various modules. Each module can be written separately and together it forms a single C program. This structure makes it easy for testing, maintaining and debugging process.

Q19. Why C called as a Procedural Language.

Answer:

C is a procedural language in which a program is written as a sequence of instructions. User has to specify “what to do” and “how to do”. These instructions are executed in sequential order.

Q20. Why C Called as Case Sensitive Language.

Answer:

C is a case sensitive language in which keywords and everything used in C program are case sensitive.

Q21. Define algorithm. List out its properties.

Answer:

Algorithm:

Algorithm is a step by step process to solve a problem.

Properties of Algorithm:

1. **Input:** An algorithm must accept zero or more input.
2. **Output:** An algorithm must produce at least one output.
3. **Finiteness:** An algorithm must be terminated at finite number of steps.
4. **Definiteness:** Every step of algorithm must be clear and unambiguous.
5. **Effectiveness:** Every step must be basic and easy to convert into program.

Example:

Write an algorithm to perform addition of two numbers

Algorithm:

Step-1: Start

Step-2: Read a number into A

Step-3: Read another number into B

Step-4: Perform addition and store result in total

Step-5: Display total value

Step-6: Stop

Q22. Define pseudocode.

Answer:

Programming language description of algorithm is called as pseudocode.

Example:

Write a pseudocode to perform addition of two numbers

Algorithm:

Step-1: Start

Step-2: Input A

Step-3: Input B

Step-4: total = A + B

Step-5: Print Total






Step-6: Stop

Q23. Define flowchart. List the different symbols used to draw a flowchart.

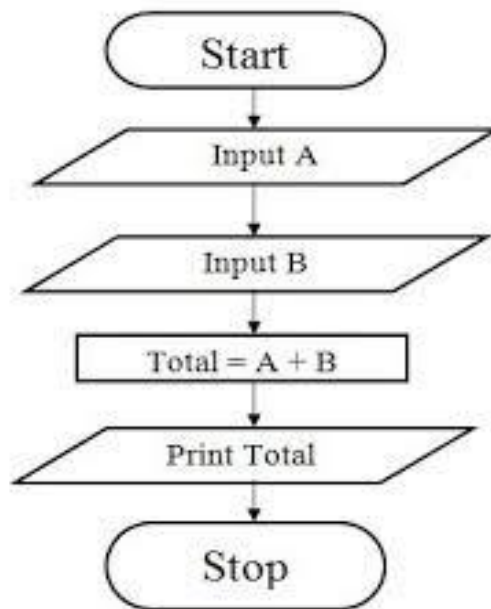
Answer:

Pictorial representation of algorithm is known as flowchart.

Symbols used in flowchart:

Symbol	Name	Function
	Start/end	An oval represents a start or end point.
	Arrows	A line is a connector that shows relationships between the representative shapes.
	Input/Output	A parallelogram represents input or output.
	Process	A rectangle represents a process.
	Decision	A diamond indicates a decision.

Example:



Unit 1**Introduction to C Language**

1. C language elements
2. Variable declarations and data types
3. Operators and expressions
4. Decision statements - If and switch statements
5. Loop control statements - while, for, do-while statements
6. Arrays

1. C language elements

Q 1.1. Explain in detail about structure of C Program.

Answer:

- C is a structured programming language. Every C program and its statements must be in a particular structure.
- Every C program has the following general structure.

1	Documentation Section (Used for Comments)
2	Pre-processing Commends
3	Global Declarations
4	<pre>int main() { Local Declaration Executable Statements . . . return 0; }</pre>
5	User defined functions

1. Documentation Section:

- This section is used to provide a small description of the program.
- The comment lines are simply ignored by the compiler, that means they are not executed.
- In C, there are two types of comments.
 1. **Single Line Comments:** Single line comment begins with // symbol. We can write any number of single line comments.
 2. **Multiple Lines Comments:** Multiple lines comment begins with /* symbol and ends with */. We can write any number of multiple lines comments in a program.
- All the comment lines in a C program just provide the guidelines to understand the program and its code.

2. Pre-processing Commands:

- Pre-processing commands are used to include header files and to define constants.
- Pre-processing commands are beginning with # symbol.
- C Pre-processing commands are: #include, #define, #if, #else etc...,

3. Global Declarations:

- The global declaration is used to define the global variables, which are common for all the functions after its declaration.
- We also use the global declaration to declare functions.

4. Main Function:

- Every C program must contain this main function.
- Here, main () is a user-defined function which tells the compiler that this is the starting point of the program execution.
- The open brace ({) indicates the beginning of the main function.
- Local declaration contains local variables which are used with in main () function.
- Executable statements perform tasks like reading data, displaying the result, calculations, etc.,
- Closing Brace (}) indicates the end of the main () function.
- The statement return 0 returns a value zero to the Operating System after completing the main () execution.
- If we don't want to return any value, we can use it as void.

5. User Defined Functions:

- This is the place where we implement the user-defined functions.

Q1.2. Write short note on C Character set.

Answer:

- A program is a set of statements.
- These statements are constructed using words
- And these words are constructed using characters from C character set.
- C language supports 256 characters.
- C character set contains the following set of characters...
 1. Alphabets
 2. Digits
 3. Special Symbols

1. Alphabets:

- C language supports all the alphabets from the English language.
- Lower and upper case letters together support 52 alphabets.
- lower case letters - **a to z**
- UPPER CASE LETTERS - **A to Z**

2. Digits:

- C language supports 10 digits which are used to construct numerical values.
- Digits - **0, 1, 2, 3, 4, 5, 6, 7, 8, 9**

3. Special Symbols:

- C language supports a rich set of special symbols that include symbols to perform mathematical operations, to check conditions, white spaces, backspaces, and other special symbols.
- Example: **+, -, *, /, @, #, \$, %, etc...**,

Q1.3. Define a C-token. Explain different types of C-tokens?**Answer:**

- Every smallest unit of a c program is called token.
- Every instruction in a c program is a collection of tokens.
- Tokens are used to construct c programs and they are said to be the basic building blocks of a C program.
- There are 5 types of C-tokens. They are:
 1. Keywords
 2. Identifiers
 3. Special Symbols
 4. Constants
 5. Strings
 6. Operators

1. Keywords:

- Keywords are the reserved words with predefined meaning which are already known to the compiler.
- In the C programming language, there are **32 keywords**.

C- Keywords											
	Data Types		User Defined Data Types		Condition Statements		Looping Statements		Storage Classes		Others
1	Int	10	struct	14	if	18	Do	23	auto	27	const
2	char	11	union	15	else	19	while	24	static	28	default
3	double	12	enum	16	switch	20	For	25	register	29	goto
4	float	13	typedef	17	case	21	break	26	extern	30	return
5	long					22	continue			31	sizeof
6	short									32	volatile
7	signed										
8	unsigned										
9	void										

2. Identifiers:

- An identifier is a name used to identify a variable, function, structure, etc.
- In C length of identifier is 31 characters.

Rules to define Identifiers:

1. Identifiers can be a combination of letters (a to z/ A to Z) or digits (0 to 9) or an underscore (_).

Example: myClass, var_1, print_this_to_screen, _number are valid identifiers.

2. An identifier can start with a letter or an underscore (_), but not with a digit.

Example: 1_variable is invalid

Variable_1 is Valid

3. Keywords cannot be used as identifiers.

Example: if is invalid identifier

4. Special symbols like !, @, #, \$, % etc. are not allowed in identifiers except one special symbol underscore (_).

Example: company#name, \$name, email@id are invalid identifiers

3. Special Symbols:

- Some special characters are used in C, and they have a special meaning which cannot be used for another purpose.
 1. Square brackets []: Used in Arrays
 2. Simple brackets (): Used in Functions
 3. Curly braces { }: Used in the opening and closing of the block.
 4. Comma (,): It is a separator.
 5. Hash/pre-processor (#): It is used for pre-processor directive.
 6. Asterisk (*): This symbol is used to represent pointers
 7. Period (.): It is used to access a member of a structure or a union.

4. Constants:

- A constant is a value assigned to the variable which will cannot be changed.
- There are two ways of declaring constant:

1. Using const keyword

2. Using #define pre-processor

Types of constants in C:

Constant	Example
Integer constant	10, 11, 34, etc.
Floating-point constant	45.6, 67.8, 11.2, etc.
Octal constant	011, 088, 022, etc.
Hexadecimal constant	0x1a, 0x4b, 0x6b, etc.
Character constant	'a', 'b', 'c', etc.
String constant	"java", "c++", ".net", etc.

5. Strings:

- Strings in C are always represented as an array of characters having null character '\0' at the end of the string.
- This null character denotes the end of the string.
- Strings in C are enclosed within double quotes, while characters are enclosed within single characters.
- The size of a string is a number of characters that the string contains.

6. Operators:

- Operator is a special symbol used to perform the operation.
- The data items on which the operators are applied are known as operands.
- Operators are applied between the operands.
- Depending on the number of operands, operators are classified as follows:

1. Unary Operator

- A unary operator is an operator applied to the single operand.
- Example: Unary Minus (-), not (!), increment operator (++), decrement operator (--), sizeof etc...,

2. Binary Operator:

- The binary operator is an operator applied between two operands.

The following is the list of the binary operators:

1. Arithmetic Operators
2. Relational Operators
3. Logical Operators

4. Bitwise Operators
5. Conditional Operator
6. Assignment Operator

3. Ternary Operator:

- The ternary operator is an operator applied between three operands.
- Ternary operator in c is conditional operator(?:)

Q1.4. Define Data Type. List out different Data Types in C.

Answer:

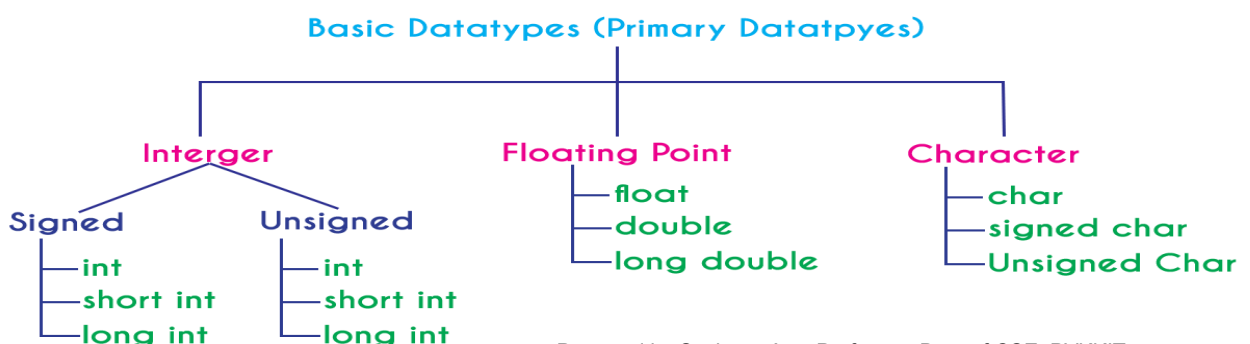
- A data type specifies the type of data that a variable can store such as integer, floating, character, etc.
- There are the following data types in C language.

Types	Data Types
Basic Data Type (Primitive Data Type)	integer, floating point, double and character
Derived Data Type (User Defined Data Type)	array, pointer, structure, union
Enumeration Data Type	enum
Void Data Type	void

Q1.5. Give brief discussion on primitive data types.

Answer:

- The primary data types in the C programming language are the basic data types.
- Primary data types are also called as Built-In data types.
- The following are the primary data types in c programming language.



1. Integer:

- The integer data type is a set of whole numbers.
- Every integer value does not have the decimal value.
- We use the keyword "**int**" to represent integer data type in C.
- The following table provides complete details about the integer data type according to the 32-bit architecture.

Type	Size	Range (For Signed: -2^{N-1} to $2^{N-1} - 1$, For Unsigned: 0 to $2^N - 1$, Here N is number of bits)
int	2 bytes	-32,768 to 32,767
unsigned int	2 bytes	0 to 65,535
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
long	4 bytes	-2^{32-1} to $2^{32-1} - 1$
unsigned long	4 bytes	0 to $2^{32} - 1$

2. Floating Point:

- Floating-point data types are a set of numbers with the decimal value.
- We use the keyword "**float**" to represent floating-point data type and "**double**" to represent double data type in C.
- The following table provides complete details about floating-point data types.

Type	Size	Range (-2^{N-1} to $2^{N-1} - 1$, Here N is number of bits)	Precision
float	4 byte	-2^{32-1} to $2^{32-1} - 1$	6 decimal places
double	8 byte	-2^{64-1} to $2^{64-1} - 1$	15 decimal places
long double	10 byte	-2^{80-1} to $2^{80-1} - 1$	19 decimal places

3. Character:

- The character data type is a set of characters enclosed in single quotations.
- The following table provides complete details about the character data type.

Type	Size	Range (For Signed: -2^{N-1} to $2^{N-1} - 1$, For Unsigned: 0 to $2^N - 1$, Here N is number of bits)
char (signed char)	1 byte	-128 to 127
unsigned char	1 byte	0 to 255

Q1.6. Write short note on format specifiers in C.

Answer:

- The format specifiers are used in C for input and output purposes.
- Using this concept the compiler can understand that what type of data will be taking input using the scanf() function and printing using printf() function.
- Here is a list of format specifiers.

Format Specifier	Type
%c	Character
%d	Signed integer
%e or %E	Scientific notation of floats
%f	Float values
%g or %G	Similar as %e or %E
%hi	Signed integer (short)
%hu	Unsigned Integer (short)
%i	Unsigned integer
%l or %ld or %li	Long
%lf	Double
%Lf	Long double
%lu	Unsigned int or unsigned long

%lli or %lld	Long long
%llu	Unsigned long long
%o	Octal representation
%p	Pointer
%s	String
%u	Unsigned int
%x or %X	Hexadecimal representation
%n	Prints nothing
%%	Prints % character

2. Variable declarations and data types

Q2.1. Define variable. Explain how to declare and initialize a variable in C.

Answer:

Variable:

A variable is a name of the memory location which can store a value.

Declaration:

Declaration of a variable tells the compiler to allocate the required amount of memory with the specified variable name and allows only specified datatype values into that memory location.

We can declare a variable using below syntax.

Syntax:

```
Datatype Variable_name;
```

Example:

```
int number;
```

The above declaration tells to the compiler that allocates **2 bytes** of memory with the name **number** and allows only **integer values** into that memory location.

Initialization:

Initialization of a variable means assigning a value to the variable.

Prepared by Sushma.,Asst.Professor.,Dept of CSE.,PVKKIT

We can initialize a variable with a value using below syntax.

Syntax:

```
Variable_name = Value
```

Example:

```
int number;  
number = 10;
```

In the above example, value 10 is assigned to the variable number.

Q2.2. What are the rules to define a variable.

Answer:

Rules to define Identifiers:

1. Variables can be a combination of letters (a to z/ A to Z) or digits (0 to 9) or an underscore (_).

Example: myClass, var_1, print_this_to_screen, _number are valid variables.

2. A variable can start with a letter or an underscore (_), but not with a digit.

Example: 1_variable is invalid

Variable_1 is Valid

3. Keywords cannot be used as variables.

Example: if is invalid variable

4. Special symbols like !, @, #, \$, % etc. are not allowed in variables except one special symbol underscore (_).

Example: company#name, \$name, email@id are invalid variables

Q2.3. What is the use of void data type.

Answer:

Void data type:

- The void data type means nothing or no value.
- Generally, the void is used to specify a function which does not return any value.
- We also use the void data type to specify empty parameters of a function.

3. Operators and expressions

Q3.1. Define Operator. Explain different operators in C.

Answer:

Operator:

Operator is symbol is used perform operation in between operands.

Different types of Operators:

C operators are classified as:

1. Arithmetic Operators
2. Relational Operators
3. Logical Operators
4. Assignment Operator
5. Increment & Decrement Operators
6. Bitwise Operators
7. Conditional Operator
8. Special Operators

1. Arithmetic Operators:

- The arithmetic operators are used to perform basic mathematical operations like addition, subtraction, multiplication, division and percentage modulo.
- The following table provides information about arithmetic operators.

Operator	Name	Example	Result
+	Addition	10 + 5	15
-	Subtraction	10 - 5	5
*	Multiplication	10 * 5	50
/	Division	10 / 5	2
%	Modulo Division (Gives the Remainder for integer division)	5 % 2	1

2. Relational Operators:

- The relational operators are used to check the relationship between two values.
- Every relational operator has two results TRUE or FALSE.

- In simple words, the relational operators are used to define conditions in a program.
- The following table provides information about relational operators.

Operator	Name	Example	Result
<	Less than	10 < 5	False
>	Greater than	10 > 5	True
<=	Less than or equal to	10 <= 5	False
>=	Greater than equal to	10 >= 5	True
==	Equal to	10 == 5	False

3. Logical Operators:

- The logical operators are used to check logical relationship between two conditions.
- The following table provides information about logical operators.

Operator	Name	Meaning	Example	Result
&&	Logical AND	Returns True, if both conditions are True. Otherwise False	10 < 5 && 12 > 10	False
	Logical OR	Return False, If both conditions are False Otherwise True	10 < 5 12 > 10	True
!	Logical NOT	Return False, if condition is True and vice versa	!(10 < 5 && 12 > 10)	True

4. Assignment Operator:

- Assignment operator is used to assigning values to the variables.

Example:

a= 5

Shorthand Assignment Operators:

The following table illustrates different types of shorthand operators.

Operator	Example (A = 10)	Equivalent equation	Resultant value of A
+=	A += 5	A = A + 5	15
-=	A -= 5	A = A - 5	10
*=	A *= 5	A = A * 5	50
/=	A /= 5	A = A / 5	2
%=	A %= 5	A = A % 5	0

5. Increment & Decrement Operators:

- Increment operator is used to increase the value of variable by one.
- Decrement operator is used to decrease the value of variable by one.
- We can use these operators before or after variable name.
- The below table illustrates the different increment and decrement operators.

Operator	Meaning	Example A = 5	Resultant value of A
A++	Post Increment	A++	6
++A	Pre Increment	++A	6
A--	Post Decrement	A--	4
--A	Pre Decrement	--A	4

6. Bitwise Operators

- The bitwise operators are used to perform bit-level operations in the C.
- When we use the bitwise operators, the operations are performed based on the binary values.

- The following table describes all the bitwise operators in the C.

Operator	Name	Meaning	Example (A = 25, B = 20)	Result
&	Bitwise AND	Return 1 if both the bits are 1 otherwise it is 0	A & B	16
	Bitwise OR	Return 0 if both bits are 0 otherwise it is 1	A B	29
^	Bitwise XOR	Return 0 if all the bits are same otherwise it is 1	A ^ B	13
~	Bitwise NOT	Return 0 if the bit is 0	~A	6
<<	Left shift	Shifts all the bits to the left by the specified number of positions	A << 2	100
>>	Right shift	Shifts all the bits to the right by the specified number of positions	A >> 2	6

7. Conditional Operator:

- The conditional operator is also called a **ternary operator** because it requires three operands.
- This operator is used for decision making.

Syntax:

Condition? TRUE Part : FALSE Part;

- If the condition is TRUE the TRUE Part is performed, if the condition is FALSE the FALSE Part is performed.

Example:

A = (10<15) ? 100 : 200;

⇒ A value is 100

8. Special Operators

- The following are the special operators in C.

1. sizeof operator:

- This operator is used to find the size of the memory (in bytes) allocated for a variable.

Syntax:

sizeof(variableName)

Example:

int a;

sizeof(a); ⇒ the result is 2

2. Pointer operator (*):

- This operator is used to define pointer variables.

3. Comma operator (,):

- This operator is used to separate variables while they are declaring, separate the expressions in function calls, etc.

4. Dot operator (.) and Member operator (->):

- This operator is used to access members of structure or union.

Q3.2. What is an expression? Explain different types of Expressions?

Answer:

Expression:

An expression is a collection of operators and operands that represents a specific value.

Example:

C = A + B

Expression Types:

There are 3 types of expressions.

1. Infix Expression
2. Postfix Expression
3. Prefix Expression

1. Infix Expression:

In Infix expression the operator is used between operands.

Example:

A + B

2. Postfix Expression:

In Postfix expression the operator is used after operands.

Example:

A B +

3. Pre fix Expression:

In Infix expression the operator is used after operands.

Example:

+ A B

Q3.3. What is Operator Precedence?

Answer:

Operator precedence is used to determine the order of operators evaluated in an expression. In c programming language every operator has precedence (priority).

When there is more than one operator in an expression the operator with higher precedence is evaluated first and the operator with the least precedence is evaluated last.

Q3.4. What is Operator Associativity?

Answer:

Operator associativity is used to determine the order of operators with equal precedence evaluated in an expression.

When an expression contains multiple operators with equal precedence, we use associativity to determine the order of evaluation of those operators.

Q3.5. Draw the table of operator precedence and associativity.**Answer:**

Precedence	Operator	Operator Meaning	Associativity
1	() [] -> .	function call array reference structure member access structure member access	Left to Right
2	! ~ + - ++ -- & * sizeof (type)	negation Bitwise Not Unary plus Unary minus increment operator decrement operator address of operator pointer returns size of a variable type conversion	Right to Left
3	* / %	multiplication division remainder	Left to Right
4	+ -	addition subtraction	Left to Right
5	<< >>	left shift right shift	Left to Right
6	< <=	less than less than or equal to	Left to Right

	> >=	greater than greater than or equal to	
7	== !=	equal to not equal to	Left to Right
8	&	bitwise AND	Left to Right
9	^	bitwise EXCLUSIVE OR	Left to Right
10		bitwise OR	Left to Right
11	&&	logical AND	Left to Right
12		logical OR	Left to Right
13	?:	conditional operator	Left to Right
14	= *= /= %= += -= &= ^= = <<= >>=	assignment assign multiplication assign division assign remainder assign addition assign subtraction assign bitwise AND assign bitwise XOR assign bitwise OR assign left shift assign right shift	Right to Left
15	,	separator	Left to Right

Q3.6. How the expression is evaluated.**Answer:**

An expression is evaluated based on the precedence and associativity of the operators in that expression.

Example:

$$10 + 4 * 3 / 2$$

In the above expression, there are three operators **+**, ***** and **/**. Among these three operators, both multiplication and division have the same higher precedence and addition has lower precedence. So, according to the operator precedence both multiplication and division are evaluated first and then the addition is evaluated.

As multiplication and division have the same precedence they are evaluated based on the associativity. Here, the associativity of multiplication and division is **left to right**. So, multiplication is performed first, then division and finally addition. So, the above expression is evaluated in the order of *** / and +**. It is evaluated as follows...

$$4 * 3 \implies 12$$

$$12 / 2 \implies 6$$

$$10 + 6 \implies 16$$

The expression is evaluated to **16**.

Q3.7. Write short note Type Casting and Conversion in C.**Answer:**

In a c programming language, the data conversion is performed in two different methods as follows...

1. Type Conversion
2. Type Casting

1. Type Conversion:

The type conversion is the process of converting a data value from one data type to another data type automatically by the compiler.

Sometimes type conversion is also called **implicit type conversion**. The implicit type conversion is automatically performed by the compiler.

Example:

```
int i = 10 ;
float x = 15.5 ;
char ch = 'A' ;
```

`i = x ;` =====> x value 15.5 is converted as 15 and assigned to variable i

`x = i ;` =====> Here i value 10 is converted as 10.000000 and assigned to variable x

`i = ch ;` =====> Here the ASCII value of A (65) is assigned to i

2. Type Casting:

Typecasting is also called an **explicit type conversion**. Compiler converts data from one data type to another data type implicitly. When compiler converts implicitly, there may be a data loss.

In such a case, we convert the data from one data type to another data type using explicit type conversion. To perform this, we use the **unary cast operator**.

To convert data from one type to another type we specify the target data type in parenthesis as a prefix to the data value that has to be converted.

The general syntax of typecasting is as follows.

```
(TargetDatatype) DataValue
```

Example:

```
int totalMarks = 450, maxMarks = 600 ;
float average ;
average = (float) totalMarks / maxMarks * 100 ;
```

In the above example code, both totalMarks and maxMarks are integer data values. When we perform totalMarks / maxMarks the result is a float value, but the destination (average) datatype is a float. So we use type casting to convert totalMarks and maxMarks into float data type.

4. Decision statements - If and switch statements

Q4.1. Define a statement. What are the different types of statements?

Answer:

A Statement is an instruction in a program that can be formed using C-Tokens. There are 3 types of statements.

1. Sequential Statements
 2. Conditional Statements (Decision Statements)
 3. Looping Statements.
- In Sequential, each and every statement is executed one by one without skipping.
 - In Conditional, particular block of statements is executed based on condition. Conditional statements in C are: if and switch
 - In Looping, a block of statements is executed up to number of times based on condition. Looping statements in C are: do-while, while and for.

Q4.2. Define null statement. What is the importance of it?

Answer:

Null statement means it executes nothing in the program that means it performs no operation.

Example:

; → it is a null statement

Importance:

It is useful when the syntax of the language calls for a statement but no expression evaluation.

Q4.3. Explain in detail about if statement with example.

Answer:

There are 3 types of if statement. They are:

1. Simple if
2. if-else statement
3. if-else-if statement (else-if ladder)

1. Simple if:

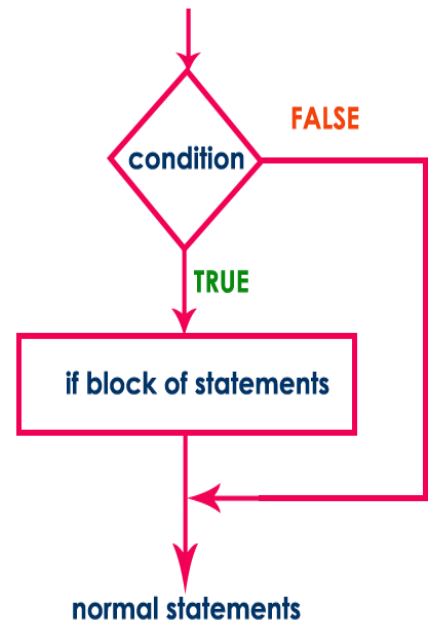
In simple if, if the condition is true, then True block statements are executed.

Simple if does not concentrate on the false condition.

Syntax

```
if ( condition )  
{  
    ....  
    block of statements;  
    ....  
}
```

Execution flow diagram



Example Program | Test whether given number is divisible by 5.

```
#include<stdio.h>  
  
int main()  
{  
    int n ;  
  
    printf("Enter any integer number: ");  
  
    scanf("%d", &n) ;  
  
    if ( n%5 == 0 )  
    {  
        printf("Given number is divisible by 5\n") ;  
    }  
  
    return 0;  
}
```


2. if-else statement:

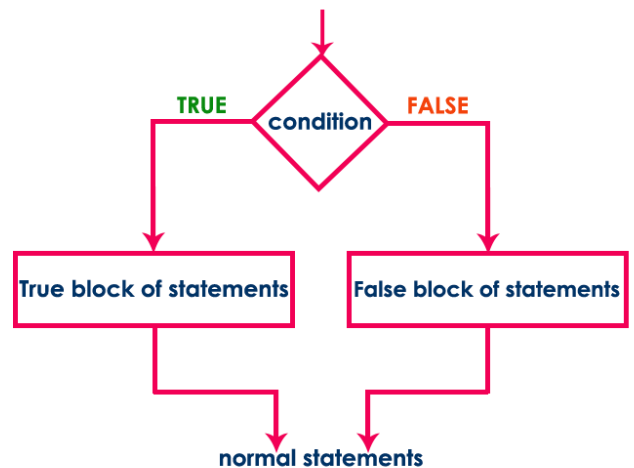
In if-else-if statement, if the condition is true then True block statements are executed otherwise False block statements are executed.

Syntax

```

if ( condition )
{
....
True block of statements;
....
}
else
{
....
False block of statements;
....
}

```

Execution flow diagram**Example Program | Test whether given number is even or odd.**

```

#include<stdio.h>
int main()
{
    int n ;
    printf("Enter any integer number: ");
    scanf("%d", &n) ;
    if ( n%2 == 0 )
        printf("Given number is EVEN\n");
    else
        printf("Given number is ODD\n");
    return 0;
}

```

Output:

```

Enter any integer number: 65
Given number is ODD

```

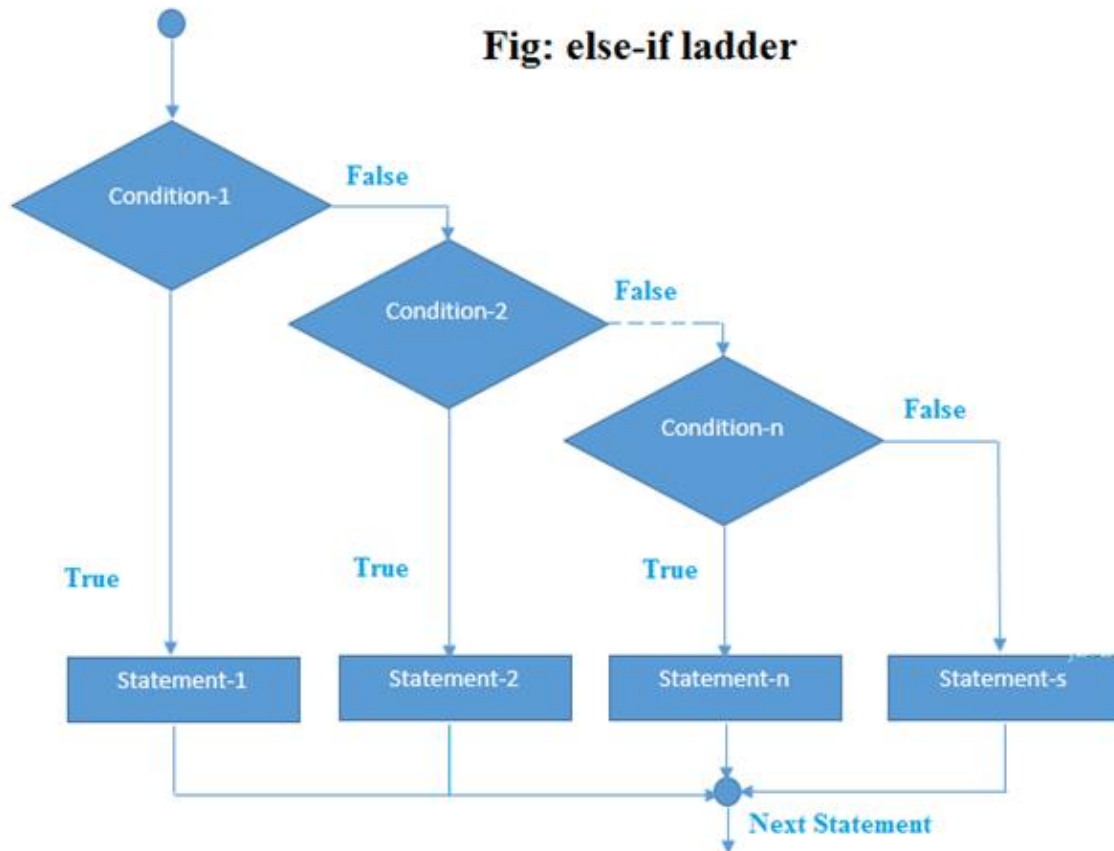
3. if-else-if statement (else-if ladder):

It is used in the scenario where there are multiple cases to be performed for different conditions.

In if-else-if ladder statement, if a condition is true then the statements defined in the if block will be executed, otherwise if some other condition is true then the statements defined in the else-if block will be executed, at the last if none of the condition is true then the statements defined in the else block will be executed.

Syntax:

```
if(condition1){
//code to be executed if condition1 is true
}else if(condition2){
//code to be executed if condition2 is true
}
else if(condition3){
//code to be executed if condition3 is true
}
...
else{
//code to be executed if all the conditions are false
}
```

Flowchart:**Example Program | Find the largest of three numbers.**

```

#include<stdio.h>
int main()
{
    int a, b, c ;
    printf("Enter any three integer numbers: ");
    scanf("%d%d%d", &a, &b, &c);
    if( a>=b && a>=c)
        printf("%d is the largest number", a);
    else if (b>=a && b>=c)
        printf("%d is the largest number", b);
    else
        printf("%d is the largest number", c);
    return 0;
}
  
```

Output:

Enter any three integer numbers:

35

65

41

65 is the largest number

Q4.4. Write short note on nested-if statement.

Answer:

Nested-if statement:

Writing an if statement inside another if statement is called nested if statement.

Syntax

```

if ( condition1 )
{
    if ( condition2 )
    {
        ....
        True block of statements 1;
    }
    ....
}
else
{
    False block of condition1;
}

```

Example Program | Test whether given number is even or odd if it is below 100.

```
#include<stdio.h>
```

```
int main()
```

```
{
```

```
    int n ;
```

```
    printf("Enter any integer number: ");
```

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

Prepared by Sushma.,Asst.Professor.,Dept of CSE.,PVKKIT

```
if ( n < 100 )
{
    printf("Given number is below 100\n") ;
    if( n%2 == 0)
        printf("And it is EVEN") ;
    else
        printf("And it is ODD") ;
}
else
    printf("Given number is not below 100") ;
return 0;
}
```

Output:

Enter any integer number: 25

Given number is below 100

And it is ODD

Q4.5. Explain in detail about switch statement.

Answer:

Switch statement:

The switch statement in C is an alternate to if-else-if ladder statement which allows us to execute multiple operations for the different possible values of a single variable called switch variable.

Here, we can define various statements in the multiple cases for the different values of a single variable.

The syntax of switch statement is given below;

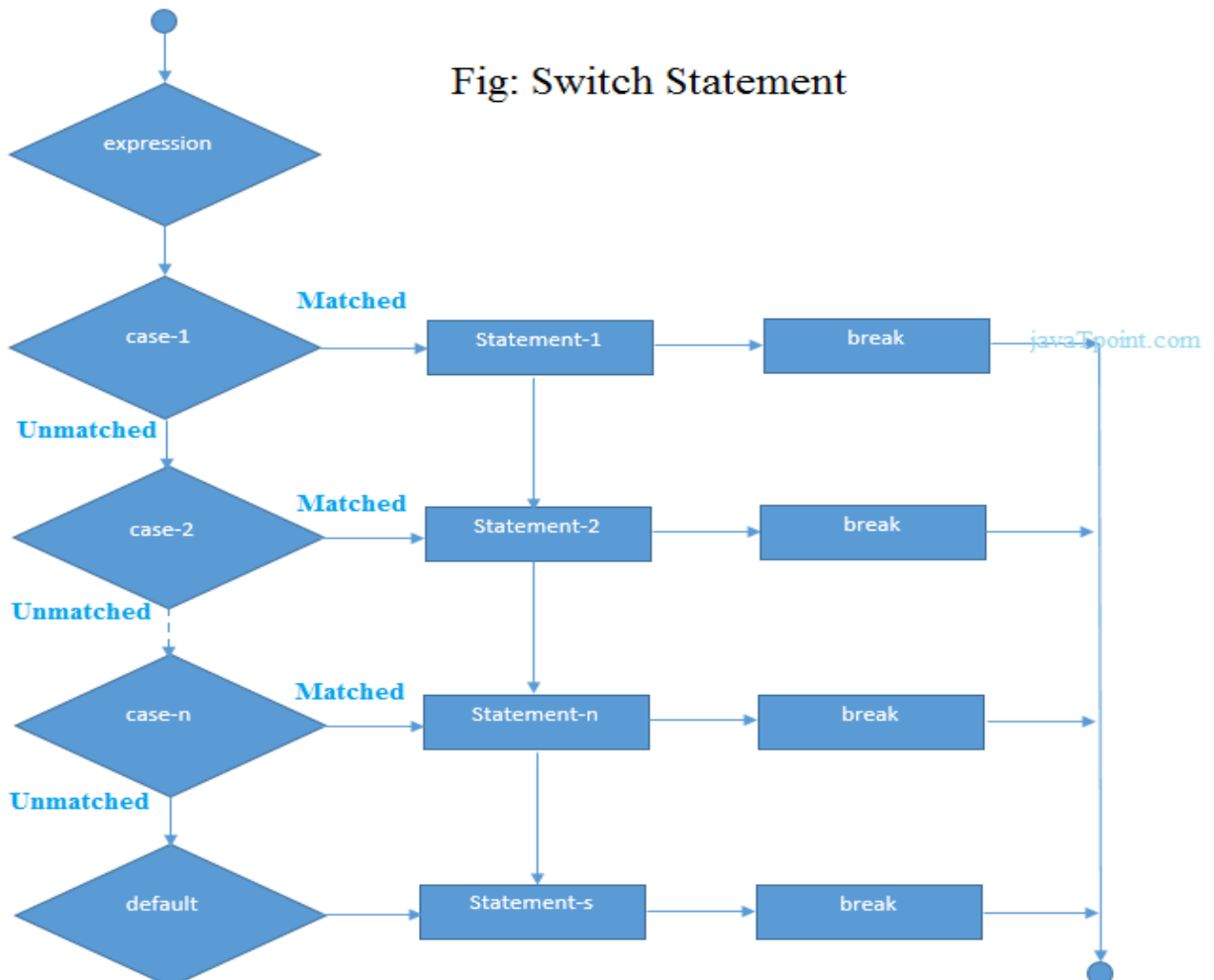
Syntax:

```

switch(expression){
case value1:
//code to be executed;
break; //optional
case value2:
//code to be executed;
break; //optional
.....

default:
code to be executed if all cases are not matched;
}

```

Flowchart:

Example Program | Display pressed digit in words.

```
#include<stdio.h>
int main()
{
    int n ;
    printf("Enter any digit: ");
    scanf("%d", &n) ;
    switch( n )
    {
        case 0:
            printf("ZERO") ;
            break ;

        case 1:
            printf("ONE") ;
            break ;

        case 2:
            printf("TWO") ;
            break ;

        case 3:
            printf("THREE") ;
            break ;

        case 4:
            printf("FOUR") ;
            break ;

        case 5:
            printf("FIVE") ;
            break ;

        case 6:
            printf("SIX") ;
            break ;
```

```

case 7:
    printf("SEVEN");
    break ;

case 8:
    printf("EIGHT");
    break ;

case 9:
    printf("NINE");
    break ;

default:
    printf("Not a Digit");

}

return 0;
}

```

Output:
Enter any digit: 5
FIVE

5. Loop control statements - while, for, do-while statements:

Q5.1. Explain in detail about looping statements.

Answer:

The looping statements are used to execute a single statement or block of statements repeatedly until the given condition is FALSE.

C has 3 looping statements:

1. do-while
2. while
3. for

1. do-while statement:

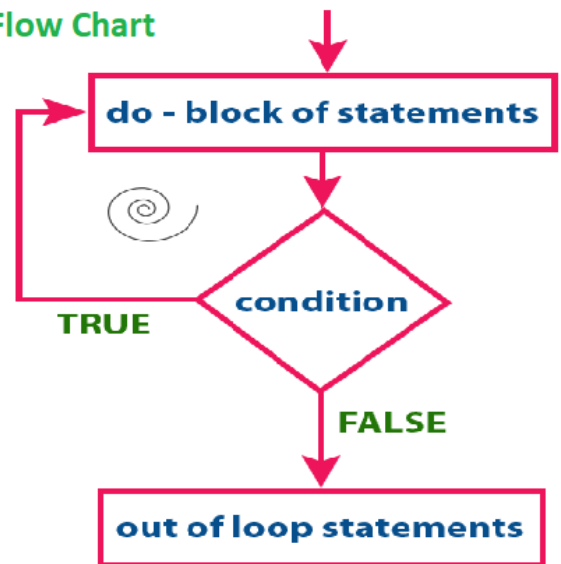
- The do-while statement is used to execute a single statement or block of statements repeatedly until given condition is False.
- The do-while statement is also known as the **Exit control looping statement**.
- In do-while, the block of statements is executed at least once.

Syntax:

```

do
{
    ...
    block of statements;
    ...
} while( condition ) ;

```

Flow Chart**Example Program | Program to display even numbers upto 10.**

```

#include<stdio.h>
int main()
{
    int n = 0;
    printf("Even numbers upto 10\n");
    do
    {
        if( n%2 == 0)
            printf("%d\t", n);
        n++;
    }while( n <= 10 );
}

```

Output:

```

Even numbers upto 10
0  2  4  6  8  10

```

2. while statement:

- The while statement is used to execute a single statement or block of statements repeatedly until given condition is False.
- The while statement is also known as the **Exit control looping statement**.

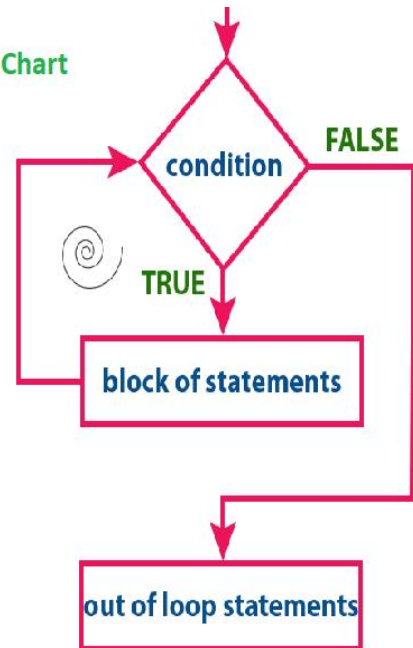
Syntax:

```

while( condition )
{
    ...
    block of statements;
    ...
}

```

Flow Chart



Example Program | Program to display even numbers upto 10.

```

#include<stdio.h>
int main()
{
    int n = 0;
    printf("Even numbers upto 10\n");
    while( n <= 10 )
    {
        if( n%2 == 0)
            printf("%d\t", n);
        n++;
    }
}

```

Output:

```

Even numbers upto 10
0  2  4  6  8  10

```

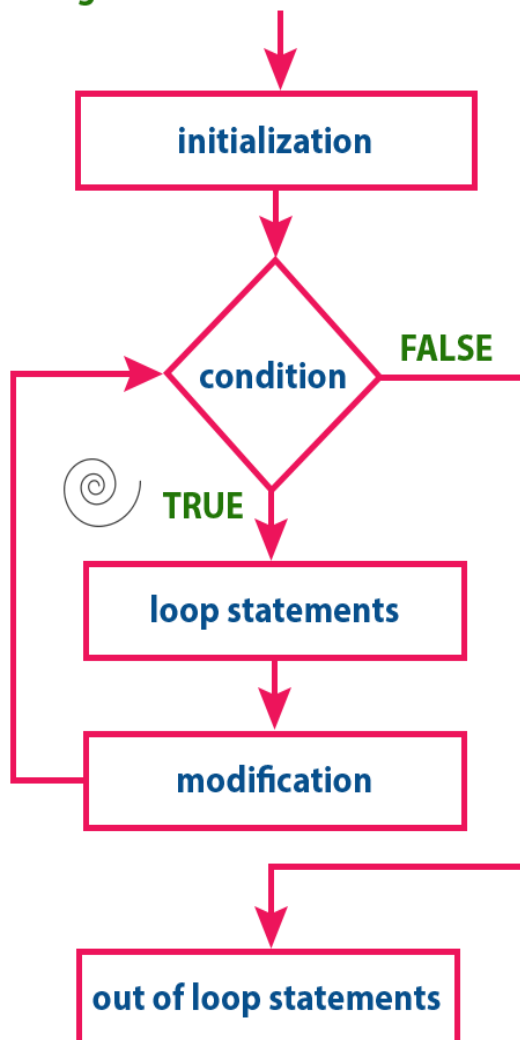
3. for statement:

- The for statement is used to execute a single statement or block of statements repeatedly until given condition is False.

Syntax:

```
for( initialization ; condition ; modification )  
{  
    ...  
    block of statements;  
    ...  
}
```

Execution flow diagram:



Example Program | Program to display even numbers upto 10.

```
#include<stdio.h>
int main()
{
    int n ;
    printf("Even numbers upto 10\n");
    for( n = 0 ; n <= 10 ; n++ )
    {
        if( n%2 == 0)
            printf("%d\t", n) ;
    }
    return 0;
}
```

Output:

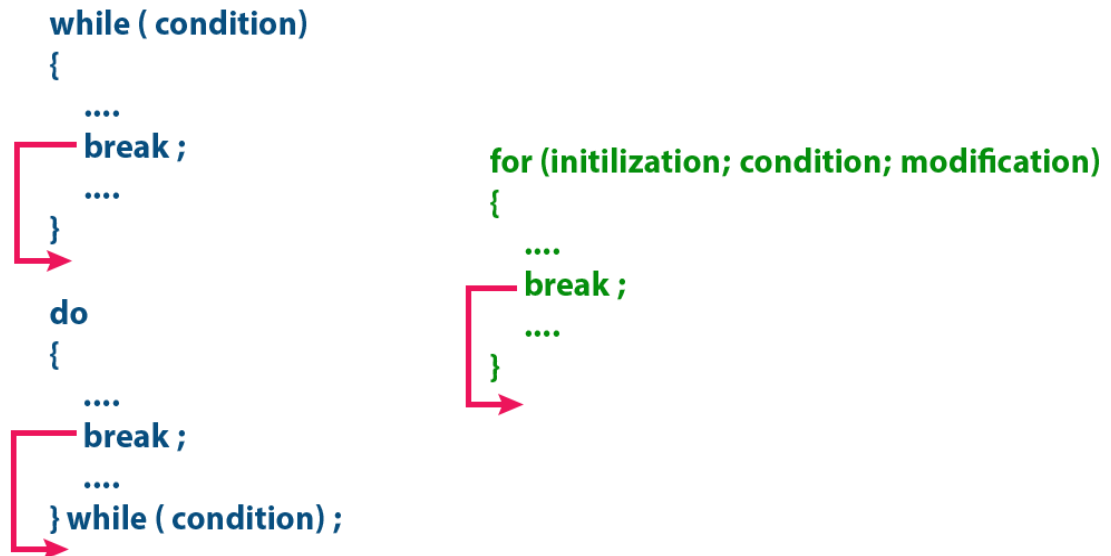
```
Even numbers upto 10
0  2  4  6  8  10
```

Q.5.2. Write the difference between do-while and while.**Answer:**

while	do-while
Condition is checked first then statement(s) is executed.	Statement(s) is executed atleast once, thereafter condition is checked.
It might occur statement(s) is executed zero times, If condition is false.	At least once the statement(s) is executed.
No semicolon at the end of while. while(condition)	Semicolon at the end of while. while(condition);
If there is a single statement, brackets are not required.	Brackets are always required.
Variable in condition is initialized before the execution of loop.	variable may be initialized before or within the loop.
while loop is entry controlled loop.	do-while loop is exit controlled loop.
while(condition) { statement(s); }	do { statement(s); }while(condition);

Q5.3. Write short note on break statement.**Answer:**

- In C, the break statement is used to perform the following two things:
 1. break statement is used to terminate the switch case statement
 2. break statement is also used to terminate looping statements like while, do-while and for.
- The break statement execution is as shown in the following figure.

**Example Program | Program break statement.**

```
#include<stdio.h>
int main()
{
    int i;
    for( i = 0; i <= 10; i++)
    {
        if( i == 5)
            break;
        printf("%d\t",i);
    }
    return 0;
}
```

Output:

```
0  1  2  3  4
```

Q5.4. Write short note on continue statement.**Answer:**

- The **continue** statement is used to move the program execution control to the beginning of the looping statement.
- The **continue** statement can be used with looping statements like while, do-while and for.
- When we use **continue** statement with **while** and **do-while** statements the execution control directly jumps to the condition.
- When we use **continue** statement with **for** statement the execution control directly jumps to the modification portion (increment/decrement/any modification) of the for loop.
- The **continue** statement execution is as shown in the following figure.

```

while ( condition)
{
  ...
  continue;
  ...
}
do
{
  ...
  continue;
  ...
} while ( condition);

for (initilization; condition; modification)
{
  ...
  continue;
  ...
}

```

Example Program | Program on continue statement

```

#include<stdio.h>
int main()
{
    int i;
    for( i = 0; i <= 10; i++)
    {
        if( i == 5)
            continue;
        printf("%d\t",i);
    }
}

```

```
    }  
    return 0;  
}  
Output:  
0  1  2  3  4  6  7  8  9  10
```

Q5.5. Write short note on go to statement.**Answer:**

- The **goto** statement is used to jump from one line to another line in the program.
- To jump from one line to another line, the goto statement requires a **label**.
- Label is a name given to the instruction or line in the program.

Example Program for goto statement.

```
#include<stdio.h>  
int main()  
{  
    printf("We are at first printf statement!!!\n");  
    goto last ;  
    printf("We are at second printf statement!!!\n");  
    printf("We are at third printf statement!!!\n");  
    last: printf("We are at last printf statement!!!\n");  
    return 0;  
}
```

Output:

We are at first printf statement!!!

We are at last printf statement!!!

Q.5.6. Write the difference between break and continue.**Answer:**

Break	Continue
Break is used to terminate the execution of the loop.	Continue is not used to terminate the execution of loop.
It breaks the iteration.	It skips the iteration.
When this statement is executed, control will come out from the loop and executes the statement immediate after the loop.	When this statement is executed, it will not come out of the loop but jumps to the next iteration.
Break is used with loops and switch case.	Continue is only used in loops, it is not used in switch case.

6. Arrays**Q6.1. Define an Array. Explain how to declare and initialize an array.****Answer:****Array:**

- An array is a variable which can store more than one value of same datatype.
- The elements of array stored in continues memory locations.

Declaration of Array:

- when we want to create an array we must know the datatype of values to be stored in that array and also the number of values to be stored in that array.
- We use the following general syntax to create an array...

```
datatype arrayName [ size ] ;
```

- In the above syntax, the **datatype** specifies the type of values we store in that array and **size** specifies the maximum number of values that can be stored in that array.

Example

```
int a [3] ;
```

- Here, the compiler allocates 6 bytes of memory locations with a single name 'a' and tells the compiler to store three different integer values (each in 2 bytes of memory).
- For the above declaration, the memory is organized as follows...

**Initialization of Array:**

- Syntax for creating an array with size and initial values.

```
datatype arrayName [ size ] = {value1, value2, ...} ;
```

Example

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

Here, an array 'a' stores 5 values.

- Syntax for creating an array without size and with initial values

```
datatype arrayName [ ] = {value1, value2, ...} ;
```

Example

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

Here, an array 'a' stores 5 values.

Q6.2. Explain how to access elements of an array?**Answer:**

- An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array.

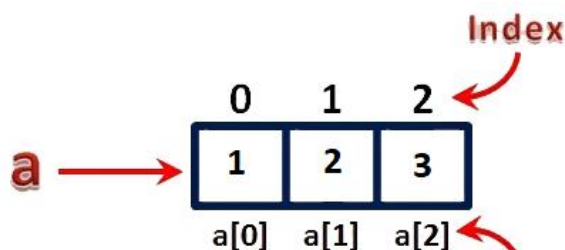
Syntax:

```
arrayName [ indexValue ] ;
```

Example:

```
int a[5] = {1, 2, 3}
```

For the above example the individual elements can be denoted as follows:



3. We can access the elements as follows:

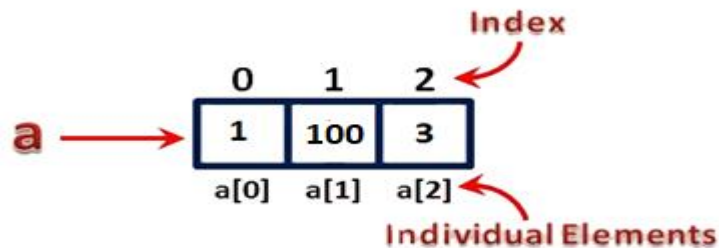
```
printf("%d", a[2]);
```

i. Output of the above statement is: 3

4. We can change the individual elements of array as follows:

```
a[1] = 100;
```

5. Then array elements are:



Q6.3. Explain the different types of arrays.

Answer:

- Arrays are classified into **two types**. They are as follows...
 - Single Dimensional Array / One Dimensional Array
 - Multi-Dimensional Array

1. Single Dimensional Array (One Dimensional Array):

- An array contains one subscript is known as One or Single Dimensional Array.
- Single dimensional arrays are used to store list of values of same datatype.
- Single dimensional arrays are also called as **one-dimensional arrays, Linear Arrays** or simply **1-D Arrays**.

Declaration of Single Dimensional Array:

- We use the following general syntax for declaring a single dimensional array.

```
datatype arrayName [ size ] ;
```

Example

```
int rollNumbers [60];
```

The above declaration of single dimensional array reserves 60 continuous memory locations of 2 bytes each with the name **rollNumbers** and tells the compiler to allow only integer values into those memory locations.

- We use the following general syntax for declaring and initializing a single dimensional array with size and initial values.

```
datatype arrayName [ size ] = {value1, value2, ...};
```

Example

```
int marks [6] = { 89, 90, 76, 78, 98, 86 };
```

The above declaration of single dimensional array reserves 6 contiguous memory locations of 2 bytes each with the name **marks** and initializes with value 89 in first memory location, 90 in second memory location, 76 in third memory location, 78 in fourth memory location, 98 in fifth memory location and 86 in sixth memory location.

- We can also use the following general syntax to initialize a single dimensional array without specifying size and with initial values.

```
datatype arrayName [ ] = {value1, value2, ...};
```

- The array must be initialized if it is created without specifying any size. In this case, the size of the array is decided based on the number of values initialized.

Example:

```
int marks [ ] = { 89, 90, 76, 78, 98, 86 };
```

```
char studentName [ ] = "GATESIT" ;
```

In the above example declaration, size of the array '**marks**' is **6** and the size of the array '**studentName**' is **16**. This is because in case of character array, compiler stores one extra character called **\0** (NULL) at the end.

Accessing Elements of Single Dimensional Array

- To access the elements of single dimensional array we use array name followed by index value of the element that to be accessed.
- The index value of single dimensional array starts with zero (0) for first element and incremented by one for each element.

- The index value in an array is also called as **subscript** or **indices**.
- We use the following general syntax to access individual elements of single dimensional array.

```
arrayName [ indexValue ]
```

Example

```
marks [2] = 99 ;
```

In the above statement, the third element of 'marks' array is assigned with value '99'.

2. Multi-Dimensional Array:

- If an array contains more than one subscript is known as multi-dimensional array.
- Multi-dimensional array can be of **two dimensional array** or **three dimensional array** or **four dimensional array** or more.
- Most popular and commonly used multi-dimensional array is **two dimensional array**.
- The 2-D arrays are used to store data in the form of table.
- We also use 2-D arrays to create mathematical **matrices**.

Declaration of Two Dimensional Array:

- We use the following general syntax for declaring a two dimensional array.

```
datatype arrayName [ rowSize ] [ columnSize ] ;
```

Example

```
int matrix_A [2][3] ;
```

The above declaration of two dimensional array reserves 6 continuous memory locations of 2 bytes each in the form of **2 rows** and **3 columns**.

Initialization of Two Dimensional Array:

- We use the following general syntax for declaring and initializing a two dimensional array with specific number of rows and columns with initial values.

```
datatype arrayName [rows][colms] = {{r1c1value, r1c2value, ...},{r2c1, r2c2,...}...} ;
```

Example

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

The above declaration of two-dimensional array reserves 6 contiguous memory locations of 2 bytes each in the form of 2 rows and 3 columns. And the first row is initialized with values 1, 2 & 3 and second row is initialized with values 4, 5 & 6.

- We can also initialize as follows.

Example

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

Accessing Individual Elements of Two Dimensional Array:

- To access elements of a two-dimensional array we use array name followed by row index value and column index value of the element that to be accessed.
- Here the row and column index values must be enclosed in separate square braces.
- We use the following general syntax to access the individual elements of a two-dimensional array...

```
arrayName [ rowIndex ] [ columnIndex ]
```

Example

```
matrix_A [0][1] = 10 ;
```

In the above statement, the element with row index 0 and column index 1 of **matrix_A** array is assigned with value **10**.

Q6.4. What are the applications of array?**Answer:**

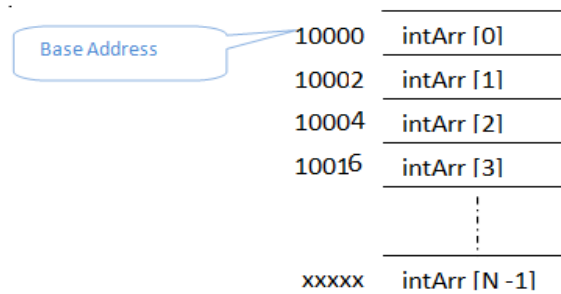
1. Arrays are used to Store List of values
2. Arrays are used to Perform Matrix Operations

3. Arrays are used to implement Search Algorithms
4. Arrays are used to implement Sorting Algorithms
5. Arrays are used to implement Data structures.
6. Arrays are also used to implement CPU Scheduling Algorithms

Q6.5. Explain how memory is allocated for 1-D arrays, and how can we address of i^{th} element.

Answer:

- Whenever an array is declared in the program, contiguous memory to its elements are allocated.
- Initial address of the array – address of the first element of the array is called base address of the array.
- Each element will occupy the memory space required to accommodate the values for its type, i.e.; depending on element's datatype, 1, 2 or 4 bytes of memory is allocated for each element.
- Below diagram shows how memory is allocated to an integer array of N elements. Its base address – address of its first element is 10000.



- Since it is an integer array, each of its elements will occupy 2 bytes of space. Hence the first element occupies memory from 10000 to 10001. The second element of the array occupies the immediate next memory address in the memory, i.e.; 10002 to 10003.
- In this way all the N elements of the array occupy the memory space.
- We can calculate i^{th} element address as follows:

$$\text{Address of Arr[} i \text{]} = \text{Base_Address} + i * \text{datatype_size}$$

Example:

- If an array 'P' is created with 50 elements of integer type. The base address is 5050, then calculate 28th element address.

$$\text{Address of P}[28] = \text{base_address} + 28 * \text{datatype_size}$$

$$\text{Address of P}[28] = 5000 + 28 * 2$$

$$\text{Address of P}[28] = 5030$$

Q6.6. Explain about memory allocation of 2D-Arrays.

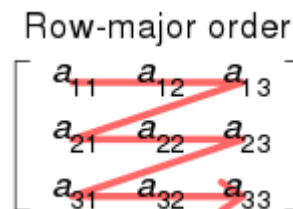
Answer:

- Whenever an array is declared in the program, contiguous memory to its elements are allocated, but the elements assigned to the memory location depend on the two different methods

1. Row Major Order
2. Column Major Order

1. Row Major Order:

- In 2D array, its elements are considered as rows and columns of a table.
- When we store the array elements in row major order, first we will store the elements of first row followed by second row and so on as shown in below figure.



- Hence in the memory we can find the elements of first row followed by second row and so on.

Example:

```
int intArr[3][3] = {{10,20,30}, {40,50,60}, {70,80,90}}
```

Matrix representation is:

$$\begin{bmatrix} 10 & 20 & 30 \\ 40 & 50 & 60 \\ 70 & 80 & 90 \end{bmatrix}$$

Row major order allocation as follows:

Base Address	10000	10	intArr [0][0]
	10002	20	intArr [0][1]
	10004	30	intArr [0][2]
	10006	40	intArr [1][0]
	10008	50	intArr [1][1]
	10010	60	intArr [1][2]
	10012	70	intArr [2][0]
	10014	80	intArr [2][1]
	10016	90	intArr [2][2]

- o We can find an element address of an array of size M*N as follows:

$$\text{Address of } a[i][j] = \text{Base_address} + \text{datatype_size} * (\text{total_rows} * i + j)$$

- o **Example:**

$$\text{Address of intarr}[1][2] = 10000 + 2 * (3 * 1 + 2)$$

$$\text{Address of intarr}[1][2] = 10000 + 2 * (3 + 2)$$

$$\text{Address of intarr}[1][2] = 10000 + 2 * 5$$

$$\text{Address of intarr}[1][2] = 10000 + 10$$

$$\text{Address of intarr}[1][2] = 10010$$

2. Column Major Order:

- o In this method all the first column elements are stored first, followed by second column elements and so on as shown in below figure.

Column-major order

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

Example:

```
int intArr[3][3] = {{10,20,30},{40,50,60},{70,80,90}}
```

Matrix representation is:

$$\begin{bmatrix} 10 & 20 & 30 \\ 40 & 50 & 60 \\ 70 & 80 & 90 \end{bmatrix}$$

Column major order allocation as follows:

Base Address	10000	10	intArr [0][0]
	10002	40	intArr [1][0]
	10004	70	intArr [2][0]
	10006	20	intArr [0][1]
	10008	50	intArr [1][1]
	10010	80	intArr [2][1]
	10012	30	intArr [0][2]
	10014	60	intArr [1][2]
	10016	90	intArr [2][2]

- o We can find an element address of an array of size M*N as follows:

$$\text{Address of } a[i][j] = \text{Base_address} + \text{datatype_size} * (\text{total_rows} * j + i)$$

- o **Example:**

$$\text{Address of intarr}[1][2] = 10000 + 2 * (3 * 2 + 1)$$

$$\text{Address of intarr}[1][2] = 10000 + 2 * (6 + 1)$$

$$\text{Address of intarr}[1][2] = 10000 + 2 * 7$$

$$\text{Address of intarr}[1][2] = 10000 + 14$$

$$\text{Address of intarr}[1][2] = 10014$$

2 Marks Questions

1. Define algorithm. List out its properties.
2. Define flowchart. List symbols of flowchart.
3. Define pseudocode.
4. What will be the output of the following program?

(a)

```
#include <stdio.h>
```

```
int main()
```

```

{
    int a=010;
    printf("\n a=%d",a);
    return 0;
}

```

Output: a = 8

(b)

```

#include<stdio.h>
int main()
{
    int a=010;
    printf("\n a=%o",a);
    return 0;
}

```

Output: a = 10

Explanation:

In (a), the integer constant 010 is taken to be octal as it is preceded by a zero (0). Here the variable 'a' is printed with %d specifier. The decimal equivalent of the octal value 10, which is 8, will be printed. Whereas in (b) the same variable is printed with %o format specifier, so 10 is printed on the screen.

(c)

```

#include <stdi.h>
int main()
{
    int a=010;
    printf("\n a=%x",a);
    return 0;
}

```

Output: a = 8

Explanation: In (c), the octal value 10 is printed with %x format specifier. That is hexadecimal equivalent of 10 which is 8 will be printed. Basics of C.

(d)

```
#include<stdio.h>
int main()
{
    int a=53;
    printf("\n a=%o",a);
    return 0;
}
```

Output: a = 65

Explanation: In (d), an integer constant 53 is stored in the variable 'a' but is printed with %o. The octal equivalent of 53, which is 65, will be printed.

(e)

```
#include<stio.h>
int main()
{
    int a=53;
    printf("\n a=%X",a);
    return 0;
}
```

Output: a = 35

Explanation: In (e), an integer constant 53 is stored in the variable 'a' but is printed with %X. The hexadecimal equivalent of 53, which is 35, will be printed.

5. Write short note on keywords.
6. Write short note on identifiers.
7. Define operator. List out different operators in C.
8. Define unary operator.
9. Define binary operator.
10. Define ternary operator.
11. What is the output of the following expression?

$$10+20+16-3*10/6-4$$

12. What is the output of the following expression?

$$8 \gg 3$$

13. What is the output of the following expression?

14. Define null statement.
15. Define expression. List out its types.
16. Write syntax and flow chart for if-else statement.
17. Define conditional statements.
18. Define looping statements.
19. Write syntax for do-while statement.
20. Write differences between while and do-while.
21. Define array.
22. Define one dimensional array.
23. How to declare and initialize one dimensional array.
24. Define two dimensional array.
25. How to declare and initialize two dimensional array.
26. What are the applications of array?
27. Write the difference between break and continue statements.
28. Define header file.

The header files primarily contain declarations relating to standard library functions and macros that are available with C.

Programming Questions

1. Data types and Operators

1. Finding the sum of three numbers
2. Exchange of two_numbers.
3. Write a program for the following exchanges
$$a \leftarrow b \leftarrow c \leftarrow d$$
4. Write a program to calculate area of a circle.
5. Write a program to calculate third angle of a triangle using two angles.
6. Write a program to perform left shift by 3 digits for a given number.
7. Write a program to perform right shift by 3 digits for a given number.
8. Write a program to print maximum of two numbers.
9. Write a program to perform arithmetic operations on given integers.

2. Conditional Statements:

1. Write a program to print given number is positive or not.
2. Write a program to print given number even or odd.
3. Write a program to find the roots of a Quadratic equation.
4. Write a program to print largest number among 3 numbers.
5. Write a program to print whether the given year is leap or not.

3. Iterative Statements:

1. Write a program to compute the factorial of a given number.
2. Write a program to check whether the number is prime or not.
3. Write a program to find the series of prime numbers in the given range.
4. Write a program to generate Fibonacci numbers in the given range.
5. Write a program to reverse the digits of a number.
6. Write a program to find the sum of the digits of a number.

7. Write a program to check for number palindrome.
8. Write a program to print GCD of two numbers.
9. Write a program to print LCM of two numbers.
10. Write a program to solve Towers of Hanoi problem.

4. Patterns and Formats:

1. Write a program to print the below format:

```

1     2     3     4     5
1     2     3     4     5
1     2     3     4     5
1     2     3     4     5

```

2. Write a program to print the below format:

```

1
1  2
1  2  3
1  2  3  4
1  2  3  4  5

```

3. Write a program to generate Pascal Triangle.
4. Write a program to print multiplication table of given number.

5. Series:

1. Write a program to evaluate the sum of the following series up to 'n' terms

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \dots$$

2. Write a program which computes the sum of the first n terms of the series

$$\text{Sum} = 1 - 3 + 5 - 7 + 9$$

3. Write a program which finds the sum of the infinite series

$$1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \dots$$

6. Arrays:

1. Write a program to find the sum of positive and negative numbers in a given_set of numbers.
2. Write a program to read two matrices and print their sum in the matrix form.
3. Write a program to read two matrices and print their product in the matrix form.
4. Write a program to read matrix and Find the sum of Diagonal Elements of a matrix.
5. Write a program to read matrix and Print Transpose of a matrix.
6. Write a program to read matrix and Print sum of even and odd numbers in a given matrix.
7. Write a program which finds the second maximum number among the given list of_numbers.
8. Construct a program which finds the kth smallest number among the given list of_numbers.
9. Write a program which reverses the elements of the array.
10. Write a program to find both the largest and smallest number in a list of integers.

Unit 2

1. Functions
2. Types of functions
3. Argument passing
4. Recursion
5. Pointers & storage allocation
6. Pointers to functions
7. Expressions involving pointers
8. Storage classes – auto, register, static, extern
9. Structures
10. Unions
11. Strings
12. String handling functions
13. Command line arguments

1. Functions

Q1.1. Define a function. How to define and call a function in C.

Answer:

- A function is a block of statements which perform specific task.
- Every function in C has the following:
 1. Function Definition
 2. Function Call
 3. Function Declaration (Function Prototype)

1. Function Definition:

The function definition is also known as the **body of the function**.

Syntax

```
Return_Type function_Name(parameters_List)
{
    Actual code...
}
```


In the above syntax,

Return_Type specifies the value return by a function

function_Name is a user-defined name used to identify the function uniquely

parameters_List is the data values that are sent to the function definition

Example:

```
void add(int a, int b)
{
    printf("Addition of a and b = %d", a+b);
}
```

In this example,

The function name is add, it is taking two integer variables as parameters and it is not returning any value.

2. Function Call:

The function call tells the compiler when to execute the function definition. When a function call is executed, the execution control jumps to the function definition where the actual code gets executed and returns to the same functions call once the execution completes.

The function call is performed inside the main function or any other function or inside the function itself.

Syntax

```
functionName(parameters);
```

Example:

```
add(15, 20)
```

3. Function Declaration (Function Prototype):

The function declaration tells the compiler about function name, the data type of the return value and parameters. The function declaration is also called a function prototype.

The function declaration is performed before the main function or inside the main function or any other function.

Syntax:

```
Return_Type functionName(parameters_List);
```

Example:

```
void add( int , int );
```

Q1.2. define parameter. What are its types.**Answer:**

- Parameters are the data values that are passed from calling function to called function.
- Parameters are also called as arguments.
- In C, there are two types of parameters and they are as follows...
 1. Actual Parameters
 2. Formal Parameters
- The actual parameters are the parameters that are specified in calling function.
- The formal parameters are the parameters that are declared at called function.
- When a function gets executed, the copy of actual parameter values are copied into formal parameters.

Example:

```

void main ( int argc, char* argv[ ] )
{
    double a = 3.14 ;
    update( a );
}

void update ( double a )
{
    a = a + 1 ;
}

```

Q1.3. Write the differences between actual parameters and formal parameters.**Answer:**

Actual Parameters	Formal Parameters
These are specified in function definition.	These are declared in function calling.
It may be variable or constant.	It must be variable.
Example: <pre>void main() { int a=2; sum(a, 40) }</pre> Here a, 40 are actual parameters.	Example: <pre>void sum(int x, int y) { } </pre> Here x and y are format parameters.

Q1.4. Define local and global variables.**Answer:****Local Variable:**

- Local variable is declared inside a function.
- Local variables are created when the function has started execution and is lost when the function terminates.
- If a value of local variable is updated, it will effect within that function only.

Global Variable:

- Global variable is declared outside the function.
- Global variable is created as execution starts and is lost when the program ends.
- If a value of global variable updated, it will effect on entire program.

Example:**Example Program**

```
#include<stdio.h>
int a = 10; // Global Variable
int main()
{
    int x = 20; // Local Variable
    printf("a = %d, x = %d", a, x);
    return 0;
}
```

Output:

```
a = 10, x = 20
```

Q1.5. Write short note on return statement.**Answer:**

return statement will return a value to the calling function and control is transfers from called to calling function.

Syntax:

```
return <value>;
```

Example:

```
return 10;
```

2. Types of Functions

Q2.1. What are the different types of functions? Explain.

Answer:

Bases on function definition, functions are divided into two types:

1. Pre-Defined Functions
2. User Defined Functions

1. Pre-Defined Functions:

The function whose definition is defined by the developers of language is called as system defined function.

The pre-defined functions are also called as Library Functions or Standard Functions or system Functions.

In C, all the system defined functions are defined inside the header files like stdio.h, conio.h, math.h, string.h, stdlib,h etc.,

For example, the funtions printf() and scanf() are defined in the header file called stdio.h.

2. User Defined Functions:

The function whose definition is defined by the user is called as user defined function.

Example Program

```
#include<stdio.h>
int addition(int,int) ; // function declaration
int main()
{
    int num1, num2, result ;
    printf("Enter any two integer numbers : ");
    scanf("%d%d", &num1, &num2);
    result = addition(num1, num2) ; // function call
    printf("SUM = %d", result);
    return 0;
}
int addition(int a, int b) // function definition
{
```

```

return a + b ;
}

```

Output:

Enter any two integer numbers : 23 54

SUM = 77

In the above example addition() is user defined function.

Q2.2. What are different types of functions based on parameters and return values?

Explain.

Answer:

Based on parameters and return types, functions are classified into 4 types:

1. Function without parameters and without return values
2. Function with parameters and without return values
3. Function without parameters and with return values
4. Function with parameters and with return values

1. Function without parameters and without return values:

- This type of functions doesn't contain parameters and return values.

Example Program

```

#include<stdio.h>
void addition() ; // function declaration
int main()
{
    addition() ; // function call
}
void addition() // function definition
{
    int num1, num2 ;
    printf("Enter any two integer numbers : ") ;
    scanf("%d%d", &num1, &num2);
    printf("Sum = %d", num1+num2 ) ;
}

```

Output:

Enter any two integer numbers : 20 30

Sum = 50

2. Function with parameters and without return values:

These type of functions having the parameters and doesn't have the return values.

Example Program

```
#include<stdio.h>
void addition(int, int) ; // function declaration
int main()
{
    int num1, num2 ;
    printf("Enter any two integer numbers : ") ;
    scanf("%d%d", &num1, &num2);
    addition(num1, num2) ; // function call
    return 0;
}
void addition(int a, int b) // function definition
{
    printf("Sum = %d", a+b) ;
}
```

Output:

```
Enter any two integer numbers : 20 30
Sum = 50
```

3. Function without parameters and with return values:

These type of functions doesn't have any parameters and contains return values.

Example Program

```
#include<stdio.h>
int addition() ; // function declaration
int main()
{
    int result ;
    result = addition() ; // function call
```

```

        printf("Sum = %d", result) ;
        return 0;
    }
int addition() // function definition
{
    int num1, num2 ;
    printf("Enter any two integer numbers : ") ;
    scanf("%d%d", &num1, &num2);
    return (num1+num2) ;
}

```

Output:

```

Enter any two integer numbers : 20 30
Sum = 50

```

4. Function with parameters and with return values:

These type of functions contain both parameters and return values.

Example Program

```

#include<stdio.h>
int main()
{
    int num1, num2, result ;
    int addition(int, int) ; // function declaration
    printf("Enter any two integer numbers : ") ;
    scanf("%d%d", &num1, &num2);
    result = addition(num1, num2) ; // function call
    printf("Sum = %d", result) ;
    return 0;
}
int addition(int a, int b) // function definition
{
    return (a+b) ;
}

```

Output:

Enter any two integer numbers : 20 30

Sum = 50

3. Argument passing**Q3.1. Explain in detail about parameter passing techniques.****(Or)****Explain call by value and call by reference with example.****Answer:**

There are two methods to pass parameters from calling function to called function and they are:

1. Call by Value
2. Call by Reference

1. Call by Value

In **call by value** parameter passing method, the copy of actual parameter values are copied to formal parameters and these formal parameters are used in called function.

The changes made on the formal parameters does not affect the values of actual parameters.

Example Program

```
#include<stdio.h>
void swap(int,int) ; // function declaration
int main()
{
    int num1, num2 ;
    printf("Enter two numbers: ");
    scanf("%d%d",&num1,&num2);
    printf("\nBefore swap: num1 = %d, num2 = %d", num1, num2) ;
    swap(num1, num2) ; // calling function
    printf("\nAfter swap: num1 = %d, num2 = %d", num1, num2);
    return 0;
}
```



```
void swap(int a, int b) // called function
{
    int temp ;
    temp = a ;
    a = b ;
    b = temp ;
}
```

Output:

Enter two numbers: 10 20

Before swap: num1 = 10, num2 = 20

After swap: num1 = 10, num2 = 20

2. Call by reference:

In **Call by Reference** parameter passing method, the memory address of the actual parameters is copied to formal parameters.

This address is used to access the memory locations of the actual parameters in called function. In this method, the formal parameters must be **pointer** variables.

Any changes made on the formal parameters effects the values of actual parameters.

Example Program

```
#include<stdio.h>
void swap(int*, int*) ; // function declaration
int main()
{
    int num1, num2 ;
    printf("Enter two numbers: ");
    scanf("%d%d",&num1,&num2);
    printf("\nBefore swap: num1 = %d, num2 = %d", num1, num2) ;
    swap(&num1, &num2) ; // calling function
    printf("\nAfter swap: num1 = %d, num2 = %d", num1, num2);
    return 0;
}
```

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

Output:

Enter two numbers: 10 20

Before swap: num1 = 10, num2 = 20

After swap: num1 = 20, num2 = 10

4. Recursion

Q4.1. Explain recursive function with an example.

Answer:

A function called by itself is called recursive function.

Syntax:

```
Return_type Function_name(Parameter_list)
{
    ....
    ....
    Function_name(Parameter_list);
}
```

Example Program--Factorial Using Recursive Function

```
#include<stdio.h>
int factorial( int );
int main()
{
    int fact, n ;
    printf("Enter any positive integer: ");
    scanf("%d", &n) ;
    fact = factorial( n ) ;
    printf("Factorial of %d is %d\n", n, fact) ;
    return 0;
}
int factorial( int n )
{
    int temp ;
    if( n == 0)
        return 1 ;
    else
        temp = n * factorial( n-1 ) ; // recursive function call
    return temp ;
}
```

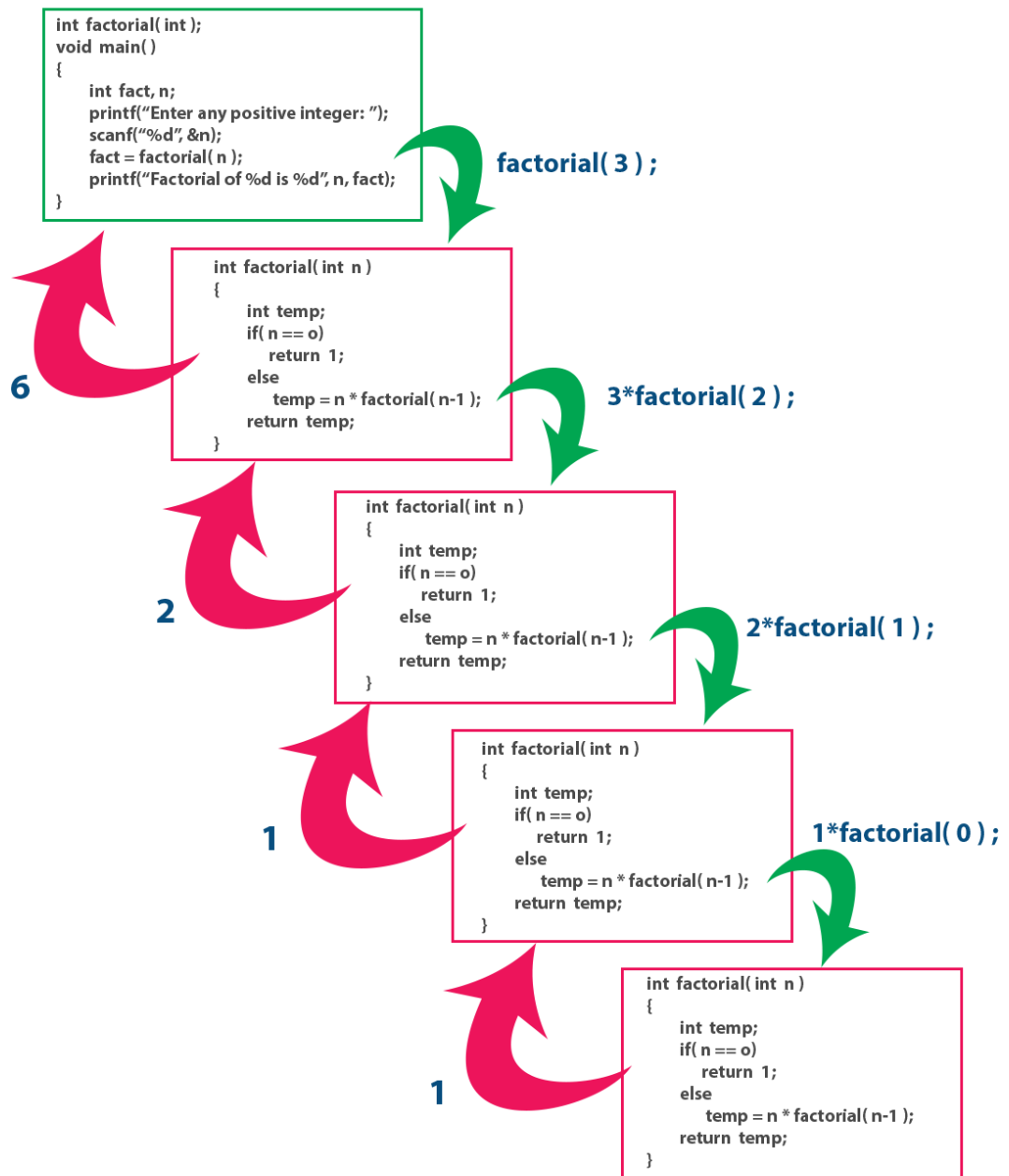
Output:

Enter any positive integer: 5

Factorial of 5 is 120

In the above example program, the **factorial()** function call is initiated from main() function with the value 3. Inside the factorial() function, the function calls factorial(2), factorial(1) and factorial(0) are called recursively.

The complete execution process of the above program is shown in the following figure:



5. Pointers & storage allocation

Q5.1. How to access address of a variable?

Answer:

- In c language, we use the **reference operator "&"** to access the address of variable.
- For example, to access the address of a variable "**marks**" we use "**&marks**".
- We use the following printf statement to display memory location address of variable "**marks**".

Example:

```
printf("Address : %u", &marks) ;
```

- In the above example statement **%u** is used to display address of **marks** variable.
- Address of any memory location is unsigned integer value.

Q5.2. Define a pointer. Explain how to declare and initialize a pointer.

Answer:

Pointer:

A pointer is a variable which holds memory address of another variable.

Declaring Pointer:

Declaration of pointer variable is similar to the creation of normal variable but the name is prefixed with * symbol.

Syntax:

```
datatype *pointerName ;
```

Example:

```
int *ptr ;
```

In the above example declaration, the variable "ptr" is a pointer variable that can be used to store any integer variable address.

Initializing a Pointer:

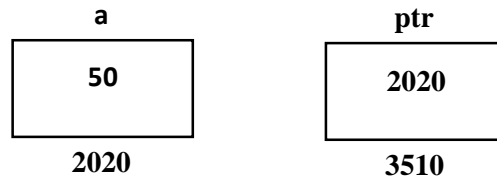
To assign address to a pointer variable we use assignment operator with the following syntax.

```
pointerVariableName = & variableName ;
```

Example:

```
int a = 50, *ptr ;
ptr = &a ;
```

Diagrammatic representation of above example is:



Accessing Variable Value Using Pointer

Pointer variables are used to store the address of other variables. We can use this address to access the value of the variable through its pointer.

We use the symbol "*" in front of pointer variable name to access the value of variable to which the pointer is pointing.

Syntax:

```
*pointerVariableName
```

Example Code

```
#include<stdio.h>
int main()
{
    int a = 10, *ptr ;
    ptr = &a ;
    printf("Address of variable a = %u\n", ptr) ;
    printf("Value of variable a = %d\n", *ptr) ;
    printf("Address of variable ptr = %u\n", &ptr) ;
    return 0;
}
```

Output:

```
Address of variable a = 6487580
Value of variable a = 10
Address of variable ptr = 6487568
```

Q5.3. How many bytes of memory allocated for pointer variable.**Answer:**

Every pointer variable is used to store the address of another variable. In computer memory address of any memory location is an **unsigned long integer** value.

In c programming language, unsigned long integer requires **4 bytes** of memory. So, irrespective of pointer datatype every pointer variable is allocated with 4 bytes of memory in 32-bit architecture.

Q5.4. Define pointer to pointer.**Answer:**

- A pointer variable to store the address of another pointer variable is called a pointer to pointer variable. Sometimes we also call it a double pointer.

Syntax:

```
datatype **pointerName ;
```

Example:

```
int a;
int *p1, **p2, ***p3;
p1 = &a;
p2 = &p1;
p3 = &p2
```

Note:

1. To store the address of normal variable we use single pointer variable
2. To store the address of single pointer variable we use double pointer variable
3. To store the address of double pointer variable we use triple pointer variable
4. Similarly, the same for remaining pointer variables also...

Q5.5. Define void pointer with example.**Answer:**

A void pointer is a pointer variable used to store the address of a variable of any datatype.

That means single void pointer can be used to store the address of integer variable, float variable, character variable, double variable or any structure variable.

We use the keyword "**void**" to create void pointer.

Syntax:

```
void *pointerName ;
```

Example:

```
int a;
float b;
void *p1, *p2;
p1 = &a;
p2 = &b;
```

6. Expressions involving pointers

Q6.1. What are the operations performed on pointers? Explain.

Answer:

Pointer variables are used to store the address of variables. Address of any variable is an unsigned integer value i.e., it is a numerical value. So we can perform following arithmetic operations on pointer values:

1. Addition
2. Subtraction
3. Increment
4. Decrement
5. Comparison

1. Addition:

The addition operation on pointer variables is calculated using the following formula:

$$\text{AddressAtPointer} + (\text{NumberToBeAdd} * \text{Datatype_size})$$

Example:

```
int a, *intPtr ;
float b, *floatPtr ;
double c, *doublePtr ;
```

```
intPtr = &a ; // Asume address of a is 1000
```



```
floatPtr = &b ; // Asume address of b is 2000
```

```
doublePtr = &c ; // Asume address of c is 3000
```

```
intPtr = intPtr + 3 ; // intPtr = 1000 + ( 3 * 2 ) → 1006
```

```
floatPtr = floatPtr + 2 ; // floatPtr = 2000 + ( 2 * 4 ) → 2008
```

```
doublePtr = doublePtr + 5 ; // doublePtr = 3000 + ( 5 * 6 ) → 3030
```

2. Subtraction:

The subtraction operation on pointer variables is calculated using the following formula:

$$\text{AddressAtPointer} - (\text{NumberToBeAdd} * \text{Datatype_Size})$$

Example:

```
int a, *intPtr ;
```

```
float b, *floatPtr ;
```

```
double c, *doublePtr ;
```

```
intPtr = &a ; // Asume address of a is 1000
```

```
floatPtr = &b ; // Asume address of b is 2000
```

```
doublePtr = &c ; // Asume address of c is 3000
```

```
intPtr = intPtr - 3 ; // intPtr = 1000 - ( 3 * 2 ) → 994
```

```
floatPtr = floatPtr - 2 ; // floatPtr = 2000 - ( 2 * 4 ) → 1992
```

```
doublePtr = doublePtr - 5 ; // doublePtr = 3000 - ( 5 * 6 ) → 2970
```

3. Increment:

The increment operation on pointer variable is calculated as follows:

$$\text{AddressAtPointer} + \text{Datatype_size}$$

Example:

```
int a, *intPtr ;
```

```
float b, *floatPtr ;
```

```
double c, *doublePtr ;
```

```
intPtr = &a ; // Asume address of a is 1000
```

```
floatPtr = &b ; // Asume address of b is 2000
doublePtr = &c ; // Asume address of c is 3000

intPtr++ ; // intPtr = 1000 + 2 → 1002
floatPtr++ ; // floatPtr = 2000 + 4 → 2004
doublePtr++ ; // doublePtr = 3000 + 6 → 3006
```

4. Decrement:

The decrement operation on pointer variable is calculated as follows:

$\text{AddressAtPointer} - \text{NumberOfBytesRequiresByDatatype}$
--

Example

```
int a, *intPtr ;
float b, *floatPtr ;
double c, *doublePtr ;

intPtr = &a ; // Asume address of a is 1000
floatPtr = &b ; // Asume address of b is 2000
doublePtr = &c ; // Asume address of c is 3000

intPtr-- ; // intPtr = 1000 - 2 → 998
floatPtr-- ; // floatPtr = 2000 - 4 → 1996
doublePtr-- ; // doublePtr = 3000 - 6 → 2994
```

5. Comparison:

The comparison operation is performing between the pointers of same datatype only. In c programming language, we can use all comparison operators (relational operators) with pointers.

Note:

- **We can't perform multiplication and division operations on pointers.**

7. Pointers to functions

Q7.1. Explain about pointers to arrays?

Answer:

- In c, when we declare an array the compiler allocates the required amount of memory and also creates a constant pointer with array name and stores the base address of that pointer in it.
- The address of the first element of an array is called as **base address** of that array.
- The array name itself acts as a pointer to the first element of that array.
- Consider the following example of array declaration...

```
int marks[6];
```

- For the above declaration, the compiler allocates 12 bytes of memory and the address of first memory location (i.e., marks[0]) is stored in a constant pointer called **marks**. That means in the above example, **marks** is a pointer to **marks[0]**.
- An array name is a **constant pointer**.

Note:

1. **marks** is same as **&marks[0]**
2. **marks + 1** is same as **&marks[1]**
3. **marks + 2** is same as **&marks[2]**
4. **marks + 3** is same as **&marks[3]**
5. **marks + 4** is same as **&marks[4]**
6. **marks + 5** is same as **&marks[5]**
7. ***marks** is same as **marks[0]**
8. ***(marks + 1)** is same as **marks[1]**
9. ***(marks + 2)** is same as **marks[2]**
10. ***(marks + 3)** is same as **marks[3]**
11. ***(marks + 4)** is same as **marks[4]**
12. ***(marks + 5)** is same as **marks[5]**
13. **marks[1][2]** is same as ***(*(marks + 1) + 2)**

Q7.2. Explain about pointers to functions (Call by reference).

Answer:

- We can pass address of a variable to the functions as a parameter.

- In **Call by Reference** parameter passing method, the memory address of the actual parameters is copied to formal parameters.
- This address is used to access the memory locations of the actual parameters in called function. In this method, the formal parameters must be **pointer** variables.
- **Any changes made on the formal parameters effects the values of actual parameters.**

Example Program

```
#include<stdio.h>
void swap(int*,int*) ; // function declaration
int main()
{
    int num1, num2 ;
    printf("Enter two numbers: ");
    scanf("%d%d",&num1,&num2);
    printf("\nBefore swap: num1 = %d, num2 = %d", num1, num2) ;
    swap(&num1, &num2) ; // calling function
    printf("\nAfter swap: num1 = %d, num2 = %d", num1, num2);
    return 0;
}
void swap(int *a, int *b) // called function
{
    int temp ;
    temp = *a ;
    *a = *b ;
    *b = temp ;
}
```

Output:

Enter two numbers: 10 20

Before swap: num1 = 10, num2 = 20

After swap: num1 = 20, num2 = 10

Q7.3. Define static memory allocation. What are its limitations.**Answer:**

- Allocation of memory during compile time is called static memory allocation.
- When we declare variables memory is allocated in space called stack. The memory allocated in the stack is fixed at the time of compilation and remains until the end of the program execution.

Limitations:

- When we create an array, we must specify the size at the time of the declaration itself and it cannot be changed during the program execution. This is a major problem when we do not know the number of values to be stored in an array.

Q7.4. Explain about dynamic memory allocation in C.**Answer:**

- Allocation of memory during the program execution is called dynamic memory allocation.
- We use pre-defined functions to allocate memory dynamically.
- There are FOUR pre-defined functions that are defined in the header file known as "stdlib.h".
- They are as follows:
 1. malloc()
 2. calloc()
 3. realloc()
 4. free()

1. malloc():

- malloc() is the pre-defined function used to allocate a memory block of specified number of bytes and returns void pointer.
- The void pointer can be casted to any datatype.
- If malloc() function unable to allocate memory due to any reason it returns NULL pointer.

Syntax:

```
void* malloc(size_in_bytes)
```

Example:

```
char *title;  
title = (char *) malloc(15);
```

2. calloc():

- calloc() is the pre-defined function used to allocate multiple memory blocks of the specified number of bytes and initializes them to ZERO.
- calloc() function returns void pointer.
- If calloc() function unable to allocate memory due to any reason it returns a NULL pointer.
- Generally, calloc() is used to allocate memory for array and structure.
- calloc() function takes two arguments and they are
 1. The number of blocks to be allocated
 2. Size of each block in bytes

Syntax:

```
void* calloc(number_of_blocks, size_of_each_block_in_bytes)
```

Example:

```
int *ptr;  
ptr = (int*)calloc(5, sizeof(int));
```

3. realloc():

- realloc() is the pre-defined function used to modify the size of memory blocks that were previously allocated using malloc() or calloc().
- realloc() function returns void pointer.
- If realloc() function unable to allocate memory due to any reason it returns NULL pointer.

Syntax

```
void* realloc(*pointer, new_size_of_each_block_in_bytes)
```

Example:

```
char *title;  
title = (char *) malloc(15);  
title = (char*) realloc(title, 30);
```

4. **free():**

- free() is the pre-defined function used to deallocate memory block that was previously allocated using malloc() or calloc().

Syntax

```
void free(*pointer)
```

Example:

```
char *title;  
title = (char *) malloc(15);  
free(title);
```

8. **Storage classes – auto, register, static, extern**

Q8.1. Define storage class. What are the storage classes in C. Explain with example?

Answer:

Storage classes are used to define storage location (whether RAM or Register), scope, lifetime and the default value of a variable.

In C language, there are FOUR storage classes and they are as follows:

1. auto storage class
2. static storage class
3. register storage class
4. extern storage class

1. auto storage class:

The default storage class of all local variables (variables declared inside block or function) is auto storage class.

Variable of auto storage class has the following properties:

Property	Description
Keyword	Auto
Storage	Computer Memory (RAM)
Default Value	Garbage Value
Scope	Local to the block in which the variable is defined
Life time	Till the control remains Within a block of function in which variable is defined.

Example Code

```
#include<stdio.h>
void fun()
{
    auto int a=1;
    printf("%d",a);
}
int main()
{
    fun();
    fun();
    fun();
}
```

Output:

```
1
1
1
```


2. static storage class:

The static storage class is used to create variables that hold value beyond its scope until the end of the program.

The static variable allows to initialize only once and can be modified any number of times.

Variable of static storage class has the following properties:

Property	Description
Keyword	Static
Storage	Computer Memory (RAM)
Default Value	Zero
Scope	Local to the block in which the variable is defined
Life time	The value of the persists between different function calls (i.e., Initialization is done only once)

Example Code

```
#include<stdio.h>
void fun()
{
    static int a;
    printf("%d\n",a);
    a++;
}
int main()
{
    fun();
    fun();
    fun();
}
```

Output:

```
0
1
2
```

3. register storage class:

The register storage class is used to specify the memory of the variable that has to be allocated in CPU Registers.

The register variables enable faster accessibility compared to other storage class variables.

As the number of registers inside the CPU is very less we can use very less number of register variables.

Variable of register storage class has the following properties:

Property	Description
Keyword	Register
Storage	CPU Register
Default Value	Garbage Value
Scope	Local to the block in which the variable is defined
Life time	Till the control remains within the block in which variable is defined

Example Code

```
#include<stdio.h>
void fun()
{
    register int a=1;
    printf("%d\n",a);
    a++;
}
int main()
{
    fun();
    fun();
    fun();
}
```

Output:

```
1
1
1
```

4. extern storage class:

The default storage class of all global variables (variables declared outside function) is external storage class.

Variable of external storage class has the following properties:

Property	Description
Keyword	extern
Storage	Computer Memory (RAM)
Default Value	Zero
Scope	Global to the program (i.e., Throughout the program)
Life time	As long as the program's execution does not comes to end

Example:

Extern.c	Main.c
<pre>#include<stdio.h> extern int a =10;</pre>	<pre>#include<stdio.h> #include "Extern.c" int main() { printf("%d",a); return 0; }</pre>
	<p>Output:</p> <p>10</p>

9. Structures

Q9.1. Define a structure. How to declare a structure in C?

Answer:

Structure:

Structure is a collection of different datatype elements which can be referred under a single name.

Creating Structure:

To create structure, we use the keyword called "**struct**".

Syntax:

```
struct structure_name
{
    // Structure members
    data_type member1;
    data_type member2, member3;
    ....
};
struct structure_name structure_variables;
```

Example:

```
struct Student
{
    // structure members
    char stud_name[30];
    int roll_number;
    float percentage;
};
// structure variables
struct Student s1,s2;
```

Accessing Structure Members:

We use structure variables to access structure members with dot(.) operator.

Syntax:

```
Structure_variable.structure_member
```

Example:

```
s1.stud_name  
s1.roll_number
```

Example:

```
#include<stdio.h>  
struct employee  
{  
    int id;  
    char name[50];  
}e1; //declaring e1 variable for structure  
int main( )  
{  
    //store first employee information  
    e1.id=101;  
    e1.name = "Ajay";  
    //printing first employee information  
    printf( "employee 1 id : %d\n", e1.id);  
    printf( "employee 1 name : %s\n", e1.name);  
    return 0;  
}
```

Output:

```
employee 1 id : 101  
employee 1 name : Ajay
```

Q9.2. How much memory is allocated for a structure?**Answer:**

The memory does not allocate on defining a structure. The memory is allocated when we create the variable of a particular structure.

The size of memory allocated is equal to the sum of memory required by individual members of that structure.

Example:

```

struct Student{
    char stud_name[30];———— 30 bytes
    int roll_number;———— 02 bytes
    float percentage;———— 04 bytes
};
sum = 36 bytes

```

Here the variable of **Student** structure is allocated with 36 bytes of memory.

Q9.3. Explain in detail about structure with arrays with example.**Answer:**

An array of structures in C can be defined as the collection of multiple structures variables where each variable contains information about different entities.

The array of structures in C are used to store information about multiple entities of different data types.

The array of structures is also known as the collection of structures.

Example:

```

#include<stdio.h>
struct student
{
    int rollno;
    char name[10];
};
int main()
{
    int i;
    struct student st[5];
    printf("Enter Records of 5 students");
    for(i=0;i<5;i++)
    {

```

```
        printf("\nEnter Rollno:");
        scanf("%d",&st[i].rollno);
        printf("\nEnter Name:");
        scanf("%s",&st[i].name);
    }
    printf("\nStudent Information List:");
    for(i=0;i<5;i++)
    {
        printf("\nRollno:%d, Name:%s",st[i].rollno,st[i].name);
    }
    return 0;
}
```

Output:

Enter Records of 5 students

Enter Rollno:1

Enter Name:Sonoo

Enter Rollno:2

Enter Name:Ratan

Enter Rollno:3

Enter Name:Vimal

Enter Rollno:4

Enter Name:James

Enter Rollno:5

Enter Name:Sarfraz

Student Information List:

Rollno:1, Name:Sonoo

Rollno:2, Name:Ratan

Rollno:3, Name:Vimal

Rollno:4, Name:James

Rollno:5, Name:Sarfraz

Q9.4. Explain in detail about structure with functions with example.**Answer:**

We can pass a structure a variable to a function and also a function returns structure variable.

Example:

```
#include <stdio.h>
// function prototype
void display(struct student s);
struct student
{
    char name[50];
    int age;
};

int main()
{
    struct student s1;
    printf("Enter name: ");
    scanf("%s", s1.name);
    printf("Enter age: ");
    scanf("%d", &s1.age);
    display(s1); // passing struct as an argument
    return 0;
}

void display(struct student s)
{
    printf("\nDisplaying information\n");
    printf("Name: %s", s.name);
    printf("\nAge: %d", s.age);
}
```


Output

Enter name: Bond

Enter age: 13

Displaying information

Name: Bond

Age: 13

Q9.5. Explain in detail about structure with pointer with example.**Answer:**

We can create pointers to the structure variable.

```
struct name
```

```
{
```

```
    member1;
```

```
    member2;
```

```
    .
```

```
    .
```

```
};
```

```
struct name *ptr, Harry;
```

Here, ptr is a pointer to struct.

We can Access pointers to structure we use -> (membership) operator.

Example:

```
#include <stdio.h>
```

```
struct person
```

```
{
```

```
    int age;
```

```
    float weight;
```

```
};
```

```
int main()
```

```
{
```

```

struct person *personPtr, person1;

personPtr = &person1;

printf("Enter age: ");

scanf("%d", &personPtr->age);

printf("Enter weight: ");

scanf("%f", &personPtr->weight);

printf("Displaying:\n");

printf("Age: %d\n", personPtr->age);

printf("weight: %f", personPtr->weight);

return 0;

}

```

Output:

Enter age: 18

Enter weight: 52

Displaying:

Age: 18

weight: 52.000000

Q9.6. Write short note on structure with in structure.**Answer:**

C provides us the feature of nesting one structure within another structure by using which, complex data types are created.

For example, we may need to store the address of an entity employee in a structure. The attribute address may also have the subparts as street number, city, state, and pin code. Hence, to store the address of the employee, we need to store the address of the employee into a separate structure and nest the structure address into the structure employee.

Consider the following program.

Example

```

#include<stdio.h>

struct address

{

char city[20];

```

```
int pin;
char phone[14];
};
struct employee
{
char name[20];
struct address add;
};
int main ()
{
struct employee emp;
printf("Enter employee information?\n");
scanf("%s %s %d %s",emp.name,emp.add.city, &emp.add.pin, emp.add.phone);
printf("Printing the employee information....\n");
printf("name: %s\nCity: %s\nPincode: %d\nPhone:
%s",emp.name,emp.add.city,emp.add.pin,emp.add.phone);
return 0;
}
```

Output

Enter employee information?

Arun

Delhi

110001

1234567890

Printing the employee information....

name: Arun

City: Delhi

Pincode: 110001

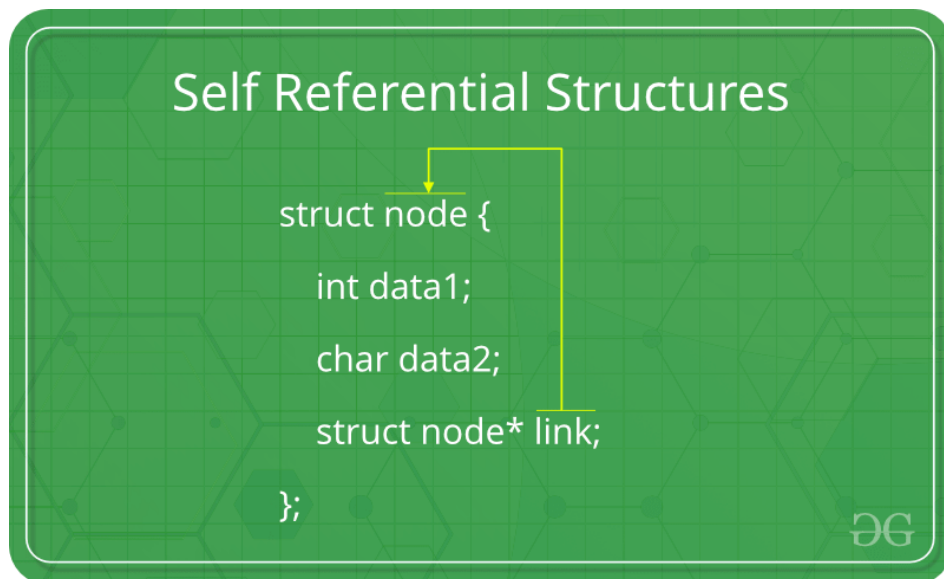
Phone: 1234567890

Q9.7. Define self-referential structure.

Answer:

Self-Referential structures are those structures that have one or more pointers which point to the same type of structure, as their member.

Example:



10. Unions

Q 10.1. Define union. Explain creation of union and accessing union members.

Answer:

Union:

Structure is a collection of different datatype elements which can be referred under a single name.

Creating Union:

To create union, we use the keyword called "**union**".

Syntax:

```
union union_name
{
    // Union members
    data_type member1;
    data_type member2, member3;
    ....
};
union union_name union_variables;
```

Example:

```
union Student
{
    // union members
    char stud_name[30];
    int roll_number;
    float percentage;
};
// union variables
union Student s1,s2;
```

Accessing Union Members:

We use union variables to access structure members with dot(.) operator.

Syntax:

```
Union_variable . Union_member
```

Example:

```
s1.stud_name
s1.roll_number
s1.percentage
```

Q10.2 How much memory is allocated for a structure?**Answer:**

The memory is allocated when we create the variable of a particular union. The size of memory allocated is equal to the maximum memory required by an individual member among all members of that union.

Example:

```

union Student{
    char stud_name[30]; ————— 30 bytes
    int roll_number; ————— 02 bytes
    float percentage; ————— 04 bytes
};
                                max = 30 bytes

```

Q10.3. Write the differences between structure and union.

	STRUCTURE	UNION
Keyword	The keyword struct is used to define a structure	The keyword union is used to define a union.
Size	When a variable is associated with a structure, the compiler allocates the memory for each member. The size of structure is greater than or equal to the sum of sizes of its members.	when a variable is associated with a union, the compiler allocates the memory by considering the size of the largest memory. So, size of union is equal to the size of largest member.
Memory	Each member within a structure is assigned unique storage area of location.	Memory allocated is shared by individual members of union.
Value Altering	Altering the value of a member will not affect other members of the structure.	Altering the value of any of the member will alter other member values.
Accessing members	Individual member can be accessed at a time.	Only one member can be accessed at a time.
Initialization of Members	Several members of a structure can initialize at once.	Only the first member of a union can be initialized.

11. Strings**Q11.1. Define a string. Explain how to create and initialize strings in C.****Answer:****String:**

String is a set of characters enclosed in double quotation marks.

In C programming, the string is a character array of single dimension.

Creating a String:

Syntax:

```
char string_name[size];
```

Example:

```
char str1[30];
```

Initialization:

Syntax:

```
string_name[size] = String
```

Example:

1. char str1[30] = "Hello"
2. char str2[20];
str = "Hello Hai"

Note:

'\0' represents end of the string.

Q11.2. Explain about gets and puts functions.

Answer:

gets() function is used to read a string from standard input device and puts is used to display a string on standard output device.

gets() and puts() functions are defined in stdio.h header file.

Example:

Example:

```
#include<stdio.h>
int main()
{
    char s[20];
    printf("Enter a String: ");
    gets(s);
    printf("Your String is: ");
```

Prepared by Sushma.,Asst.Professor.,Dept of CSE.,PVKKIT

```

    puts(s);
    return 0;
}

```

Output:

Enter a String: hello

Your String is: hello

12. String handling functions**Q.13. Explain about string handling functions.****Answer:**

- C programming language provides a set of pre-defined functions called string handling functions to work with string values.
- The string handling functions are defined in a header file called string.h.
- Whenever we want to use any string handling function we must include the header file called string.h.
- The following table provides most commonly used string handling function and their use:

Function	Syntax (or) Example	Description
strcpy()	strcpy(string1, string2)	Copies string2 value into string1
strlen()	strlen(string1)	returns total number of characters in string1
strcat()	strcat(string1,string2)	Appends string2 to string1
strcmp()	strcmp(string1, string2)	Returns 0 if string1 and string2 are the same; less than 0 if string1<string2; greater than 0 if string1>string2
strlwr()	strlwr(string1)	Converts all the characters of string1 to lower case.

Function	Syntax (or) Example	Description
strupr()	strupr(string1)	Converts all the characters of string1 to upper case.
strchr()	strchr(string1, 'b')	Returns a pointer to the first occurrence of character 'b' in string1
strrchr()	'strrchr(string1, 'b')	Returns a pointer to the last occurrence of character 'b' in string1
strstr()	strstr(string1, string2)	Returns a pointer to the first occurrence of string2 in string1
strset()	strset(string1, 'B')	Sets all the characters of string1 to given character 'B'.
strrev()	strrev(string1)	It reverses the value of string1

13. Command line arguments

Q13.1. What are the command line arguments? Explain with an example.

Answer:

- Command line arguments are the parameters passing to main() method from the command line.
- When command line arguments are passed main() method receives them with the help of two formal parameters and they are,
 1. int argc
 2. char *argv[]
- **int argc** - It is an integer argument used to store the count of command line arguments are passed from the command line.
- **char *argv[]** - It is a character pointer array used to store the actual values of command line arguments are passed from the command line.

Example Program to illustrate command line arguments in C.**Program name: main.c**

```
#include<stdio.h>
int main(int argc, char *argv[])
{
    int i;
    if(argc == 1)
        printf("Please provide command line arguments!!!");
    else
    {
        printf("Total number of arguments are - %d and they are\n\n", argc);
        for(i=0; i<argc ; i++)
        {
            printf("%d -- %s \n", i+1, argv[i]);
        }
    }
    return 0;
}
```

Output:

D:\C Programs>main.exe hello welcome to GATESIT

Total number of arguments are - 5 and they are

1 -- main.exe

2 -- hello

3 -- welcome

4 -- to

5 -- GATESIT

Q13.2. Write short note on enumeration in C with an example.**Answer:**

- Enumeration is the process of creating user defined datatype by assigning names to integral constants
- We use the keyword **enum** to create enumerated datatype.

Syntax:

```
enum enum_name{name1, name2, name3, ... }
```

- In the above syntax, integral constant '0' is assigned to name1, integral constant '1' is assigned to name2 and so on.
- We can also assign our own integral constants as follows:

```
enum enum_name {name1 = 10, name2 = 30, name3 = 15, ... }
```

- In the above syntax, integral constant '10' is assigned to name1, integral constant '30' is assigned to name2 and so on.

Example Program for enum with default values

```
#include<stdio.h>
enum day { Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday} ;
void main()
{
    printf("\nSunday = %d ", Sunday) ;
    printf("\nMonday = %d ", Monday) ;
    printf("\nTuesday = %d ", Tuesday) ;
    printf("\nWednesday = %d ", Wednesday) ;
    printf("\nThursday = %d ", Thursday) ;
    printf("\nFriday = %d ", Friday) ;
    printf("\nsaturday = %d ", Saturday) ;
    return 0;
}
```

Output:

Sunday = 0

Monday = 1
Tuesday = 2
Wednesday = 3
Thursday = 4
Friday = 5
saturday = 6

Q13.3. Write a short note on typedef.**Answer:**

- In C programming language, typedef is a keyword used to create alias name for the existing datatypes.

Syntax:

```
typedef <existing-datatype> <alias-name>
```

Example:**Example Program to illustrate typedef in C.**

```
#include<stdio.h>
typedef int Number;
int main()
{
    Number a,b,c; // Here a,b,&c are integer type of variables.
    printf("Enter any two integer numbers: ");
    scanf("%d%d", &a,&b);
    c = a + b;
    printf("Sum = %d", c);
    return 0;
}
```

Output:

```
Enter any two integer numbers: 5 3
Sum = 8
```

Q13.4. Write short note on bit fields.**Answer:**

- In C, we can specify size (in bits) of structure and union members.
- The idea is to use memory efficiently when we know that the value of a field or group of fields will never exceed a limit or is within a small range.

Example:

```
#include <stdio.h>

struct date
{
    int d : 5;
    int m : 4;
    int y;
};

int main()
{
    printf("Size of date is %lu bytes\n", sizeof(struct date));
    return 0;
}
```

Output:

Size of date is 4 bytes

2 Marks Questions

1. Define a function. Write the syntax to define a function.
2. Define recursion.
3. Define a pointer.
4. Write the differences between call by value and call by reference.
5. Differentiate formal and actual arguments.
6. Write short note on return statement.
7. Write a note on static and extern storage classes.
8. Define a storage class.
9. Write a note on register and auto storage classes.
10. Define structure.
11. Define union.
12. Write the differences between structure and union.
13. Define a string.
14. What are the command line arguments?
15. Write short note on enum.
16. Write a short note on typedef.
17. Write a short note on bitfields.

Programs

1. Develop a C program for the following using recursive and non-recursive programs.
 - a. To find the factorial number
 - b. To find GCD of two numbers
 - c. To find Fibonacci series
 - d. To find solve Towers of Hanoi Problem
2. Write a C program that performs various arithmetic operations on pointers.
3. Write a C program to demonstrate call by value and call by reference.
4. Write a C program that uses functions to perform the following operations:
 - a. To insert a sub-string in to a given main string from a given position.
 - b. To delete n characters from a given position in a given string.
5. Write a C program that displays the position or index in the string S where the string T begins, or – 1 if S doesn't contain T.
6. Write a C program to count the lines, words and characters in a given text.
7. Write a C program that uses functions to perform the following operations:
 - i) Reading a complex number
 - ii) Writing a complex number
 - iii) Addition of two complex numbers
 - iv) Multiplication of two complex numbers

Unit 3

Data Structures

1. Overview of data structures
2. Stacks
 - A. Representation of a stack
 - B. Stack related terms
 - C. Operations on a stack
 - D. Implementation of a stack
 - E. Evaluation of arithmetic expressions
 - F. Infix, prefix, and postfix notations
 - G. Evaluation of postfix expression
 - H. Conversion of expression from infix to postfix
 - I. Recursion
3. Queues
 - A. Various positions of queue
 - B. Representation of queue
 - C. Insertion, deletion, searching operations

1. Overview of Data Structures

Q1.1. Define data structure?

Answer:

Data structure is a method of organizing a large amount of data more efficiently so that any operation on that data becomes easy.

Q1.2. List the types of data structures.

Answer:

Data structures are divided into two types.

1. Linear Data Structures
2. Non - Linear Data Structures

1. Linear Data Structures

If a data structure organizes the data in sequential order, then that data structure is called a Linear Data Structure.

Example

1. Arrays
2. List (Linked List)
3. Stack
4. Queue

2. Non - Linear Data Structures

If a data structure organizes the data in random order, then that data structure is called as Non-Linear Data Structure.

Example

1. Tree
2. Graph
3. Dictionaries
4. Heaps
5. Tries, Etc.,

2. Stacks

Q2.1. Define stack?

Answer:

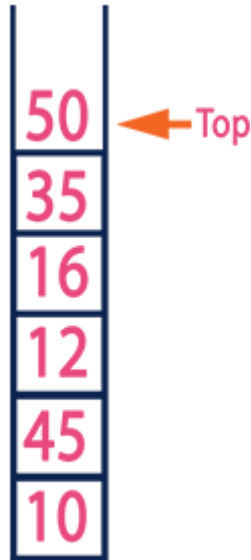
Stack:

Stack is a linear data structure in which the operations are performed based on LIFO (Last In First Out) principle.

In a stack, adding and removing of elements are performed at a single end which is known as "**top**".

Example:

If we want to create a stack by inserting 10,45,12,16,35 and 50. Then 10 becomes the bottom-most element and 50 is the topmost element. The last inserted element 50 is at Top of the stack as shown in the image below...



Q2.2. In how many ways a stack can be implemented?

Answer:

Stack data structure can be implemented in two ways. They are as follows...

1. Using Array
2. Using Linked List

When a stack is implemented using an array, that stack can organize an only limited number of elements. When a stack is implemented using a linked list, that stack can organize an unlimited number of elements.

Q2.3. Explain operations performed on stack.

Answer:

The following operations are performed on the stack...

1. Push (To insert an element on to the stack)
2. Pop (To delete an element from the stack)
3. Display (To display elements of the stack)

1. Push:

Push operation, insert an element into stack. In a stack, the new element is always inserted at **top** position.

Algorithm:

```

Algorithm: void push(int value)
Step-1: Start
Step-2: if top == SIZE-1 then
            Display "Stack is Full"
Step-3: else
            top++;
            stack[top] = value;
            Display "Insertion success"
Step-4: Stop

```

2. Pop:

Push operation, delete an element from stack. In a stack, an element is always deleted from **top** position.

Algorithm:

```

Algorithm: void pop()
Step-1: Start
Step-2: if top == -1 then
            Display "Stack is Empty"
Step-3: else
            Display "Deleted: stack[top]"
            top--;
Step-4: Stop

```

3. Display:

Display Operation, displays all elements in a stack.

Algorithm:

```

Algorithm: void display()
Step-1: Start
Step-2: if top == -1 then
            Display "Stack is Empty"
Step-3: else
            Display "Stack elements are:"
            for(i=top; i>=0; i--)
                Display stack[i]
Step-4: Stop

```

Q2.4. Write a C program to implement stack operations using array.**Answer:**

```
#include<stdio.h>
#define SIZE 20
int stack[SIZE], top = -1;

void push(int value)
{
    if(top == SIZE-1)
        printf("\nStack is Full, Insertion is not possible");
    else
    {
        top++;
        stack[top] = value;
        printf("\nInsertion success");
    }
}

void pop()
{
    if(top == -1)
        printf("\nStack is Empty, Deletion is not possible");
    else
    {
        printf("\nDeleted : %d", stack[top]);
        top--;
    }
}

void display()
{
    if(top == -1)
        printf("\nStack is Empty!!!");
    else
    {
        int i;
        printf("\nStack elements are:\n");
        for(i=top; i>=0; i--)
            printf("%d\n",stack[i]);
    }
}
```

```

int main()
{
    int value, choice;
    do
    {
        printf("\n\n***** MENU *****\n");
        printf("1. Push\n2. Pop\n3. Display\n4. Exit");
        printf("\nEnter your choice: ");
        scanf("%d",&choice);
        switch(choice)
        {
            case 1:
                printf("Enter the value to be insert: ");
                scanf("%d",&value);
                push(value);
                break;
            case 2:
                pop();
                break;
            case 3:
                display();
                break;
            case 4:
                printf("Exting...");
                break;
            default:
                printf("\nWrong selection!!! Try again!!!");
        }
    }while(choice!=4);
    return 0;
}

```

Output:

***** MENU *****

1. Push
2. Pop
3. Display
4. Exit

Enter your choice: 1

Enter the value to be insert: 10

Insertion success

***** MENU *****

1. Push
2. Pop
3. Display
4. Exit

Enter your choice: 1

Enter the value to be insert: 20

Insertion success

***** MENU *****

1. Push
2. Pop
3. Display
4. Exit

Enter your choice: 1

Enter the value to be insert: 30

Insertion success

***** MENU *****

1. Push
2. Pop
3. Display
4. Exit

Enter your choice: 3

Stack elements are:

30

20

10

***** MENU *****

1. Push
2. Pop
3. Display
4. Exit

Enter your choice: 2

Deleted : 30

***** MENU *****

1. Push
2. Pop
3. Display
4. Exit

Enter your choice: 2

Deleted : 20

***** MENU *****

1. Push
2. Pop

3. Display

4. Exit

Enter your choice: 3

Stack elements are:

10

***** MENU *****

1. Push
2. Pop
3. Display
4. Exit

Enter your choice: 2

Deleted : 10

***** MENU *****

1. Push
2. Pop
3. Display
4. Exit

Enter your choice: 2

Stack is Empty, Deletion is not possible

***** MENU *****

1. Push
2. Pop
3. Display
4. Exit

Enter your choice: 3

Stack is Empty!!!

***** MENU *****

1. Push
2. Pop
3. Display
4. Exit

Enter your choice: 4

Exiting...

Q2.5. What are the applications of stack?**Answer:**

1. Stacks can be used for expression evaluation.
2. Stacks can be used to check parenthesis matching in an expression.
3. Stacks can be used for Conversion from one form of expression to another.
4. Stacks can be used for Memory Management.
5. Stack data structures are used in backtracking problems.

Q2.6. Define an expression. Explain different types of expression?**Answer:****Expression:**

An expression is a collection of operators and operands that represents a specific value.

Example:

$$C = A + B$$

Types of Expressions:

There are 3 types of expressions.

1. Infix Expression
2. Postfix Expression
3. Prefix Expression

1. Infix Expression:

In infix expression, an operator is used in between operands.

Example:

$$A + B$$

2. Postfix Expression:

In postfix expression, an operator is used after operands.

Example:

$$A B +$$

3. Prefix Expression:

In prefix expression, an operator is used before operands.

Example:

$$+ A B$$

Q2.7. Explain the process of converting infix to postfix expression?**Answer:****Infix to Postfix Expression Conversion:****Algorithm:**

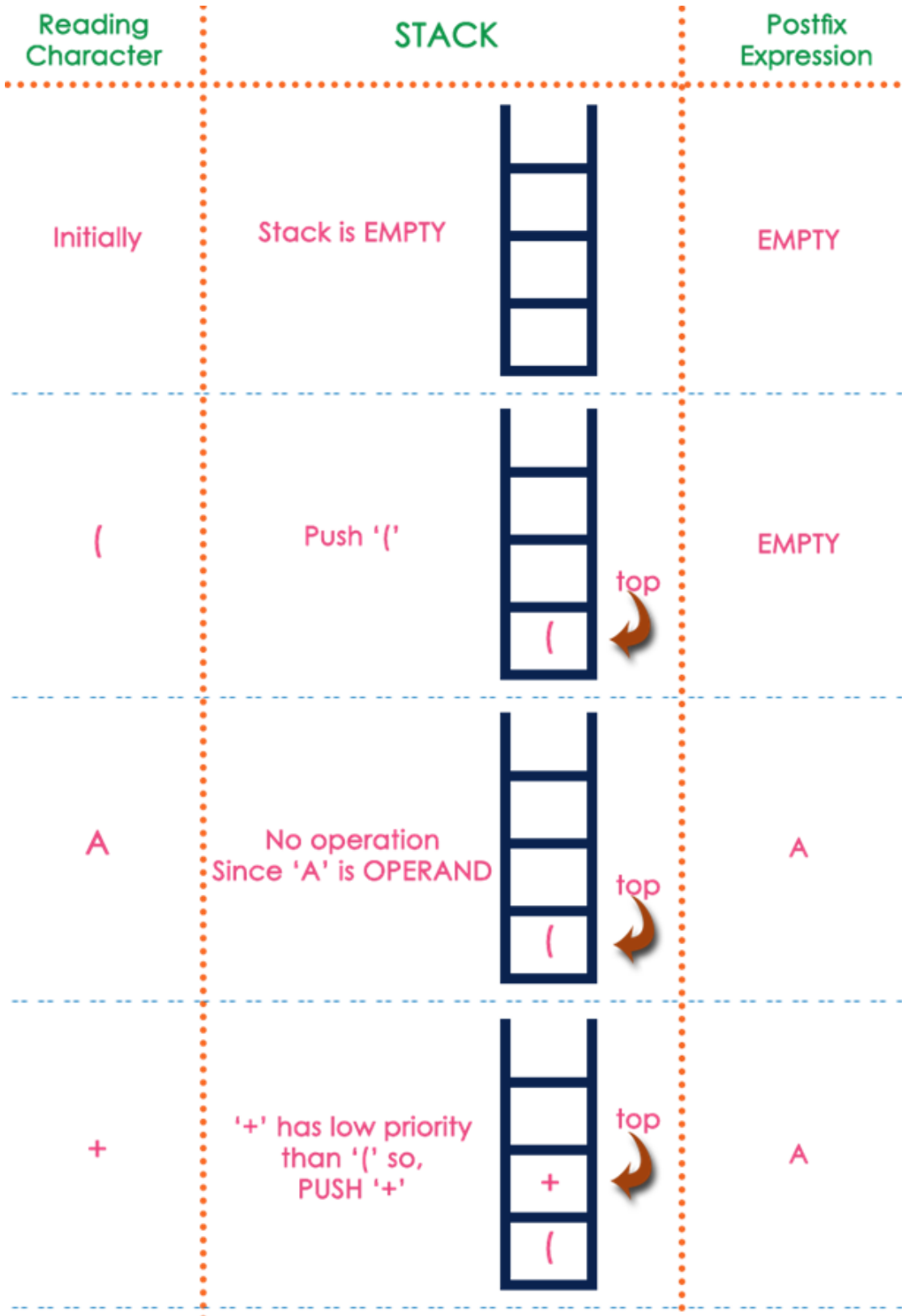
1. Scan the infix expression from left to right.
2. If the scanned character is an operand, output it.
3. Else,
 - A. If the precedence of the scanned operator is greater than the precedence of the operator in the stack (or the stack is empty or the stack contains a '('), push it.
 - B. Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator. After doing that Push the scanned operator to the stack. (If you encounter parenthesis while popping then stop there and push the scanned operator in the stack.)
4. If the scanned character is an '(', push it to the stack.
5. If the scanned character is an ')', pop the stack and output it until a '(' is encountered, and discard both the parenthesis.
6. Repeat steps 2-6 until infix expression is scanned.
7. Print the output
8. Pop and output from the stack until it is not empty.

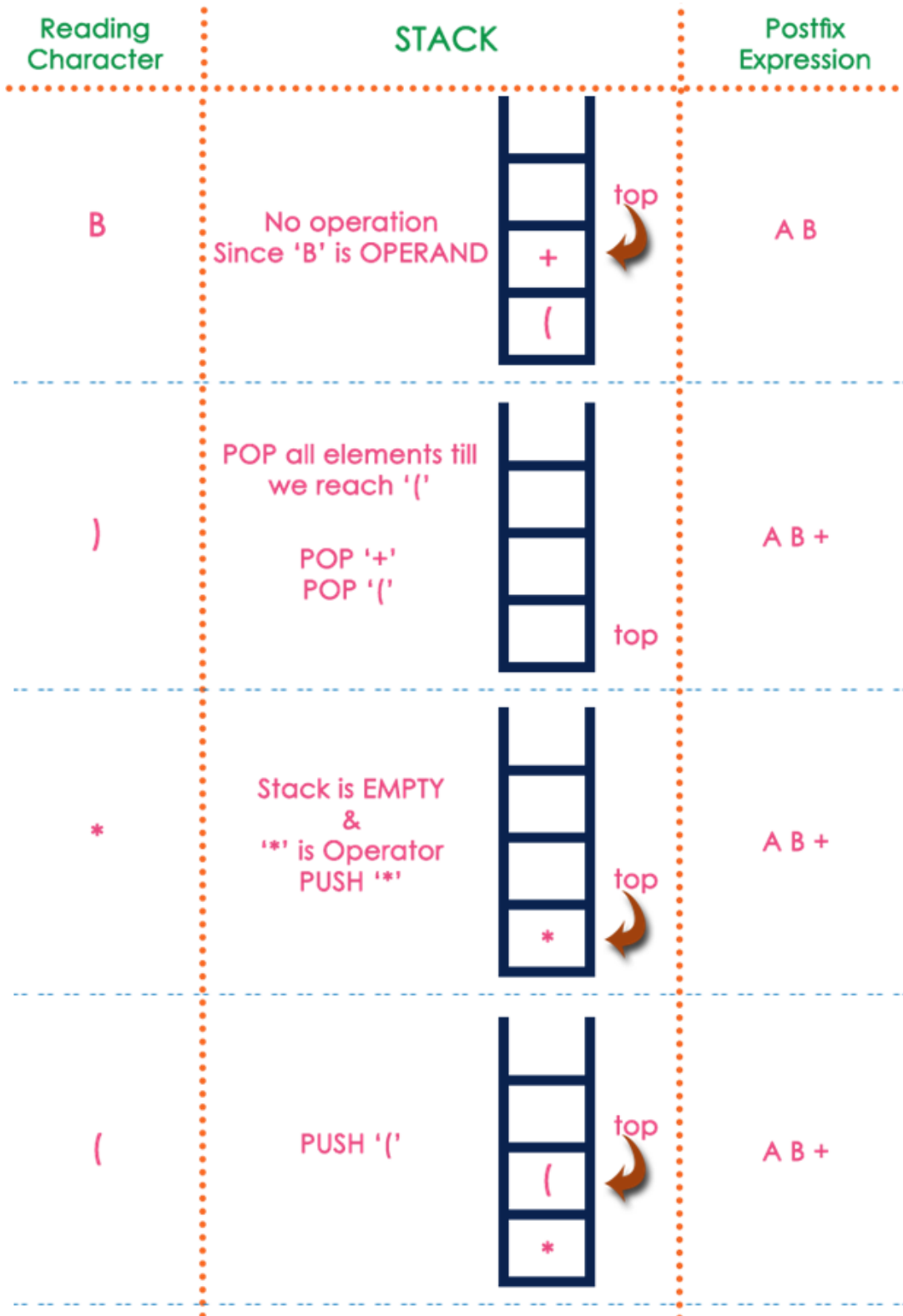
Example

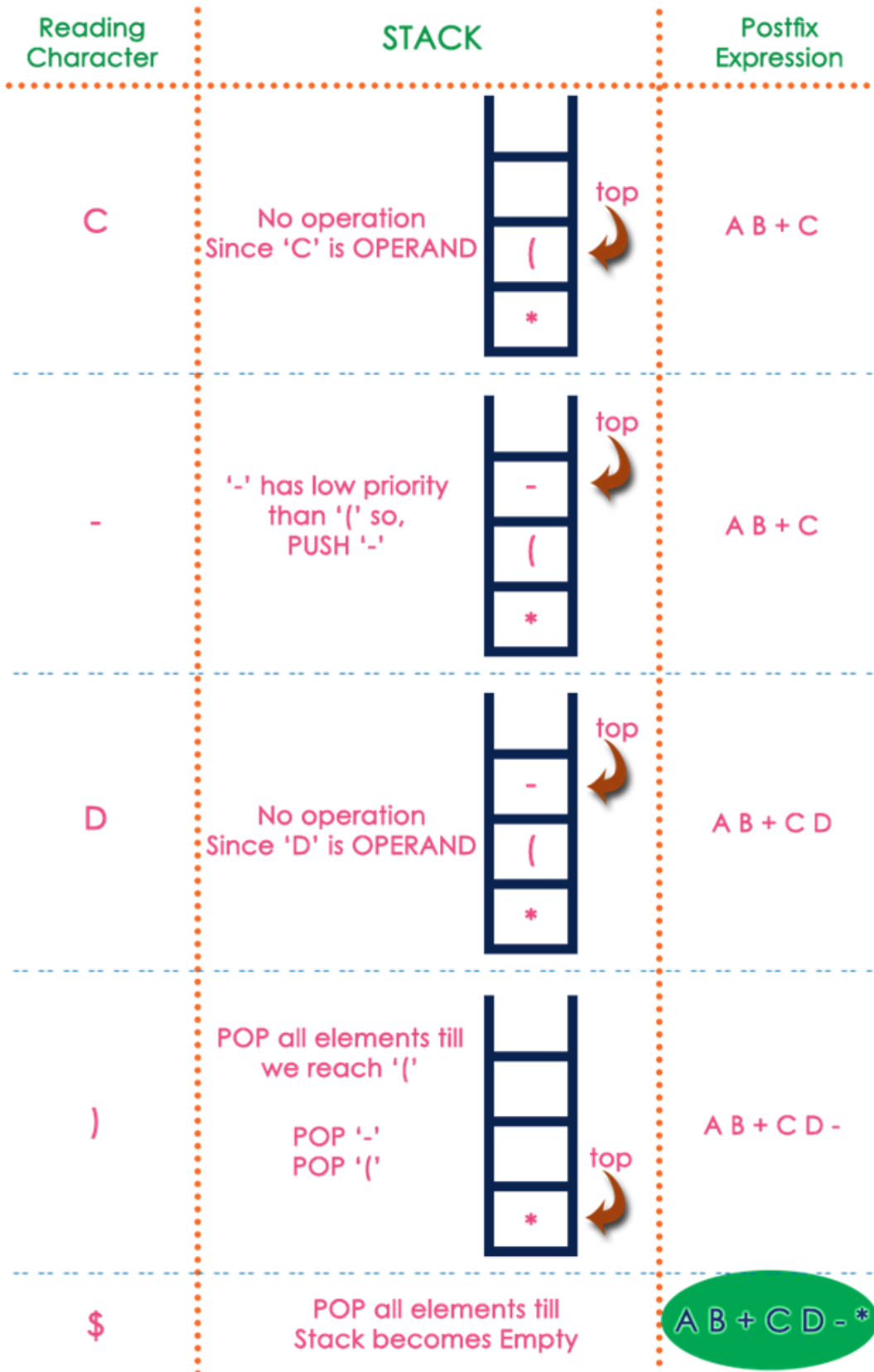
Consider the following Infix Expression...

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

The given infix expression can be converted into postfix expression using Stack Data Structure as follows...







Q2.8. Write a C program that uses Stack operations to Converting infix expression into postfix expression.

Answer:

```

#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include <string.h>

/* Variable declarations */
char infix_string[20], postfix_string[20];
int top;
int stack[20];

int pop();
int precedence(char symbol);
int isEmpty();
void infix_to_postfix();
void push(long int symbol);

int main()
{
    int count, length;
    char temp;
    top = -1;
    printf("\nINPUT THE INFIX EXPRESSION : ");
    scanf("%s", infix_string);
    infix_to_postfix();
    printf("\nEQUIVALENT POSTFIX EXPRESSION : %s\n", postfix_string);
    return 0;
}

/* Function to convert from infix to postfix */
void infix_to_postfix()
{
    int count, length, temp = 0;
    char next;
    char symbol;
    length = strlen(infix_string);
    for(count = 0; count < length; count++)
    {
        // Scanning the input expression
        symbol = infix_string[count];
    }
}

```

```

switch(symbol)
{
    case '(':
        push(symbol);
        break;
    case ')':
        // pop until '(' is encountered
        while((next = pop()) != '(')
            postfix_string[temp++] = next;
        break;
    case '+':
    case '-':
    case '*':
    case '/':
    case '%':
    case '^':
        while(!isEmpty() && precedence(stack[top]) >=
precedence(symbol)) // Check precedence and push the higher one
            postfix_string[temp++] = pop();
        push(symbol);
        break;
    default:
        postfix_string[temp++] = symbol;
}
}
while(!isEmpty())
    postfix_string[temp++] = pop();
postfix_string[temp] = '\0';
}

```

/* Function to check precedence of operators */

```

int precedence(char symbol)
{
    switch(symbol)
    {
        case '(':
            return 0;
        case '+':
        case '-':
            return 1;
        case '*':
        case '/':

```

```

        case '%':
            return 2;
        case '^':
            return 3;
        default:
            return 0;
    }
}

void push(long int symbol)
{
    if(top > 20)
    {
        printf("Stack Overflow\n");
        exit(1);
    }
    top = top + 1;
    // Push the symbol and make it as TOP
    stack[top] = symbol;
}

int isEmpty()
{
    if(top == -1)
        return 1;
    else
        return 0;
}

int pop()
{
    if(isEmpty())
    {
        printf("Stack is Empty\n");
        exit(1);
    }
    // Pop the symbol and decrement TOP
    return(stack[top--]);
}

```

Output:

INPUT THE INFIX EXPRESSION : 2+3*5 Prepared by Sushma.,Asst.Professor.,Dept of CSE.,PVKKIT
EQUIVALENT POSTFIX EXPRESSION : 235*+

Q2.9. Explain the process of postfix expression evaluation with an example.

Answer:

Postfix Expression Evaluation:

- A postfix expression can be evaluated using the Stack data structure.
- To evaluate a postfix expression using Stack data structure we can use the following steps...
 1. Create a stack to store operands (or values).
 2. Scan the given expression and do following for every scanned element.
 - a. If the element is a number, push it into the stack.
 - b. If the element is an operator, pop operands for the operator from stack. Evaluate the operator and push the result back to the stack.
 3. When the expression is ended, the number in the stack is the final answer.





Example:






Consider the following Expression...

Infix Expression **(5 + 3) * (8 - 2)**

Postfix Expression **5 3 + 8 2 - ***

Above Postfix Expression can be evaluated by using Stack Data Structure as follows...

Reading Symbol	Stack Operations	Evaluated Part of Expression
Initially	Stack is Empty 	Nothing
5	push(5) 	Nothing
3	push(3) 	Nothing
+	<pre>value1 = pop() value2 = pop() result = value2 + value1 push(result)</pre> 	<pre>value1 = pop(); // 3 value2 = pop(); // 5 result = 5 + 3; // 8 Push(8)</pre> <p>(5 + 3)</p>

Reading Symbol	Stack Operations	Stack	Evaluated Part of Expression
8	push(8)		(5 + 3)
2	push(2)		(5 + 3)
-	<pre>value1 = pop() value2 = pop() result = value2 - value1 push(result)</pre>		<pre>value1 = pop(); // 2 value2 = pop(); // 8 result = 8 - 2; // 6 Push(6) (8 - 2) (5 + 3) , (8 - 2)</pre>
*	<pre>value1 = pop() value2 = pop() result = value2 * value1 push(result)</pre>		<pre>value1 = pop(); // 6 value2 = pop(); // 8 result = 8 * 6; // 48 Push(48) (6 * 8) (5 + 3) * (8 - 2)</pre>
\$ End of Expression	result = pop()		<p>Display (result)</p> <p>48</p> <p>As final result</p>

Q2.10. Write a C program that uses Stack to evaluate the postfix expression.

Answer:

```

#include<stdio.h>
#include<ctype.h>
int stack[20];
int top = -1;

void push(int x)
{
    top++;
    stack[top] = x;
}

int pop()
{
    return stack[top--];
}

int main()
{
    char exp[20];
    char *e;
    int n1,n2,n3,num;
    printf("Enter the expression :: ");
    scanf("%s",exp);
    e = exp;
    while(*e != '\0')
    {
        if(isdigit(*e))
        {
            num = *e - 48;
            push(num);
        }
        else
        {
            n1 = pop();
            n2 = pop();
            switch(*e)
            {
                case '+':
                    n3 = n1 + n2;
                    push(n3);
                    break;
            }
        }
        e++;
    }
    printf("Result = %d",pop());
}

```

```

        case '-':
            n3 = n2 - n1;
            break;
        case '*':
            n3 = n1 * n2;
            break;
        case '/':
            n3 = n2 / n1;
            break;
    }
    push(n3);
}
e++;
}
printf("\nThe result of expression %s = %d",exp,pop());
return 0;
}

```

Output:

Enter the expression :: 235*+

The result of expression 235*+ = 17

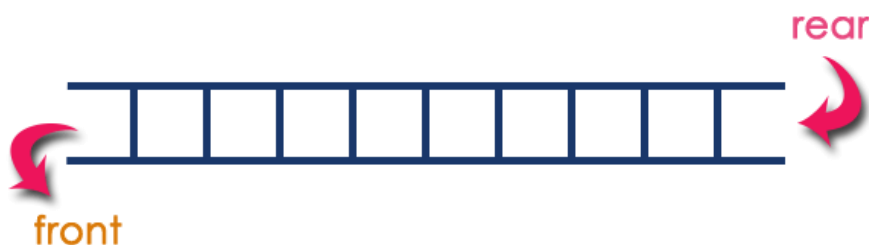
3. Queue**Q3.1. Define Queue.**

Answer:

Queue:

Queue data structure is a linear data structure in which the insertion and deletion operations are performed based on FIFO principle.

In a queue data structure, the insertion operation is performed at a position which is known as '**rear**' and the deletion operation is performed at a position which is known as '**front**'.



Example

Queue after inserting 25, 30, 51, 60 and 85.

After Inserting five elements...

**Q3.2. In how many ways a queue can be implemented?**

Answer:

Queue data structure can be implemented in two ways. They are as follows...

1. Using Array
2. Using Linked List

When a queue is implemented using an array, that queue can organize an only limited number of elements. When a queue is implemented using a linked list, that queue can organize an unlimited number of elements.

Q3.3. Explain different operations performed on queue.

Answer:

The following operations are performed on a queue data structure...

1. **insert(value)** - To insert an element into the queue

Algorithm:

```

Algorithm: void insert(int value)
Step-1: Start
Step-2: if rear == SIZE-1 then
            Display "Queue is Full"
Step-3: else
            If front == -1 then
                front = 0
            rear++;
            queue[rear] = value;
            Display "Insertion success"
Step-4: Stop
  
```

2. **delete()** - To delete an element from the queue

Algorithm:

```

Algorithm: void delete()
Step-1: Start
Step-2: if front == -1 || front > rear then
            Display "Queue is Empty"
Step-3: else
            Display "Deleted : " queue[front]
            If front == rear then
                front = rear = -1;
            front++;
Step-4: Stop
  
```

3. **display()** - To display the elements of the queue

Algorithm:

```

Algorithm: void display()
Step-1: Start
Step-2: if rear == -1 then
            Display "Queue is Empty"
Step-3: else
            Display "Queue elements are:"
            for(i=front; i<=rear; i++)
                Display queue[i]
Step-4: Stop
  
```

Q3.4. Write C programs that implement Queue using Arrays.

Answer:

```

#include<stdio.h>
#define SIZE 20

void insert(int);
void delete();
void display();

int queue[SIZE], front = -1, rear = -1;

int main()
{
    int value, choice;
  
```

```

do
{
    printf("\n\n***** MENU *****\n");
    printf("1. Insertion\n2. Deletion\n3. Display\n4. Exit");
    printf("\nEnter your choice: ");
    scanf("%d",&choice);
    switch(choice)
    {
        case 1:
            printf("Enter the value to be insert: ");
            scanf("%d",&value);
            insert(value);
            break;
        case 2:
            delete();
            break;
        case 3:
            display();
            break;
        case 4:
            printf("\nExiting...");
            break;
        default:
            printf("\nWrong selection!!! Try again!!!");
    }
}while(choice!=4);
return 0;
}

void insert(int value)
{
    if(rear == SIZE-1)
        printf("\nQueue is Full!!! Insertion is not possible!!!");
    else
    {
        if(front == -1)
            front = 0;
        rear++;
        queue[rear] = value;
        printf("\nInsertion success!!!");
    }
}

```

```

void delete()
{
    if(front == -1 || front > rear)
        printf("\nQueue is Empty!!! Deletion is not possible!!!");
    else
    {
        printf("\nDeleted : %d", queue[front]);
        if(front == rear)
            front = rear = -1;
        front++;
    }
}

void display()
{
    int i;
    if(rear == -1)
        printf("\nQueue is Empty!!!");
    else
    {
        printf("\nQueue elements are:\n");
        for(i=front; i<=rear; i++)
            printf("%d\t",queue[i]);
    }
}

```

Output:

***** MENU *****

1. Insertion
2. Deletion
3. Display
4. Exit

Enter your choice: 1

Enter the value to be insert: 10

Insertion success!!!

***** MENU *****

1. Insertion
2. Deletion
3. Display

4. Exit

Enter your choice: 1

Enter the value to be insert: 20

Insertion success!!!

***** MENU *****

1. Insertion
2. Deletion
3. Display
4. Exit

Enter your choice: 1

Enter the value to be insert: 30

Insertion success!!!

***** MENU *****

1. Insertion
2. Deletion
3. Display
4. Exit

Enter your choice: 3

Queue elements are:

10 20 30

***** MENU *****

1. Insertion
2. Deletion
3. Display
4. Exit

Enter your choice: 2

Deleted : 10

***** MENU *****

1. Insertion
2. Deletion
3. Display
4. Exit

Enter your choice: 3

Queue elements are:

20 30

***** MENU *****

1. Insertion
2. Deletion
3. Display
4. Exit

Enter your choice: 2

Deleted : 20

***** MENU *****

1. Insertion
2. Deletion
3. Display

4. Exit

Enter your choice: 3

Queue elements are:

30

***** MENU *****

1. Insertion
2. Deletion
3. Display
4. Exit

Enter your choice: 2

Deleted : 30

***** MENU *****

1. Insertion
2. Deletion
3. Display
4. Exit

Enter your choice: 2

Queue is Empty!!! Deletion is not possible!!!

***** MENU *****

1. Insertion
2. Deletion
3. Display
4. Exit

Enter your choice: 3

Queue is Empty!!!

***** MENU *****

1. Insertion
2. Deletion
3. Display
4. Exit

Enter your choice: 4

Q3.5. What are the limitations of linear queue? How they can be restricted?**Answer:**

In a normal Queue Data Structure, we can insert elements until queue becomes full. But once the queue becomes full, we cannot insert the next element until all the elements are deleted from the queue. For example, consider the queue below...

The queue after inserting all the elements into it is as follows...

Queue is Full

Now consider the following situation after deleting three elements from the queue...

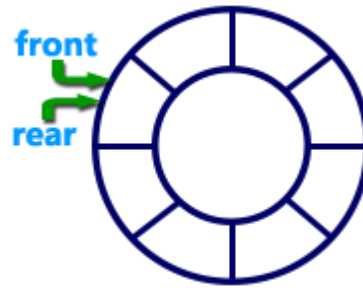
Queue is Full (Even three elements are deleted)

This situation also says that Queue is Full and we cannot insert the new element because 'rear' is still at last position. In the above situation, even though we have empty positions in the queue we cannot make use of them to insert the new element. This is the major problem in a normal queue data structure. To overcome this problem, we use a **circular queue** data structure.

Q3.6. What is a Circular Queue?**Answer:**

A circular queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle.

Graphical representation of a circular queue is as follows...



Q3.7. What is Double Ended Queue?

Answer:

Double Ended Queue is also a Queue data structure in which the insertion and deletion operations are performed at both the ends (**front** and **rear**). That means, we can insert at both front and rear positions and can delete from both front and rear positions.

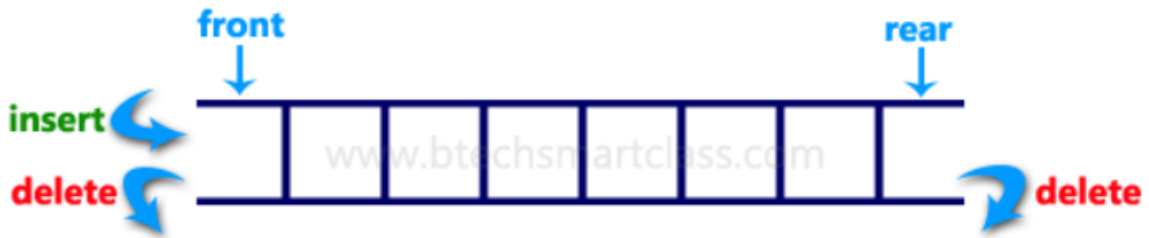


Double Ended Queue can be represented in TWO ways; those are as follows...

1. Input Restricted Double Ended Queue
2. Output Restricted Double Ended Queue

1. Input Restricted Double Ended Queue:

In input restricted double-ended queue, the insertion operation is performed at only one end and deletion operation is performed at both the ends.



2. Output Restricted Double Ended Queue:

In output restricted double ended queue, the deletion operation is performed at only one end and insertion operation is performed at both the ends.



Q3.8. Define a priority queue?

Answer:

Priority Queue:

Q3.9. What are the applications of queue?

Answer:

Queue is used in below applications:

1. Data getting transferred between the IO Buffers (Input Output Buffers).
2. CPU scheduling and Disk scheduling.
3. Managing shared resources between various processes.
4. Job scheduling algorithms.
5. Recognizing a palindrome.

2 Mark Questions

Introduction:

1. What is meant by abstract data type?
2. Define a data structure?
3. Classify different data structures?
4. What is a pointer array? Give example.

Stacks:

1. What are the basic operations that can be performed on stacks?
2. Assume that the operators +, -, x are left associative and ^ is right associative. The order of precedence (from highest to lowest) is ^, x, +, -. What is the postfix expression corresponding to the infix expression $a + b \times c - d \wedge e \wedge f$?
3. State the applications of stack.
4. Define PUSH and POP operations in a stack.
5. Consider the following stack of characters, where stack is allocated $N = 8$ memory cells. STACK:
A, C, D, F, K, _, _, _. Describe the stack as the following operations takes place.
 - (i) POP (STACK, ITEM)
 - (ii) POP (STACK, ITEM)
 - (iii) PUSH (STACK, R)
 - (iv) PUSH (STACK, L)
6. Define Push and Pop operations of stack.

Queues:

1. Give any two operations of queues.
2. What are the prerequisites for implementing the queue using array?
3. Define priority queue with diagram and give the operations.
4. Distinguish FIFO and LIFO of a queue.
5. How do you test for an empty queue?

Unit 4

Linked Lists

1. Linked Lists
2. Singly linked list: Insertion, deletion and searching operations
3. Dynamically linked stacks and queues
4. Polynomials using singly linked lists and using circularly linked lists
5. Doubly linked lists and its operations
6. Circular linked lists and its operations

1. Linked Lists

Q1.1. What are the limitations of arrays?

Answer:

- The number of elements to be stored in an array should be known in advance.
- An array is a static structure (which means the array is of fixed size). Once declared the size of the array cannot be modified. The memory which is allocated to it cannot be increased or decreased.
- Insertion and deletion are quite difficult in an array as the elements are stored in consecutive memory locations and the shifting operation is costly.
- Allocating more memory than the requirement leads to wastage of memory space and less allocation of memory also leads to a problem.

Q1.2. Define Linked List.

Answer:

The linked list is a linear data structure that contains a sequence of elements such that each element links to its next element in the sequence. Each element in a linked list is called "Node".

Q1.3. List out the types of linked list.

Answer:

Prepared by Sushma.,Asst.Professor.,Dept of CSE.,PVKKIT

There are 3 different implementations of Linked List available, they are:

1. Singly Linked List
2. Doubly Linked List
3. Circular Linked List (Single or Double)

Q1.3. List the applications of linked list.

Answer:

Applications of linked list:

1. Implementation of stacks and queues
2. Implementation of graphs: Adjacency list representation of graphs is most popular which uses linked list to store adjacent vertices.
3. Dynamic memory allocation: We use linked list of free blocks.
4. Maintaining directory of names
5. Performing arithmetic operations on long integers
6. Manipulation of polynomials by storing constants in the node of linked list
7. representing sparse matrices

2. Singly linked list: Insertion, deletion and searching operations

Q2.1. Define a Single Linked List.

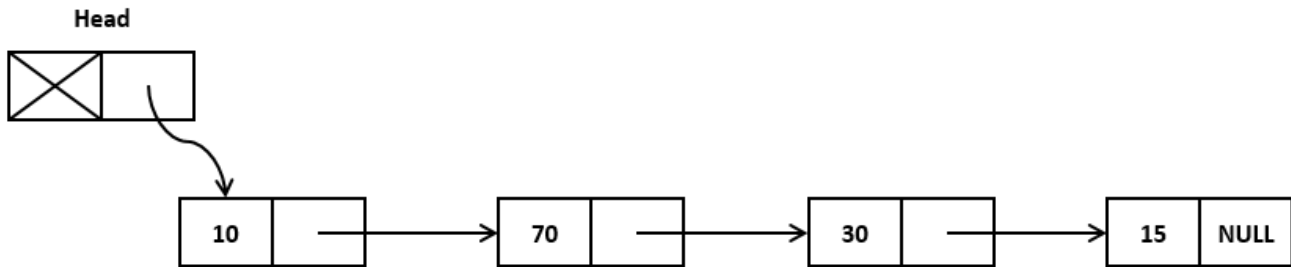
Answer:

Singly linked lists contain nodes which have a data part as well as an address part i.e. next, which points to the next node in the sequence of nodes.

Node Structure:



Example:



Q2.2. How to create a node of a single linked list in C?

Answer:

We can create a single linked list using self-referential structure as given below:

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

Q2.3. Explain the operations performed on single linked list.

Answer:

The following operations are performed on single linked list:

1. Insertion
2. Deletion
3. Traversal

1. Insertion:

In a single linked list, the insertion operation can be performed in three ways.

They are as follows:

Case-1: Inserting at Beginning of the list

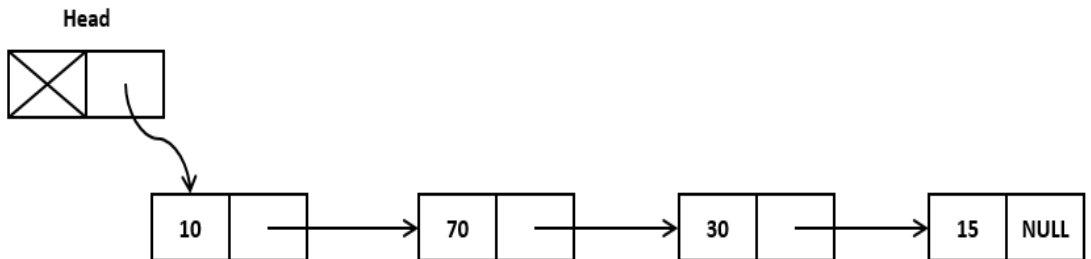
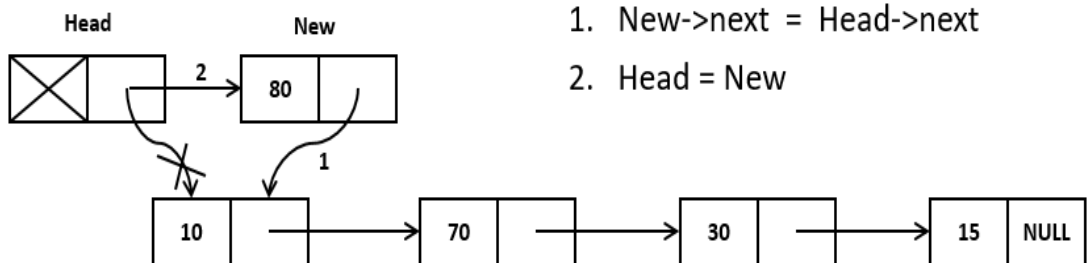
Case-2: Inserting at End of the list

Case-3: Inserting at Specific location in the list

Case-1: Inserting at Beginning of the list

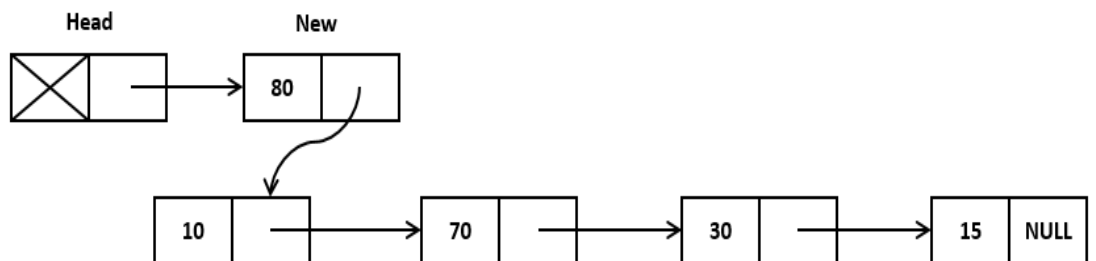
It involves inserting any element at the front of the list.

Example:

Initial**At insertion**

1. New->next = Head->next

2. Head = New

After insertion**Algorithm**

```
newNode = (struct Node*)malloc(sizeof(struct Node));
```

```
newNode->data = value;
```

```
newNode->next = head;
```

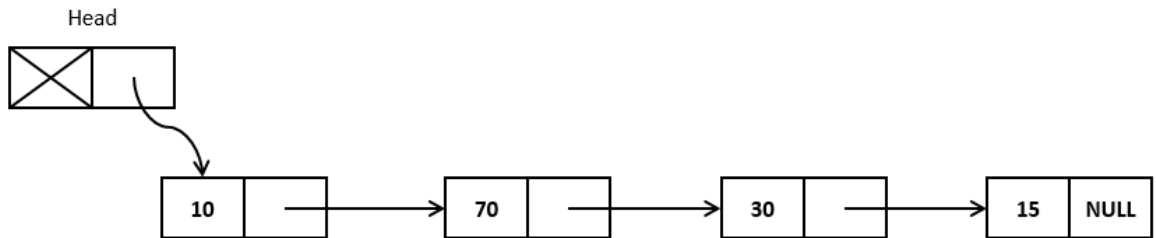
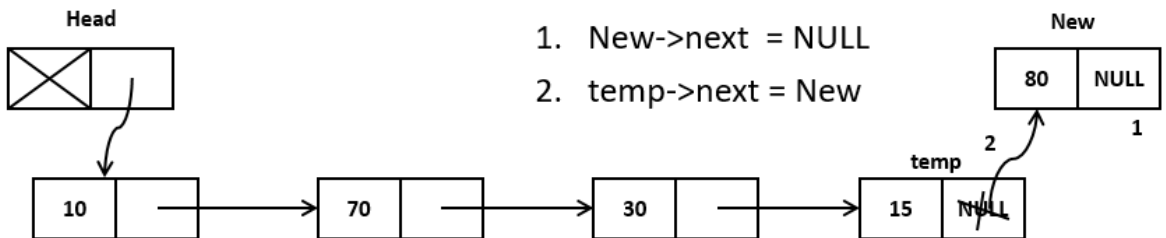
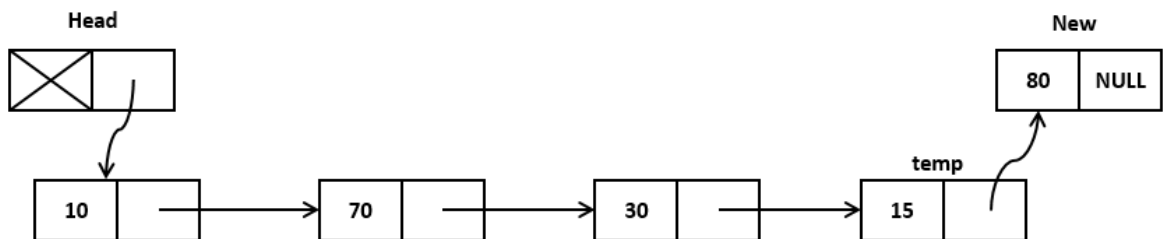
```
head = newNode;
```

Prepared by Sushma.,Asst.Professor.,Dept of CSE.,PVKKIT

Case-2: Inserting at End of the list

It involves insertion at the last of the linked list.

Example:

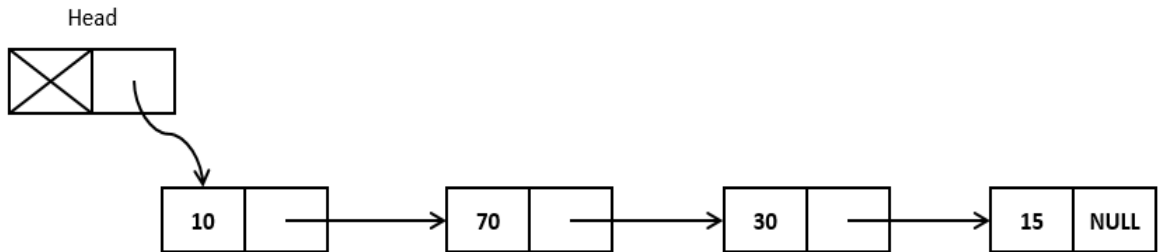
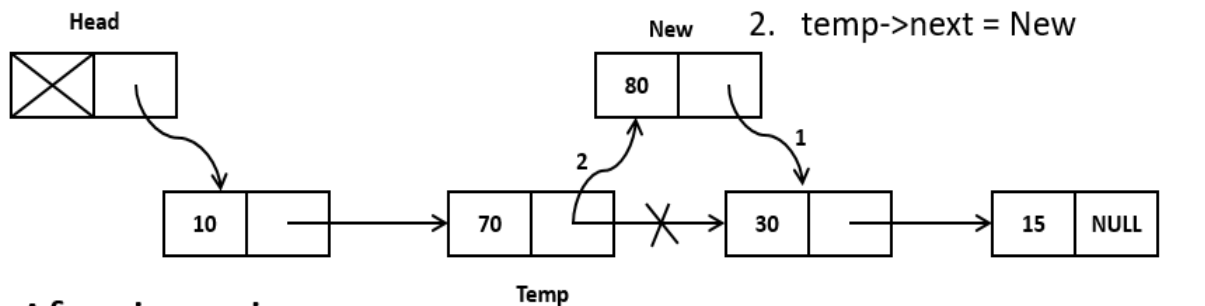
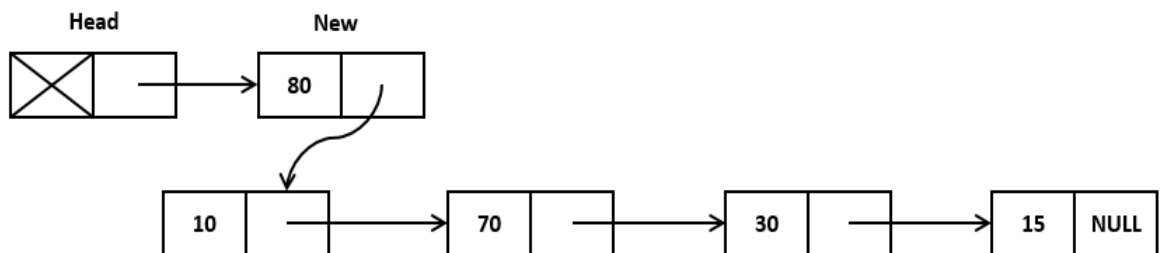
Initial**At insertion****After insertion****Algorithm**

```
newNode = (struct Node*)malloc(sizeof(struct Node));
newNode->data = value;
newNode->next = NULL;
temp->next = newNode;
```

Case-3: Inserting at Specific location in the list

It involves insertion after the specified position of the linked list.

Example:

Initial**At insertion****After insertion****Algorithm**

```

newNode = (struct Node*)malloc(sizeof(struct Node));
newNode->data = value;
newNode->next = temp->next;
temp->next = newNode;

```

2. Deletion:

In a single linked list, the deletion operation can be performed in three ways.

They are as follows:

Case-1: Deleting at Beginning of the list

Case-2: Deleting at End of the list

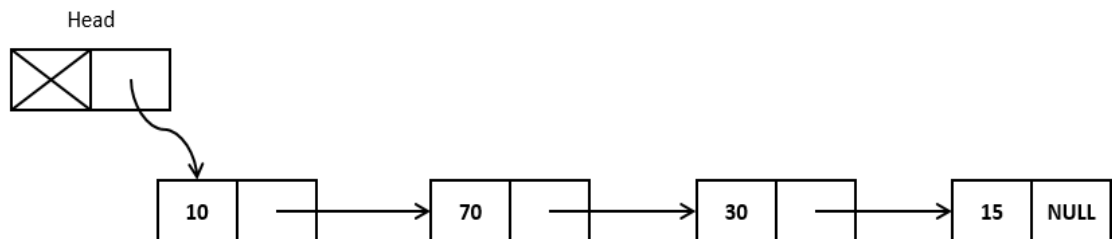
Case-3: Deleting at Specific location in the list

Case-1: Deleting at Beginning of the list

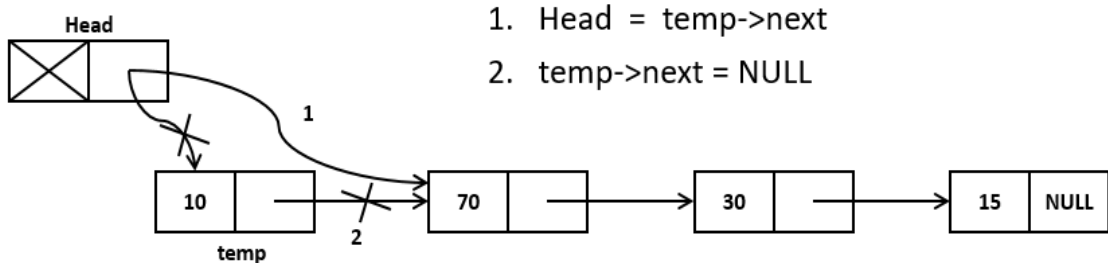
It involves deleting an element at the front of the list.

Example:

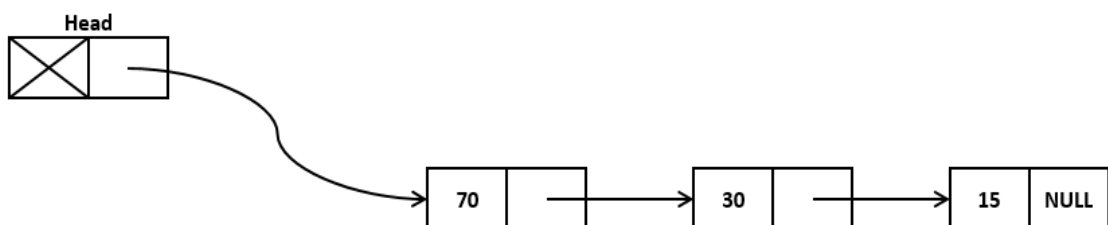
Initial



At Deletion



After Deletion



Algorithm

```
head = temp->next;
```

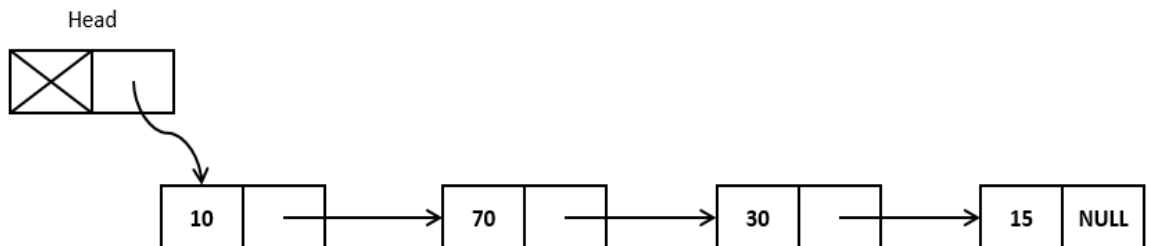
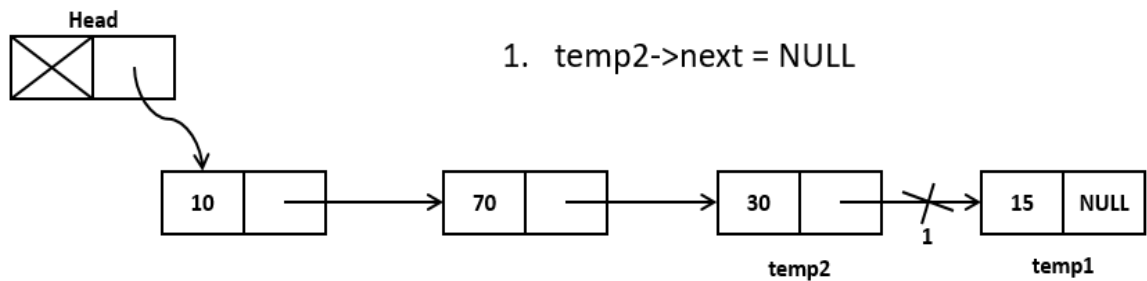
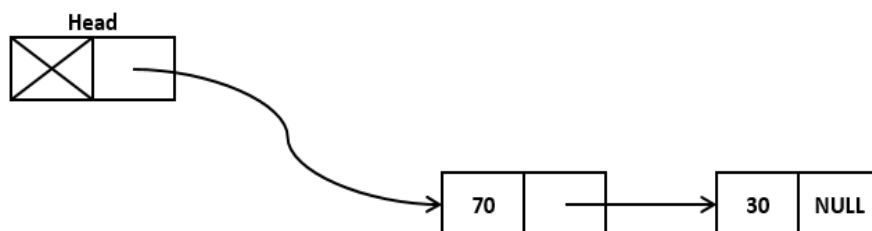
```
free(temp);
```

Prepared by Sushma.,Asst.Professor.,Dept of CSE.,PVKKIT

Case-2: Deleting at End of the list

It involves deletion at the last of the linked list.

Example:

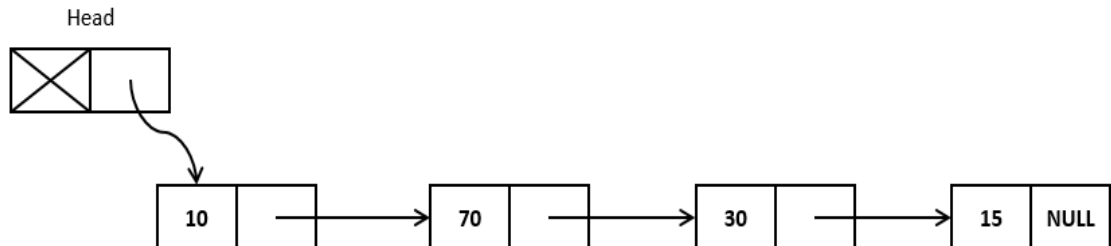
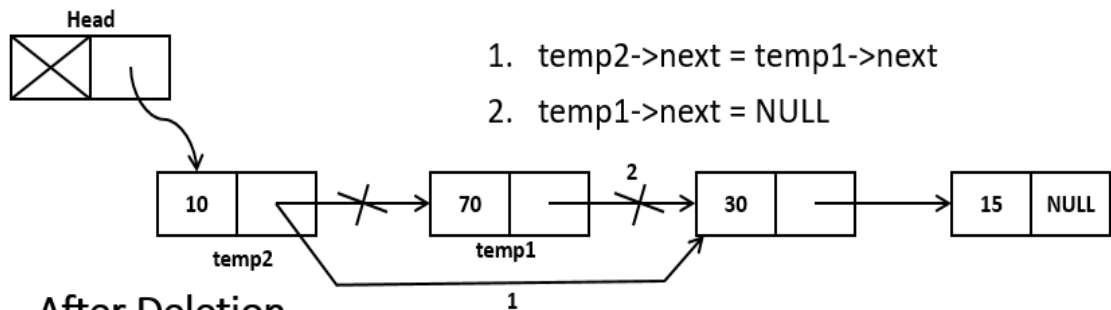
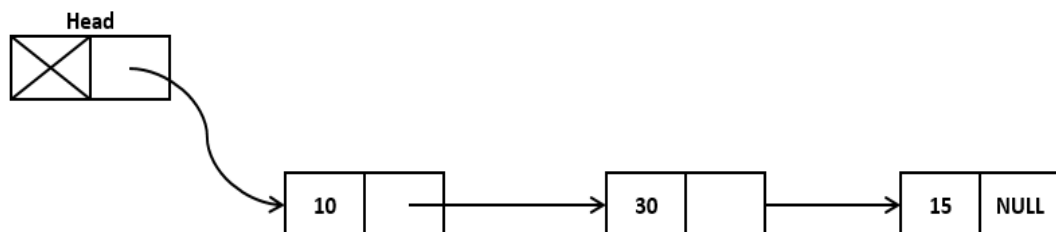
Initial**At Deletion****After Deletion****Algorithm**

```
temp2->next = NULL;
free(temp1);
```

Case-3: Deleting at Specific location in the list

It involves deletion at the specified position of the linked list.

Example:

Initial**At Deletion****After Deletion****Algorithm**

```
temp2->next = temp1->next;
free(temp1);
```

3. Traversal:

In traversing, we simply visit each node of the list at least once.

Algorithm

```
while(temp != NULL)
{
    printf("%d --->",temp->data);
    temp = temp->next;
}
```

Q2.3. Write a C program that uses functions to perform the following operations on singly linked list. i) Creation ii) Insertion iii) Deletion iv) Traversal

Answer:

```
//Implementation of Single Linked List using C Programming
#include<stdio.h>
#include<stdlib.h>
struct Node
{
    int data;
    struct Node *next;
}*head = NULL;

void insertAtBeginning(int value)
{
    struct Node *newNode;
    newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    if(head == NULL)
    {
        newNode->next = NULL;
        head = newNode;
    }
}
```

```
    else
    {
        newNode->next = head;
        head = newNode;
    }
    printf("\nOne node inserted!!!\n\n");
}

void insertAtEnd(int value)
{
    struct Node *newNode;
    newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = NULL;
    if(head == NULL)
        head = newNode;
    else
    {
        struct Node *temp = head;
        while(temp->next != NULL)
            temp = temp->next;
        temp->next = newNode;
    }
    printf("\nOne node inserted!!!\n\n");
}
```

```
void insertSpecific(int value)
```

```
{
    int pos;
    struct Node *newNode;
    newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
```

```

printf("Enter the Position where you wanto insert: ");
scanf("%d",&pos);
if(head == NULL)
{
    newNode->next = NULL;
    head = newNode;
}
else
{
    struct Node *temp = head;
    int i;
    for(i=1;i<pos;i++)
        temp = temp->next;
    newNode->next = temp->next;
    temp->next = newNode;
}
printf("\nOne node inserted!!!\n\n");
}

```

```

void removeBeginning()
{
    if(head == NULL)
        printf("\n\nList is Empty!!!");
    else
    {
        struct Node *temp = head;
        if(head->next == NULL)
            head = NULL;
        else
            head = temp->next;
        free(temp);
        printf("\nOne node deleted!!!\n\n");
    }
}

```



```
void removeEnd()
{
    if(head == NULL)
        printf("\n\nList is Empty!!!");
    else
    {
        struct Node *temp1 = head,*temp2;
        if(head->next == NULL)
            head = NULL;
        else
        {
            while(temp1->next != NULL)
            {
                temp2 = temp1;
                temp1 = temp1->next;
            }
            temp2->next = NULL;
        }
        free(temp1);
        printf("\nOne node deleted!!!\n\n");
    }
}
```

```
void removeSpecific()
{
    int pos, cnt = 0, i;
    struct Node *temp = head, *temp1;
    if (temp == NULL)
    {
        printf("List is Empty\n");
    }
}
```

```
else
{
    printf("\nEnter the position of value to be deleted:");
    scanf(" %d", &pos);
    if (pos == 1)
    {
        head = temp->next;
        printf("\nOne node deleted!!!\n\n");
    }
    else
    {
        while (temp != NULL)
        {
            temp = temp->next;
            cnt = cnt + 1;
        }
        if (pos > 0 && pos <= cnt)
        {
            temp = head;
            for (i = 1; i <= pos; i++)
            {
                temp1 = temp;
                temp = temp->next;
            }
            temp1->next = temp->next;
        }
        else
        {
            printf("Position is out of range");
        }
        free(temp);
        printf("\nOne node deleted!!!\n\n");
    }
}
}
```

```

void insert()
{
    int value, ch1;
    printf("Enter the value to be insert: ");
    scanf("%d",&value);
    printf("Where you want to insert: \n1. At Beginning\n2. At End\n\
3. Specific Position\nEnter your choice: ");
    scanf("%d",&ch1);
    switch(ch1)
    {
        case 1:
            insertAtBeginning(value);
            break;
        case 2:
            insertAtEnd(value);
            break;
        case 3:
            insertSpecific(value);
            break;
        default:
            printf("\nWrong Input!! Try again!!!\n\n");
    }
}

```

```

void delete()
{
    int ch1;
    printf("How do you want to Delete: \n1. From Beginning\n2. From End\n\
3. Spesific Position\nEnter your choice: ");
    scanf("%d",&ch1);

```

```
switch(ch1)
{
    case 1:
        removeBeginning();
        break;

    case 2:
        removeEnd();
        break;

    case 3:
        removeSpecific();
        break;

    default:
        printf("\nWrong Input!! Try again!!!\n\n");
}
}

void display()
{
    if(head == NULL)
    {
        printf("\nList is Empty\n");
    }
    else
    {
        struct Node *temp = head;
        printf("\n\nList elements are: \n");
        while(temp != NULL)
        {
            printf("%d --->",temp->data);
            temp = temp->next;
        }
    }
}
}
```

```

int main()
{
    int choice;
    do
    {
        printf("\n\n***** MENU *****\n1. Insert\n2. Display\n\
3. Delete\n4. Exit\nEnter your choice: ");
        scanf("%d",&choice);
        switch(choice)
        {
            case 1:
                insert();
                break;
            case 2:
                display();
                break;
            case 3:
                delete();
                break;
            case 4:
                printf("Exiting.....\n");
                break;
            default:
                printf("\nWrong input!!! Try again!!\n\n");
        }
    }while(choice != 4);
}

```

Output:

***** MENU *****

1. Insert
2. Display
3. Delete
4. Exit

Enter your choice: 1

Enter the value to be insert: 10

Where you want to insert:

1. At Beginning
 2. At End
 3. Specific Position
- Enter your choice: 1

One node inserted!!!

***** MENU *****

1. Insert
2. Display
3. Delete
4. Exit

Enter your choice: 1

Enter the value to be insert: 20

Where you want to insert:

1. At Beginning
2. At End
3. Specific Position

Enter your choice: 2

One node inserted!!!

***** MENU *****

1. Insert
2. Display
3. Delete
4. Exit

Enter your choice: 1

Enter the value to be insert: 30

Where you want to insert:

1. At Beginning
2. At End
3. Specific Position

Enter your choice: 3

Enter the Position where you wanto
insert: 1

One node inserted!!!

***** MENU *****

1. Insert
2. Display
3. Delete
4. Exit

Enter your choice: 2

List elements are:

10 --->30 --->20 --->

***** MENU *****

1. Insert
2. Display
3. Delete
4. Exit

Enter your choice: 1

Enter the value to be insert: 40

Where you want to insert:

1. At Beginning
2. At End
3. Specific Position

Enter your choice: 2

One node inserted!!!

***** MENU *****

1. Insert
2. Display
3. Delete
4. Exit

Enter your choice: 2

List elements are:

10 --->30 --->20 --->40 --->

***** MENU *****

1. Insert
2. Display
3. Delete
4. Exit

Enter your choice: 3

How do you want to Delete:

1. From Beginning
2. From End
3. Spesific

Enter your choice: 1

One node deleted!!!

***** MENU *****

1. Insert
2. Display
3. Delete
4. Exit

Enter your choice: 2

List elements are:

30 --->20 --->40 --->

***** MENU *****

1. Insert
2. Display
3. Delete
4. Exit

Enter your choice: 3

How do you want to Delete:

1. From Beginning
2. From End
3. Specific

Enter your choice: 2

One node deleted!!!

***** MENU *****

1. Insert
2. Display
3. Delete
4. Exit

Enter your choice: 2

List elements are:

30 --->20 --->

***** MENU *****

1. Insert
2. Display
3. Delete
4. Exit

Enter your choice: 3

How do you want to Delete:

1. From Beginning
2. From End
3. Specific

Enter your choice: 3

Enter the position of value to be deleted:1

One node deleted!!!

***** MENU *****

1. Insert
2. Display
3. Delete
4. Exit

Enter your choice: 3

How do you want to Delete:

1. From Beginning
2. From End
3. Specific

Enter your choice: 1

One node deleted!!!

***** MENU *****

1. Insert
2. Display
3. Delete
4. Exit

Enter your choice: 2

List is Empty

***** MENU *****

1. Insert
2. Display
3. Delete
4. Exit

Enter your choice: 4

Exiting.....

3. Dynamically linked stacks and queues

Q3.1. Implement stack and its operations using linked list.

```
#include <stdio.h>
#include <stdlib.h>
struct node
{
    int val;
    struct node *next;
}*head=NULL;

void push ()
{
    int val;
    struct node *newNode = (struct node*)malloc(sizeof(struct node));
    if(newNode == NULL)
        printf("Not able to push the element");
    else
    {
        printf("Enter the value: ");
        scanf("%d",&val);
        if(head==NULL)
        {
            newNode->val = val;
            newNode -> next = NULL;
            head=newNode;
        }
        else
        {
            newNode->val = val;
            newNode->next = head;
            head=newNode;
        }
        printf("Item pushed");
    }
}
```



```
void pop()
{
    int item;
    struct node *newNode;
    if (head == NULL)
        printf("Underflow");
    else
    {
        item = head->val;
        newNode = head;
        head = head->next;
        free(newNode);
        printf("Item popped");
    }
}

void display()
{
    int i;
    struct node *newNode;
    newNode=head;
    if(newNode == NULL)
        printf("Stack is empty\n");
    else
    {
        printf("Printing Stack elements \n");
        while(newNode!=NULL)
        {
            printf("%d\n",newNode->val);
            newNode = newNode->next;
        }
    }
}
```

```
int main ()
{
    int choice;
    printf("\n****Stack operations using linked list****\n");
    do
    {
        printf("\n-----MENU-----\n");
        printf("\n1.Push\n2.Pop\n3.Display\n4.Exit");
        printf("\n Enter your choice: ");
        scanf("%d",&choice);
        switch(choice)
        {
            case 1:
                push();
                break;
            case 2:
                pop();
                break;
            case 3:
                display();
                break;
            case 4:
                printf("Exiting....");
                break;
            default:
                printf("Please Enter valid choice ");
        }
    }while(choice != 4);
}
```

Output:

****Stack operations using linked list****

-----MENU-----

- 1.Push
- 2.Pop
- 3.Display
- 4.Exit

Enter your choice: 1

Enter the value: 10

Item pushed

-----MENU-----

- 1.Push
- 2.Pop
- 3.Display
- 4.Exit

Enter your choice: 1

Enter the value: 20

Item pushed

-----MENU-----

- 1.Push
- 2.Pop
- 3.Display
- 4.Exit

Enter your choice: 1

Enter the value: 30

Item pushed

-----MENU-----

- 1.Push
- 2.Pop

3.Display

4.Exit

Enter your choice: 3

Printing Stack elements

30

20

10

-----MENU-----

1.Push

2.Pop

3.Display

4.Exit

Enter your choice: 2

Item popped

-----MENU-----

1.Push

2.Pop

3.Display

4.Exit

Enter your choice: 3

Printing Stack elements

20

10

-----MENU-----

1.Push

2.Pop

3.Display

4.Exit

Enter your choice: 4

Exiting....

Q3.2. Implement queue and its operations using linked list.

```
#include<stdio.h>
#include<stdlib.h>

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

void insert()
{
    struct node *newNode;
    int item;
    newNode = (struct node *) malloc (sizeof(struct node));
    if(newNode == NULL)
    {
        printf("\nOVERFLOW\n");
        return;
    }
    else
    {
        printf("\nEnter value: ");
        scanf("%d",&item);
        newNode -> data = item;
        if(front == NULL)
        {
            front = newNode;
            rear = newNode;
            front -> next = NULL;
            rear -> next = NULL;
        }
        else
```

```
{
    rear -> next = newNode;
    rear = newNode;
    rear->next = NULL;
}
}
```

void delete ()

```
{
    struct node *newNode;
    if(front == NULL)
    {
        printf("\nUNDERFLOW\n");
        return;
    }
    else
    {
        newNode = front;
        front = front -> next;
        free(newNode);
    }
}
```

void display()

```
{
    struct node *newNode;
    newNode = front;
    if(front == NULL)
    {
        printf("\nEmpty queue\n");
    }
    else
```

```

{
    printf("\nQueue values ..... \n");
    while(newNode != NULL)
    {
        printf("\n%d\n",newNode -> data);
        newNode = newNode -> next;
    }
}

int main ()
{
    int choice;
    printf("\n****Queue operations using linked list****\n");
    do
    {
        printf("\n-----MENU-----\n");
        printf("\n1.insert an element\n2.Delete an element\n3.Display the
queue\n4.Exit\n");
        printf("\nEnter your choice: ");
        scanf("%d", &choice);
        switch(choice)
        {
            case 1:
                insert();
                break;
            case 2:
                delete();
                break;
            case 3:
                display();
                break;
            case 4:
                printf("Exiting....");
                break;
            default:
                printf("\nEnter valid choice??\n");
        }
    }while(choice != 4);
}

```

Output:

****Queue operations using linked list****

-----MENU-----

- 1.insert an element
- 2.Delete an element
- 3.Display the queue
- 4.Exit

Enter your choice: 1

Enter value: 10

-----MENU-----

- 1.insert an element
- 2.Delete an element
- 3.Display the queue
- 4.Exit

Enter your choice: 1

Enter value: 20

-----MENU-----

- 1.insert an element
- 2.Delete an element
- 3.Display the queue
- 4.Exit

Enter your choice: 1

Enter value: 30

-----MENU-----

- 1.insert an element
- 2.Delete an element
- 3.Display the queue
- 4.Exit

Enter your choice: 3

Queue values

10

20

30

-----MENU-----

- 1.insert an element
- 2.Delete an element
- 3.Display the queue
- 4.Exit

Enter your choice: 2

-----MENU-----

- 1.insert an element
- 2.Delete an element
- 3.Display the queue
- 4.Exit

Enter your choice: 3

Queue values

20

30

-----MENU-----

- 1.insert an element
- 2.Delete an element
- 3.Display the queue
- 4.Exit

Enter your choice: 4

Exiting....

4. Polynomials using singly linked lists and using circularly linked lists

Q4.1. How to represent polynomials using single linked list and circular linked list. Explain.

Answer:

Polynomial representation using single linked list:

- One of the application of single linked list is that, we can represent polynomials using linked list.
- Let, we want to represent the polynomial:

$$A(x) = a_{m-1}x^{e_{m-1}} + \dots + a_0x^{e_0}$$

Where a_i are the non-zero coefficients,

e_i are non-negative integer exponents,

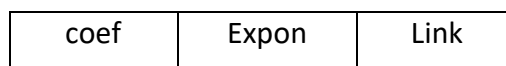
such that $e_{m-1} > e_{m-2} > \dots > e_1 > e_0$

- We represent each term as a node containing coefficient and exponent fields as well as pointer field to the next term.
- We can create a node as follows:

```

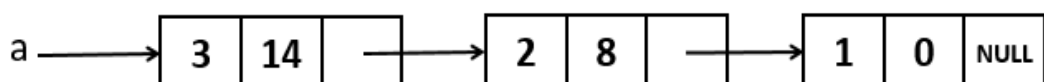
struct polyNode
{
    int coef;
    int expon;
    polynode *link;
};
  
```

- Diagrammatic represent of polyNode is given below:



- **Example:**

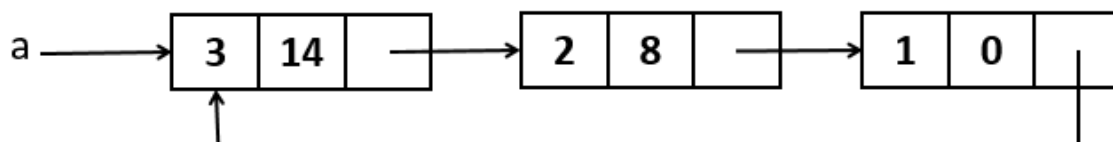
$$a = 3x^{14} + 2x^8 + 1$$



Polynomial representation using circular linked list:

- In circular linked list, the link field of last node is pointing to the first node.
- We can use circular linked list to represent polynomials.
- **Example:**

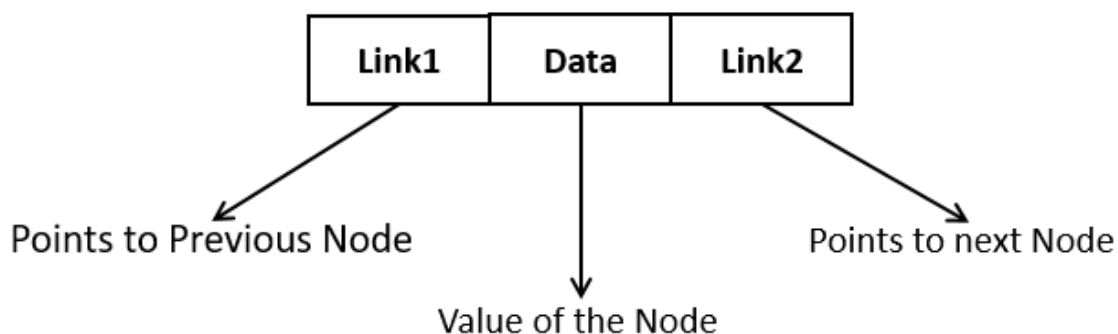
$$a = 3x^{14} + 2x^8 + 1$$

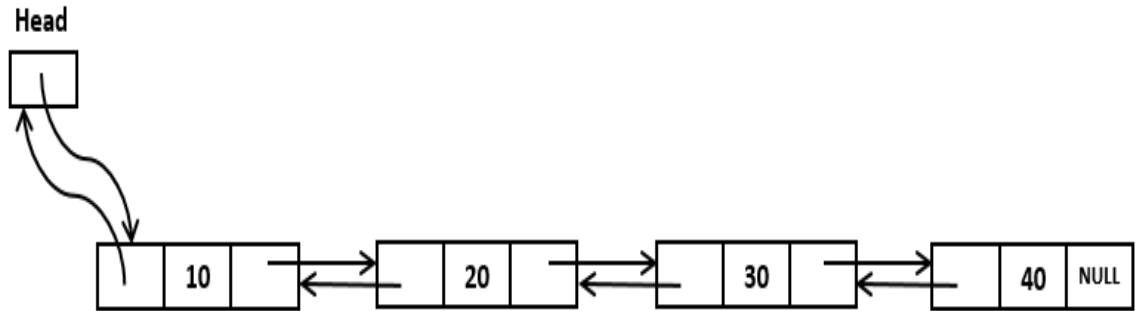
**5. Doubly linked lists and its operations**

Q5.1. Define double linked list with an example.

Answer:

In a doubly linked list, each node contains a data part and two addresses, one for the previous node and another for the next node.

Node Structure:

Example:

Q5.2. Explain different operations performed on double linked list with example.

Answer:

The following operations are performed on double linked list:

1. Insertion
2. Deletion
3. Traversal

1. Insertion:

In a double linked list, the insertion operation can be performed in three ways. They are as follows:

Case-1: Inserting at Beginning of the list

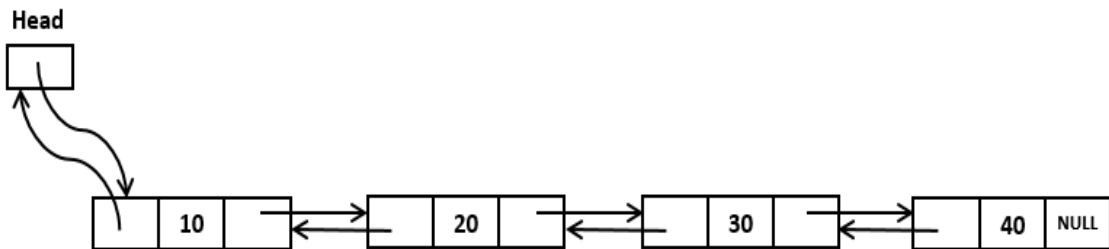
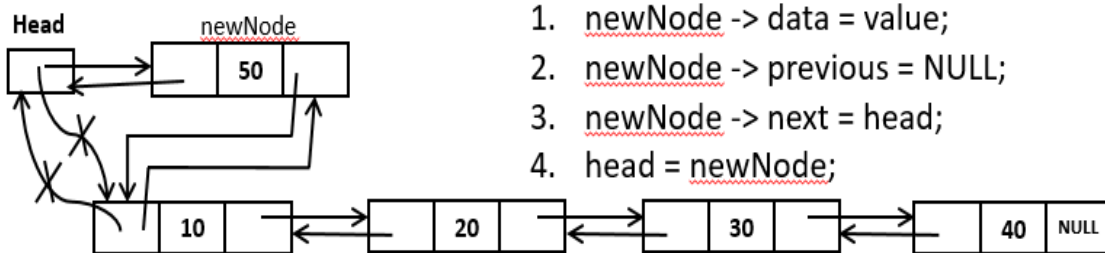
Case-2: Inserting at End of the list

Case-3: Inserting at Specific location in the list

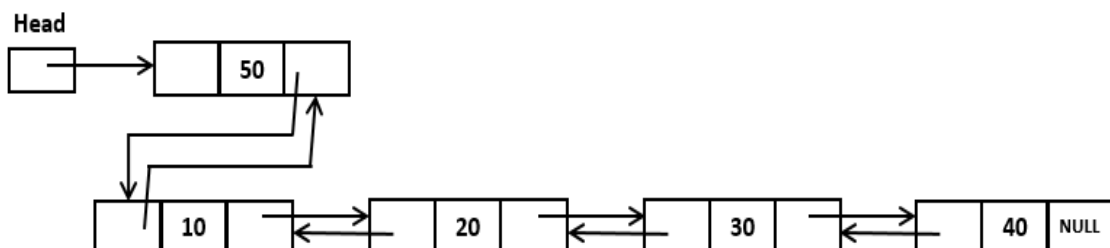
Case-1: Inserting at Beginning of the list

It involves inserting any element at the front of the list.

Example:

Initial**At Insertion**

1. newNode -> data = value;
2. newNode -> previous = NULL;
3. newNode -> next = head;
4. head = newNode;

After Insertion**Algorithm**

```

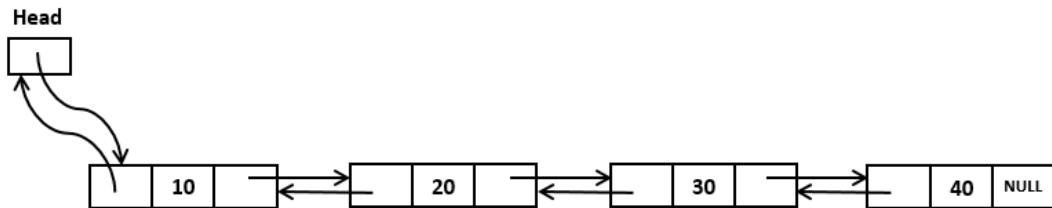
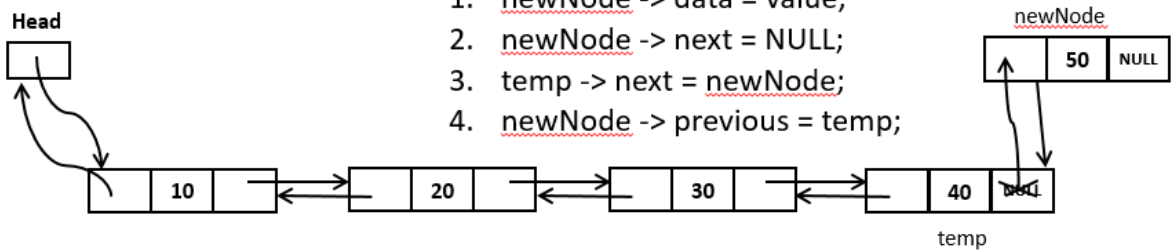
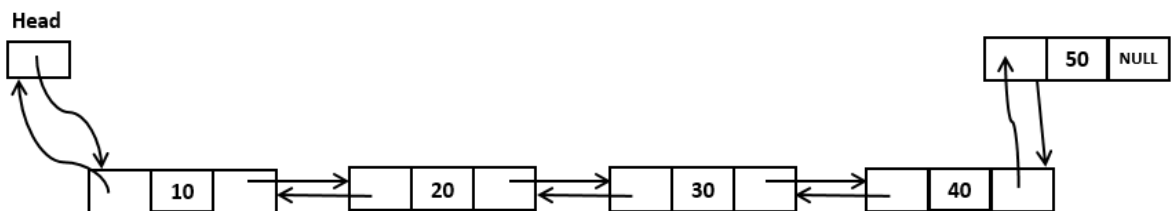
newNode = (struct Node*)malloc(sizeof(struct Node));
newNode -> data = value;
newNode -> previous = NULL;
newNode -> next = head;
head = newNode;

```

Case-2: Inserting at End of the list

It involves insertion at the last of the double linked list.

Example:

Initial**At Insertion****After Insertion****Algorithm**

```

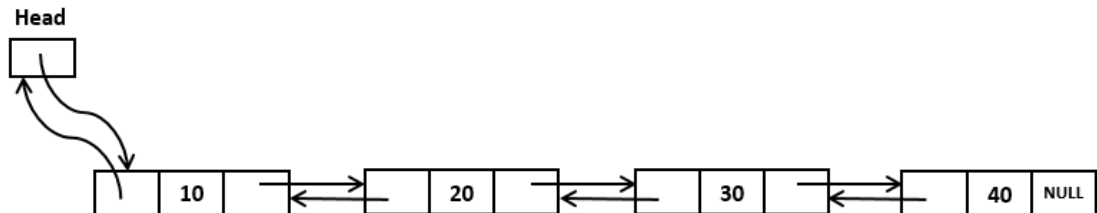
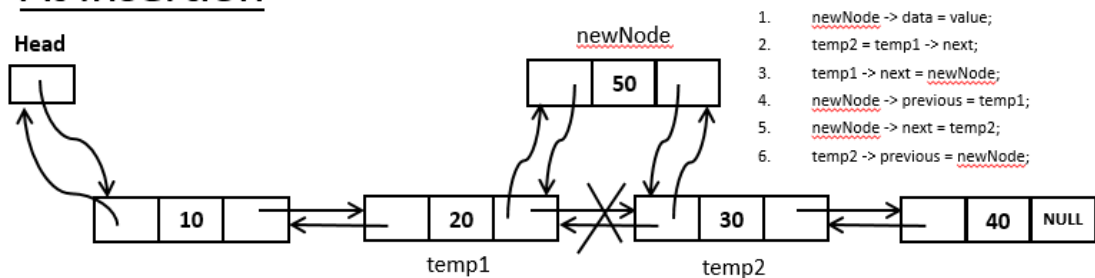
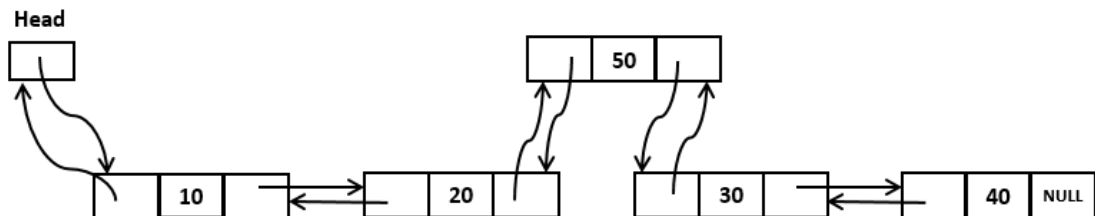
newNode = (struct Node*)malloc(sizeof(struct Node));
newNode -> data = value;
newNode -> next = NULL;
temp -> next = newNode;
newNode -> previous = temp;

```

Case-3: Inserting at Specific location in the list

It involves insertion after the specified position of the double linked list.

Example:

Initial**At Insertion****After Insertion****Algorithm**

```

newNode = (struct Node*)malloc(sizeof(struct Node));
newNode -> data = value;
temp2 = temp1 -> next;
temp1 -> next = newNode;
newNode -> previous = temp1;
newNode -> next = temp2;
temp2 -> previous = newNode;

```

1. Deletion:

In a double linked list, the deletion operation can be performed in three ways. They are as follows:

Case-1: Deleting at Beginning of the list

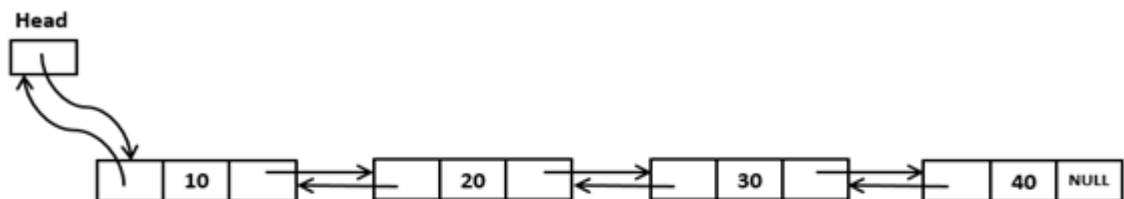
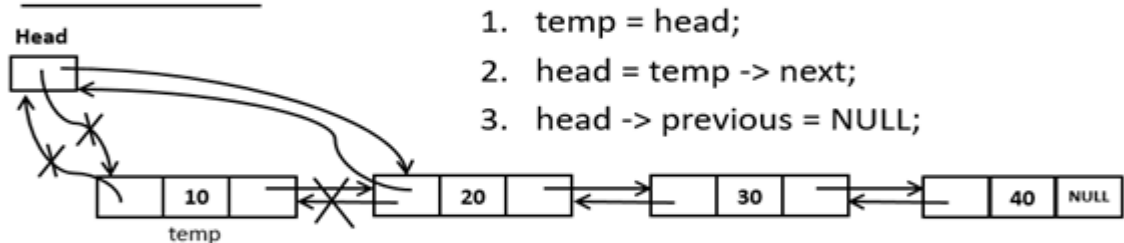
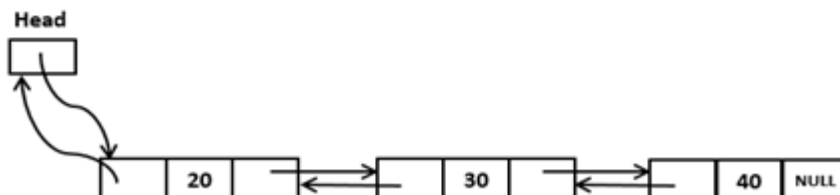
Case-2: Deleting at End of the list

Case-3: Deleting at Specific location in the list

Case-1: Deleting at Beginning of the list

It involves deleting an element at the front of the list.

Example:

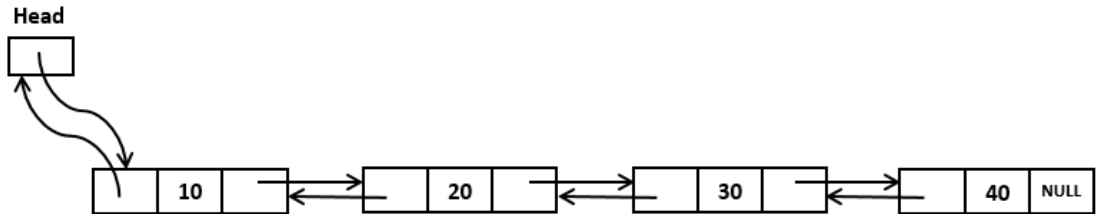
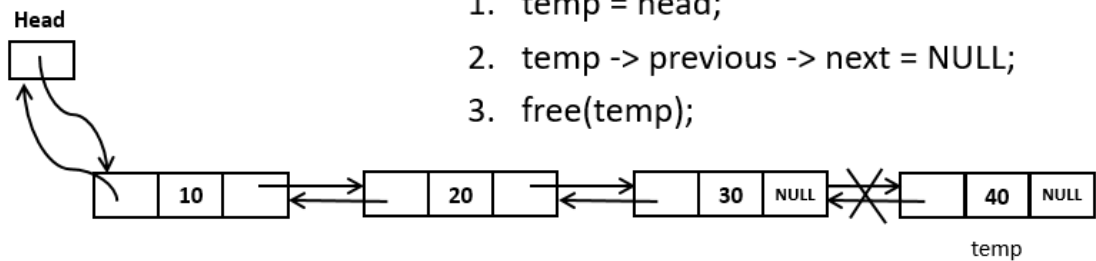
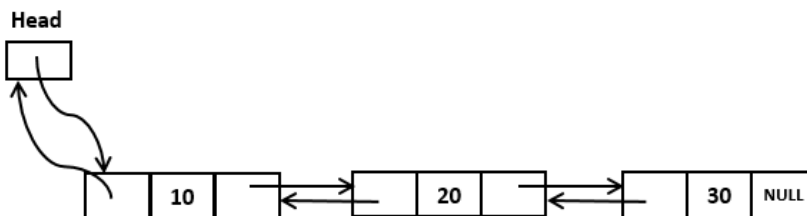
Initial**At Deletion****After Deletion****Algorithm**

```
temp = head;
head = temp -> next;
head -> previous = NULL;
free(temp);
```

Case-2: Deleting at End of the list

It involves deletion at the last of the double linked list.

Example:

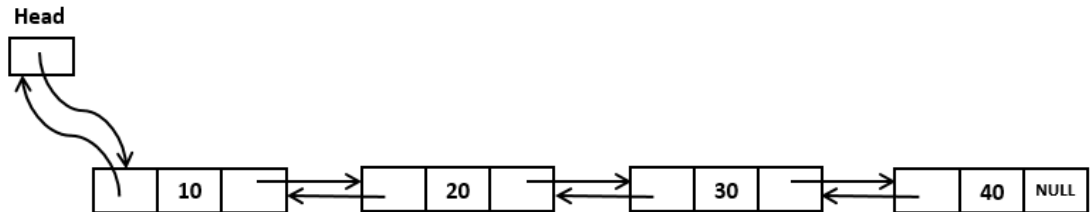
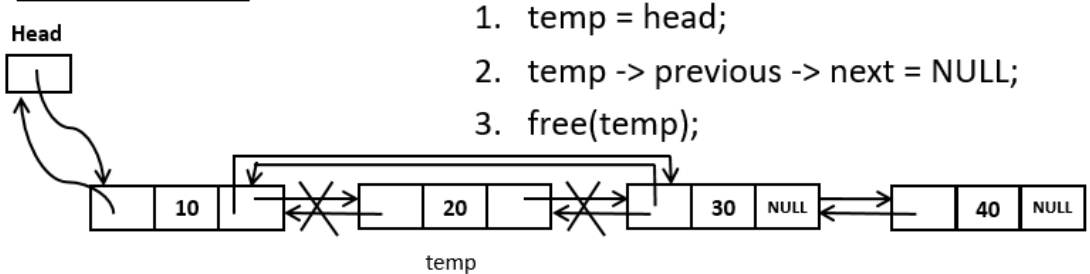
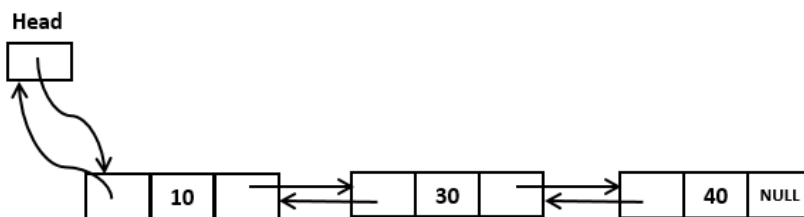
Initial**At Deletion****After Deletion****Algorithm**

1. temp = head;
2. temp -> previous -> next = NULL;
3. free(temp);

Case-3: Deleting at Specific location in the list

It involves deletion at the specified position of the double linked list.

Example:

Initial**At Deletion****After Deletion****Algorithm**

1. temp = head;
2. temp -> previous -> next = NULL;
3. free(temp);

3. Traversal:

In traversing, we simply visit each node of the list at least once.

Algorithm

```
temp = head;
printf("\nList elements are: \n");
while(temp != NULL)
{
    printf("%d ---> ",temp -> data);
    temp = temp->next;
}
```

Q5.3. Write a C program that uses functions to perform the following operations on Doubly linkedlist. i) Creation ii) Insertion iii) Deletion iv) Traversal

Answer:

Program:

```
#include<stdio.h>
#include<stdlib.h>

struct Node
{
    int data;
    struct Node *previous, *next;
}*head = NULL;

void insertAtBeginning(int value)
{
    struct Node *newNode;
    newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode -> data = value;
    newNode -> previous = NULL;
    if(head == NULL)
    {
        newNode -> next = NULL;
        head = newNode;
    }
}
```

```

else
{
    newNode -> next = head;
    head = newNode;
}
printf("\nInsertion success!!!");
}

void insertAtEnd(int value)
{
    struct Node *newNode, *temp;
    newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode -> data = value;
    newNode -> next = NULL;
    if(head == NULL)
    {
        newNode -> previous = NULL;
        head = newNode;
    }
    else
    {
        temp = head;
        while(temp -> next != NULL)
            temp = temp -> next;
        temp -> next = newNode;
        newNode -> previous = temp;
    }
    printf("\nInsertion success!!!");
}

void insertAfter(int value, int location)
{
    struct Node *newNode, *temp1, *temp2;
    newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode -> data = value;
    if(head == NULL)
    {
        newNode -> previous = newNode -> next = NULL;
        head = newNode;
    }
}

```

```

else
{
    temp1 = head;
    while(temp1 -> data != location)
    {
        if(temp1 -> next == NULL)
        {
            printf("Given node is not found in the list!!!");
            return;
        }
        else
        {
            temp1 = temp1 -> next;
        }
    }
    temp2 = temp1 -> next;
    temp1 -> next = newNode;
    newNode -> previous = temp1;
    newNode -> next = temp2;
    temp2 -> previous = newNode;
    printf("\nInsertion success!!!");
}
}
void deleteBeginning()
{
    struct Node *temp;
    if(head == NULL)
        printf("List is Empty!!! Deletion not possible!!!");
    else
    {
        temp = head;
        if(temp -> previous == temp -> next)
        {
            head = NULL;
            free(temp);
        }
        else
        {
            head = temp -> next;
            head -> previous = NULL;
            free(temp);
        }
    }
}

```

```

        printf("\nDeletion success!!!");
    }
}

void deleteEnd()
{
    struct Node *temp;
    if(head == NULL)
        printf("List is Empty!!! Deletion not possible!!!");
    else
    {
        temp = head;
        if(temp -> previous == temp -> next)
        {
            head = NULL;
            free(temp);
        }
        else
        {
            while(temp -> next != NULL)
                temp = temp -> next;
            temp -> previous -> next = NULL;
            free(temp);
        }
        printf("\nDeletion success!!!");
    }
}

```

```

void deleteSpecific(int delValue)
{
    struct Node *temp;
    if(head == NULL)
        printf("List is Empty!!! Deletion not possible!!!");
    else
    {
        temp = head;
        while(temp -> data != delValue)
        {
            if(temp -> next == NULL)
            {
                printf("\nGiven node is not found in the list!!!");
            }
        }
    }
}

```

```

        return;
    }
    else
    {
        temp = temp -> next;
    }
}
if(temp == head)
{
    head = NULL;
    free(temp);
}
else
{
    temp -> previous -> next = temp -> next;
    free(temp);
}
printf("\nDeletion success!!!");
}
}

```

```
void display()
```

```

{
    struct Node *temp;
    if(head == NULL)
        printf("\nList is Empty!!!");
    else
    {
        temp = head;
        printf("\nList elements are: \n");
        while(temp != NULL)
        {
            printf("%d ---> ",temp -> data);
            temp = temp->next;
        }
    }
}

```

```
void insert()
```

```

{
    int choice, value, location;
    printf("Enter the value to be inserted: ");
}

```

```

scanf("%d",&value);
printf("\nSelect from the following Inserting options\n");
printf("1. At Beginning\n2. At End\n3. After a Node\nEnter your choice: ");
scanf("%d",&choice);
switch(choice)
{
    case 1:
        insertAtBeginning(value);
        break;
    case 2:
        insertAtEnd(value);
        break;
    case 3:
        printf("Enter the location after which you want to insert: ");
        scanf("%d",&location);
        insertAfter(value,location);
        break;
    default:
        printf("\nPlease select correct Inserting option!!!\n");
}
}

void delete()
{
    int choice,location;
    printf("\nSelect from the following Deleting options\n");
    printf("1. At Beginning\n2. At End\n3. Specific Node\nEnter your choice: ");
    scanf("%d",&choice);
    switch(choice)
    {
        case 1:
            deleteBeginning();
            break;
        case 2:
            deleteEnd();
            break;
        case 3:
            printf("Enter the Node value to be deleted: ");
            scanf("%d",&location);
            deleteSpecific(location);
            break;
        default:

```

```

        printf("\nPlease select correct Deleting option!!!\n");
    }
}

int main()
{
    int ch;
    do
    {
        printf("\n***** MENU *****\n");
        printf("1. Insert\n2. Delete\n3. Display\n4. Exit\nEnter your choice: ");
        scanf("%d",&ch);
        switch(ch)
        {
            case 1:
                insert();
                break;
            case 2:
                delete();
                break;
            case 3:
                display();
                break;
            case 4:
                printf("Exiting...");
                break;
            default:
                printf("\nPlease select correct option!!!");
        }
    }while(ch!=4);
    return 0;
}

```

Output:

***** MENU *****

1. Insert
2. Delete
3. Display
4. Exit

Enter your choice: 1

Enter the value to be inserted: 10

Select from the following Inserting options

1. At Beginning
2. At End
3. After a Node

Enter your choice: 1

Insertion successful

***** MENU *****

1. Insert
2. Delete
3. Display
4. Exit

Enter your choice: 1

Enter the value to be inserted: 20

Select from the following Inserting options

1. At Beginning
2. At End
3. After a Node

Enter your choice: 2

Insertion success!!!

***** MENU *****

1. Insert
2. Delete
3. Display
4. Exit

Enter your choice: 1

Enter the value to be inserted: 30

Select from the following Inserting options

1. At Beginning
2. At End
3. After a Node

Enter your choice: 2

Insertion success!!!

***** MENU *****

1. Insert
2. Delete
3. Display
4. Exit

Enter your choice: 3

List elements are:

10 ---> 20 ---> 30 --->

***** MENU *****

1. Insert
2. Delete
3. Display
4. Exit

Enter your choice: 1

Enter the value to be inserted: 40

Select from the following Inserting options

1. At Beginning
2. At End
3. After a Node

Enter your choice: 3

Enter the location after which you want to insert: 20

Insertion success!!!

***** MENU *****

1. Insert
2. Delete
3. Display
4. Exit

Enter your choice: 3

List elements are:

10 ---> 20 ---> 40 ---> 30 --->

***** MENU *****

1. Insert
2. Delete
3. Display
4. Exit

Enter your choice: 2

Select from the following Deleting options

1. At Beginning
2. At End
3. Specific Node

Enter your choice: 3

Enter the Node value to be deleted: 20

Deletion success!!!

***** MENU *****

1. Insert
2. Delete
3. Display
4. Exit

Enter your choice: 3

List elements are:

10 ---> 40 ---> 30 --->

***** MENU *****

1. Insert
2. Delete
3. Display
4. Exit

Enter your choice: 2

Select from the following Deleting options

1. At Beginning
2. At End
3. Specific Node

Enter your choice: 1

Deletion success!!!

***** MENU *****

1. Insert
2. Delete
3. Display
4. Exit

Enter your choice: 3

List elements are:

40 ---> 30 --->

***** MENU *****

1. Insert
2. Delete
3. Display
4. Exit

Enter your choice: 2

Select from the following Deleting options

1. At Beginning

2. At End

3. Specific Node

Enter your choice: 2

Deletion success!!!

***** MENU *****

1. Insert
2. Delete
3. Display
4. Exit

Enter your choice: 3

List elements are:

40 --->

***** MENU *****

1. Insert
2. Delete
3. Display
4. Exit

Enter your choice: 2

Select from the following Deleting options

1. At Beginning
2. At End
3. Specific Node

Enter your choice: 1

Deletion success!!!

***** MENU *****

1. Insert
2. Delete
3. Display
4. Exit

Enter your choice: 3

List is Empty!!!

***** MENU *****

1. Insert
2. Delete
3. Display
4. Exit

Enter your choice: 4

Exiting....

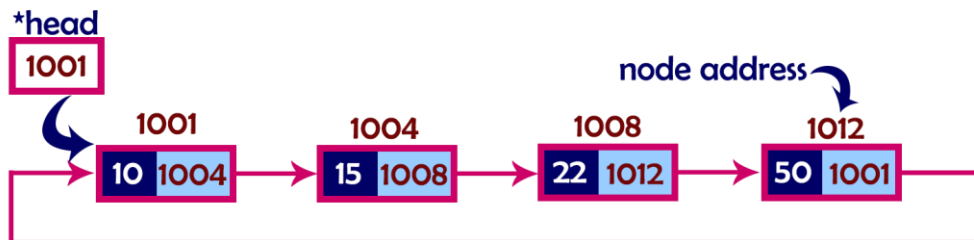
6. Circular linked lists and its operations

Q6.1. What is circular linked list?

Answer:

A circular linked list is a sequence of elements in which every element has a link to its next element in the sequence and the last element has a link to the first element.

Example:



Q.6.2. List the applications of circular linked list.

Answer:

1. Useful for implementation of queue. Unlike this implementation, we don't need to maintain two pointers for front and rear if we use circular linked list. We can maintain a pointer to the last inserted node and front can always be obtained as next of last.
2. Circular Doubly Linked Lists are used for implementation of advanced data structures like Fibonacci Heap.

Q6.3. Write a C program that uses functions to perform the following operations on circular linkedlist. i) Creation ii) Insertion iii) Deletion iv) Traversal

Answer:

```

Program:
#include<stdio.h>
#include<stdlib.h>
struct node
{
    int data;
    struct node *next;
}*head = NULL;
  
```

```

void insertAtBeginning()
{
    struct node *new,*temp;
    int item;
    new = (struct node *)malloc(sizeof(struct node));
    if(new == NULL)
    {
        printf("\nOVERFLOW");
    }
    else
    {
        printf("\nEnter the node data?");
        scanf("%d",&item);
        new -> data = item;
        if(head == NULL)
        {
            head = new;
            new -> next = head;
        }
        else
        {
            temp = head;
            while(temp->next != head)
                temp = temp->next;
            new->next = head;
            temp -> next = new;
            head = new;
        }
        printf("\nnode inserted\n");
    }
}

void insertAtEnd()
{
    struct node *new,*temp;
    int item;
    new = (struct node *)malloc(sizeof(struct node));
    if(new == NULL)
    {
        printf("\nOVERFLOW\n");
    }
    else
    {

```

```

        printf("\nEnter Data?");
        scanf("%d",&item);
        new->data = item;
        if(head == NULL)
        {
            head = new;
            new -> next = head;
        }
        else
        {
            temp = head;
            while(temp -> next != head)
            {
                temp = temp -> next;
            }
            temp -> next = new;
            new -> next = head;
        }

        printf("\nnode inserted\n");
    }
}

```

```

void removeBeginning()
{
    struct node *temp;
    if(head == NULL)
        printf("\nUNDERFLOW");
    else if(head->next == head)
    {
        head = NULL;
        free(head);
        printf("\nnode deleted\n");
    }

    else
    {
        temp = head;
        while(temp -> next != head)
            temp = temp -> next;
        temp->next = head->next;
        free(head);
    }
}

```

```

        head = temp->next;
        printf("\nnode deleted\n");
    }
}
void removeEnd()
{
    struct node *temp, *prev;
    if(head==NULL)
        printf("\nUNDERFLOW");
    else if (head ->next == head)
    {
        head = NULL;
        free(head);
        printf("\nnode deleted\n");
    }
    else
    {
        temp = head;
        while(temp ->next != head)
        {
            prev = temp;
            temp = temp->next;
        }
        prev->next = temp -> next;
        free(temp);
        printf("\nnode deleted\n");
    }
}

```

```

void display()
{
    struct node* temp;
    if (head == NULL)
        printf("\nList is empty\n");
    else
    {
        temp = head;
        do
        {
            printf("%d--->", temp->data);
            temp = temp->next;

```

```

        } while (temp !=head);
    }
}

void insert()
{
    int value, ch1;
    printf("Where you want to insert: \
\n1. At Beginning\n2. At End\n\
\nEnter your choice: ");
    scanf("%d",&ch1);
    switch(ch1)
    {
        case 1:
            insertAtBeginning(value);
            break;
        case 2:
            insertAtEnd(value);
            break;
        default:
            printf("\nWrong Input!! Try again!!!\n\n");
    }
}

void delete()
{
    int ch1;
    printf("How do you want to Delete: \n1. From Beginning\n\
2. From End\nEnter your choice: ");
    scanf("%d",&ch1);
    switch(ch1)
    {
        case 1:
            removeBeginning();
            break;
        case 2:
            removeEnd();
            break;
        default:
            printf("\nWrong Input!! Try again!!!\n\n");
    }
}

```

```

int main()
{
    int choice;
    do
    {
        printf("\n\n***** MENU *****\n1. Insert\n2. Display\n\
3. Delete\n4. Exit\nEnter your choice: ");
        scanf("%d",&choice);
        switch(choice)
        {
            case 1:
                insert();
                break;
            case 2:
                display();
                break;
            case 3:
                delete();
                break;
            case 4:
                printf("Exiting.....\n");
                break;
            default:
                printf("\nWrong input!!! Try again!!\n\n");
        }
    }while(choice != 4);
}

```

Output:

```

***** MENU *****
1. Insert
2. Display
3. Delete
4. Exit
Enter your choice: 1
Where you want to insert:
1. At Beginning
2. At End

Enter your choice: 1

```

Enter the node data?10

```

node inserted

***** MENU *****
1. Insert
2. Display
3. Delete
4. Exit
Enter your choice: 1
Where you want to insert:
1. At Beginning
2. At End

```

Enter your choice: 2

Enter Data?20

node inserted

***** MENU *****

1. Insert
2. Display
3. Delete
4. Exit

Enter your choice: 1

Where you want to insert:

1. At Beginning
2. At End

Enter your choice: 2

Enter Data?30

node inserted

***** MENU *****

1. Insert
2. Display
3. Delete
4. Exit

Enter your choice: 2

10--->20--->30---

***** MENU *****

1. Insert
2. Display
3. Delete
4. Exit

Enter your choice: 3

How do you want to Delete:

1. From Beginning
2. From End

Enter your choice: 1

node deleted

***** MENU *****

1. Insert
2. Display
3. Delete
4. Exit

Enter your choice: 2

20--->30---

***** MENU *****

1. Insert
2. Display
3. Delete
4. Exit

Enter your choice: 2

20--->30---

***** MENU *****

1. Insert
2. Display
3. Delete
4. Exit

Enter your choice: 3

How do you want to Delete:

1. From Beginning
2. From End

Enter your choice: 2

node deleted

***** MENU *****

1. Insert
2. Display
3. Delete
4. Exit

Enter your choice: 2

20--->

***** MENU *****

1. Insert
2. Display
3. Delete

4. Exit

Enter your choice: 3

How do you want to Delete:

1. From Beginning

2. From End

Enter your choice: 1

node deleted

***** MENU *****

1. Insert

2. Display

3. Delete

4. Exit

Enter your choice: 3

How do you want to Delete:

1. From Beginning

2. From End

Enter your choice: 1

UNDERFLOW

***** MENU *****

1. Insert

2. Display

3. Delete

4. Exit

Enter your choice: 2

List is empty

***** MENU *****

1. Insert

2. Display

3. Delete

4. Exit

Enter your choice: 4

Exiting.....

2 Marks Questions

1. Let P be a singly linked list and 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?
2. Differentiate singly linked list and doubly linked list.
3. List the applications of linked lists.
4. Differentiate array and linked list.
5. List the applications of linked lists.

Unit 5

1. Trees

- A. Tree terminology
- B. Binary Trees-Representation
- C. Binary tree traversals
- D. Binary tree operations

2. Graphs

- A. Graph terminology
- B. Graph representation
- C. Elementary graph operations
 - I. Breadth First Search (BFS)
 - II. Depth First Search (DFS)
 - III. Connected components
 - IV. Spanning trees

3. Searching and Sorting

- A. Sequential search
- B. Binary search
- C. Exchange (bubble) sort
- D. Selection sort
- E. Insertion sort.

1. Trees

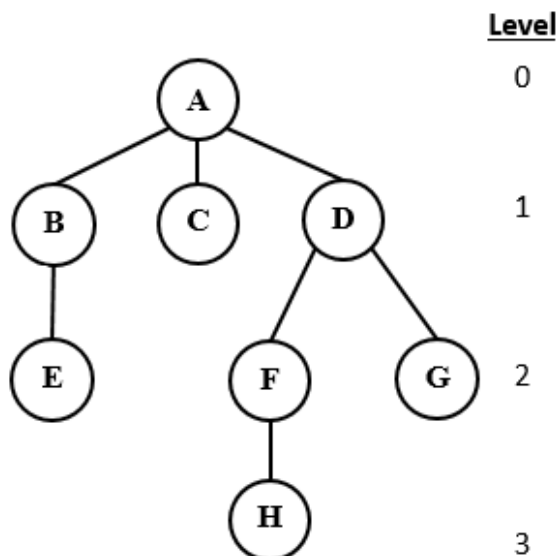
Tree Terminologies:

1. **Node:** Node of a tree stores the data and links to other node.
2. **Parent:** The parent of a node is immediate predecessor of a node.
3. **Child:** Immediate successor of a node is known as Child.
4. **Link:** It is a pointer to a node in a tree.
5. **Root:** A node that does not have any parent is called root node.
6. **Leaf:** A node that does not have any child is called leaf node.
7. **Level:** Level is the rank of hierarchy. The root node has level 0.

If a node is at Level X, then its parent is at level X-1 and its child is at level X+1.

8. **Height:** Maximum nodes in a path starting from root node to leaf node is called height of a tree
9. **Degree:** The maximum number of children for a node is called as degree of a node.
10. **Siblings:** The nodes which have the same parent are called siblings.

Example:

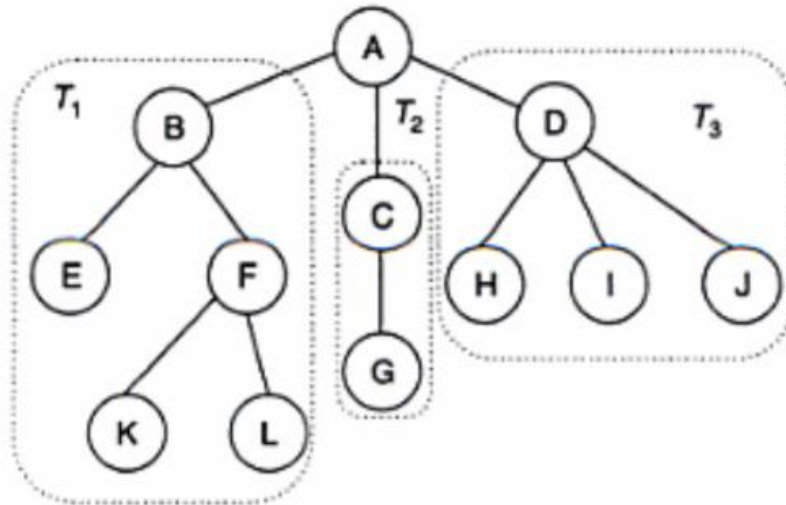


In this example:

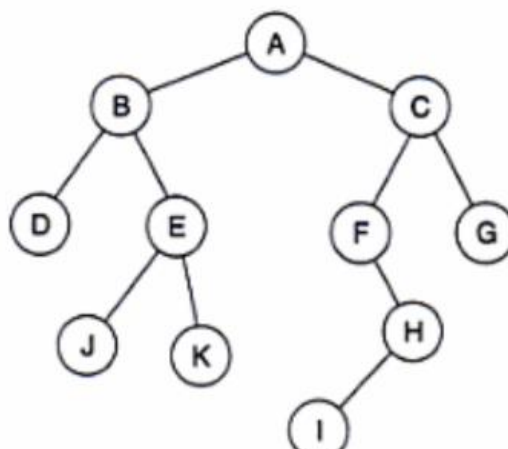
- Node A is root node,
- Nodes C, E, F, G, H are leaf nodes
- Degree of node A is 3
- Node D is parent for nodes F, G
- Node H is child for F
- Height of the tree is 4

Q1.1. Define a Tree.**Answer:**

- A tree is a finite set of one or more nodes such that:
 1. There is a special node called the root
 2. The remaining nodes are partitioned into n ($n > 0$) disjoint sets T_1, T_2, \dots, T_n . Each T_i ($i = 1, 2, \dots, n$) is a tree; T_1, T_2, \dots, T_n are called subtrees of root.

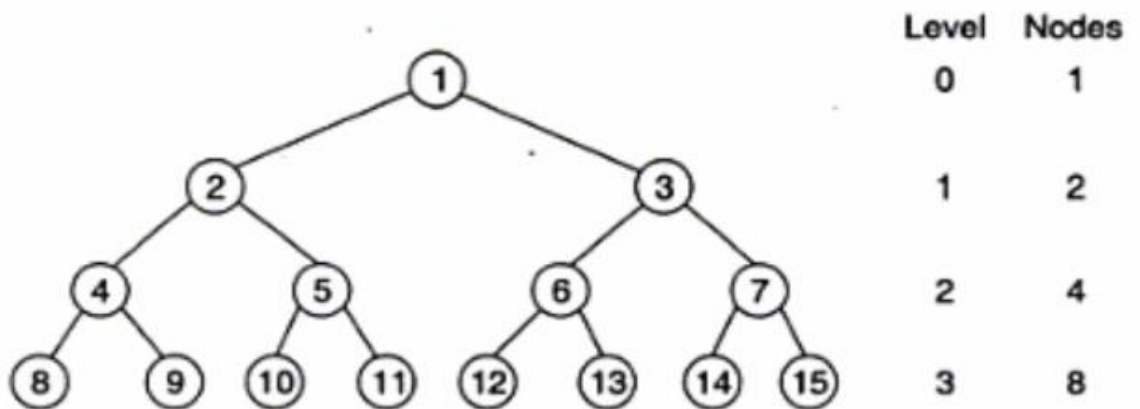
Example:**Q1.2. Define a binary tree.****Answer:**

- A Binary Tree 'T' is a finite set of nodes, such that,
 1. T is empty, or
 2. Every node can have maximum of 2 child nodes. One child is called left child and other is called right child.

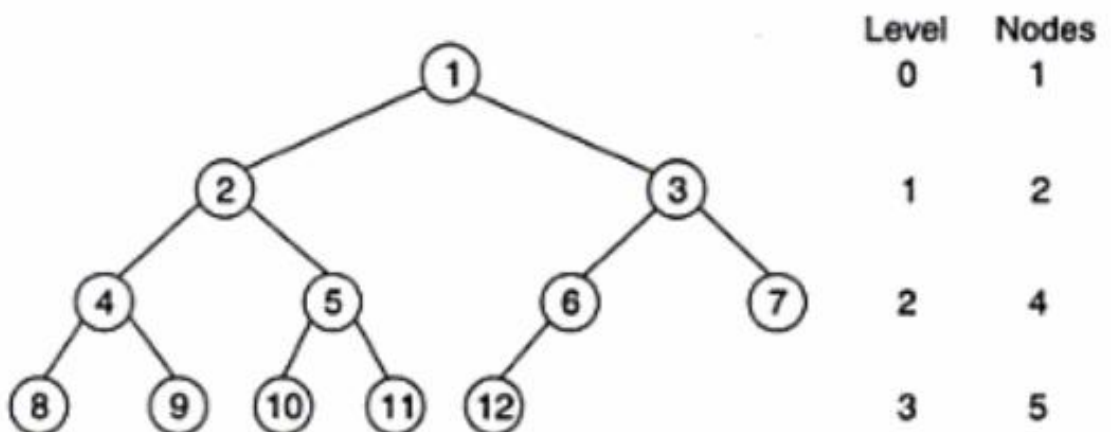
Example:

Q1.3. Define full binary tree.**Answer:**

A binary tree is a full binary if it contains the maximum number of possible nodes at all levels.

Example:**Q1.4. Define complete binary tree.****Answer:**

A binary tree is a complete binary if all its levels, except last level, contains the maximum number of possible nodes, and all the nodes in the last level appear as left as possible.

Example:

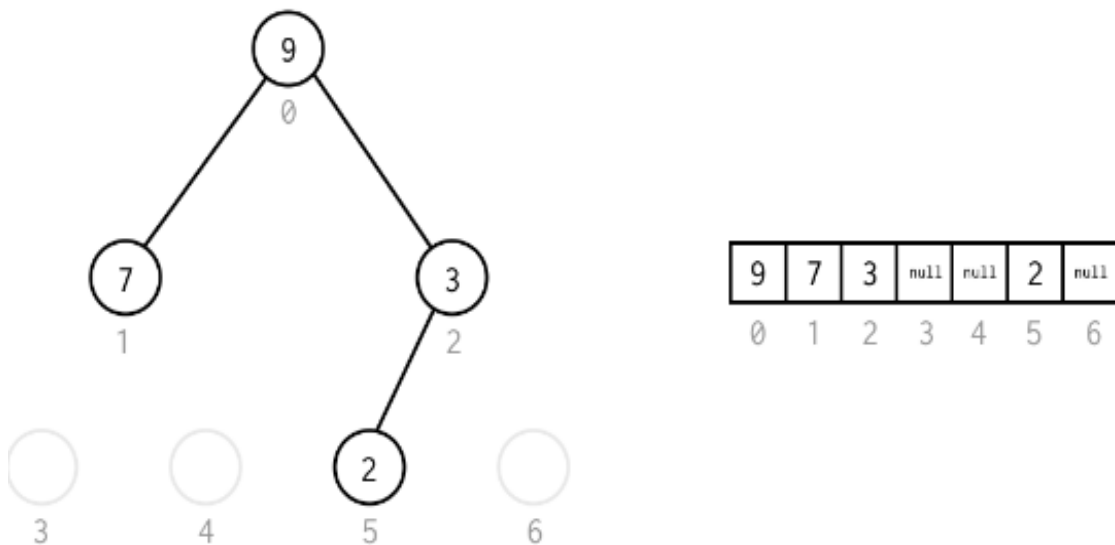
Q1.5. How to represent a binary tree? Explain?**Answer:**

A binary tree can be represented in two methods. They are:

1. Linear representation
2. Linked representation

1. Linear Representation:

- In this, One- Dimensional Array is used to represent binary tree.
- The root node is placed at index 0 (zero).
- For any node placed at location 'i', then
 - Left child of it is placed at $(2 * i + 1)$
 - Right child of it is placed at $(2 * i + 2)$

Example:**Advantages of Linear Representation:**

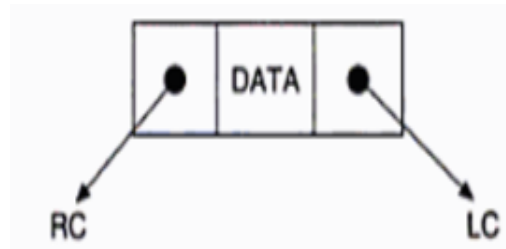
1. Any node can be accessed from any other node by calculating the index.
2. Only data is stored without pointers.

Disadvantages of Linear Representation:

1. Other than full binary trees, the majority of the array entries may be empty.
2. It allows only static representation.
3. Inserting or deleting a node in a tree is inefficient.

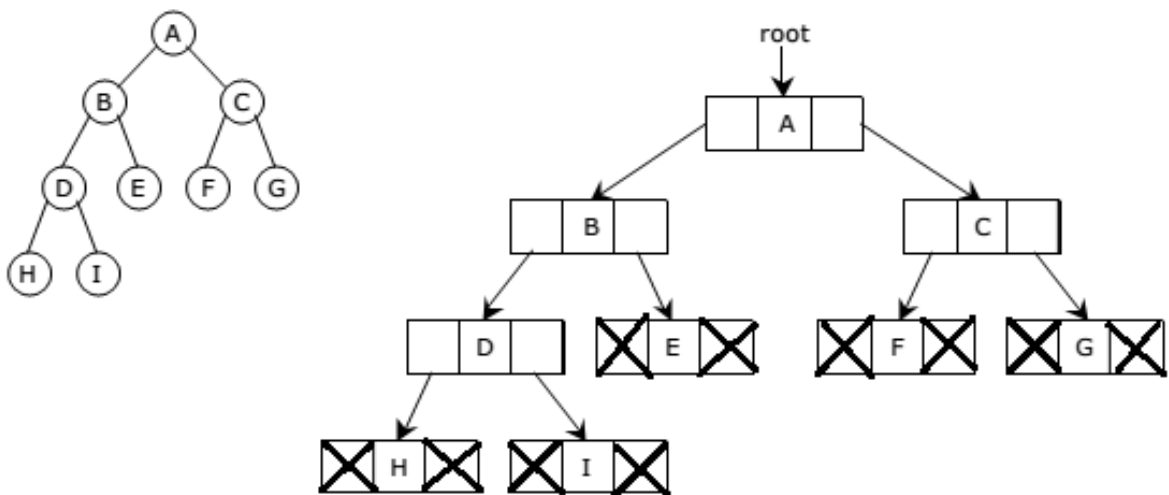
2. Linked Representation:

- In this. Double linked list is used.
- Structure of a node is shown in below figure.



- Here, LC and RC are the two link fields used to store the addresses of left child and right child of a node; DATA is the actual information of the node.

Example



Q1.6. List the properties of binary tree.

Answer:

1. In binary tree, the maximum number of nodes on level 'i' is 2^i , where $i \geq 0$.
2. The maximum number of nodes in a binary tree of height 'h' is $2^h - 1$, where $h \geq 1$.
3. The minimum number of nodes in a binary tree of height 'h' is h, where $h \geq 1$.
4. For any non-empty binary tree, if 'n' is the number of nodes and 'e' is the number of edges, then $n = e + 1$ (or) $e = n - 1$.
5. For any non-empty binary tree, if 'n' is the number of leaf nodes and 'm' is the number of internal nodes, then $n = m + 1$.
6. The height of complete binary tree with 'n' number of nodes is $\log_2(n + 1)$.
7. The total number of binary trees possible with 'n' nodes is $\frac{1}{n+1} \binom{2n}{n}$.

8. The maximum size that an array require to store a binary tree with 'n' nodes is

$$Size_{max} = 2^n - 1.$$

9. The minimum size that an array require to store a binary tree with 'n' nodes is

$$Size_{min} = 2^{\log_2(n+1)} - 1.$$

10. In linked representation of a binary tree, if there are 'n' number of nodes then the number of null links = $n + 1$.

Q1.7. Explain in detail about binary tree traversals.

Answer:

- Traversing is nothing but visiting each node in the tree exactly once.
- There are 4 major traversing techniques.

Depth First Traversals:

1. Pre-order Traversal
2. In-order Traversal
3. Post-order Traversal
4. Level order Traversal

- The pre-order, in-order, post-order traversals are known as Depth First Traversals.
- The level order traversal is known as Breath First Traversal.

1. Pre-Order Traversal:

This traversal can be defined as follows:

- a. Visit the root node R.
- b. Traverse the left subtree of R in Pre-order.
- c. Traverse the right subtree of R in Pre-order.

Recursive Algorithm:

Preorder(root)

1. ptr = root
2. if (ptr != NULL) then
3. visit(ptr)
4. Preorder(ptr → LC)
5. Preorder(ptr → RC)
6. End if

2. In-Order Traversal:

This traversal can be defined as follows:

- a. Traverse the left subtree of R in In-order.
- b. Visit the root node R.
- c. Traverse the right subtree of R in In-order.

Recursive Algorithm:

Inorder(root)

1. ptr = root
2. if (ptr != NULL) then
3. Inorder(ptr → LC)
4. visit(ptr)
5. Inorder(ptr → RC)
6. End if

3. Post-order Traversal:

This traversal can be defined as follows:

- a. Traverse the left subtree of R in Post-order.
- b. Traverse the right subtree of R in Post-order.
- c. Visit the root node R.

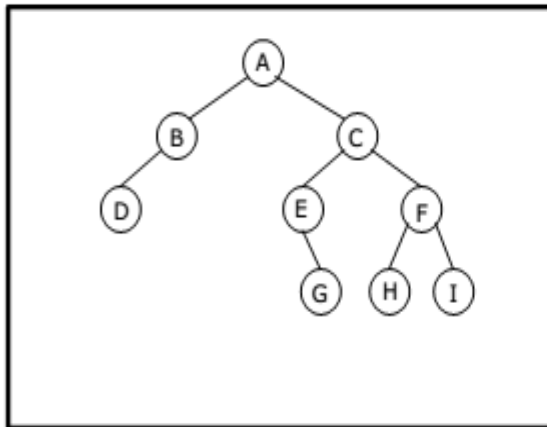
Recursive Algorithm:

Postorder(root)

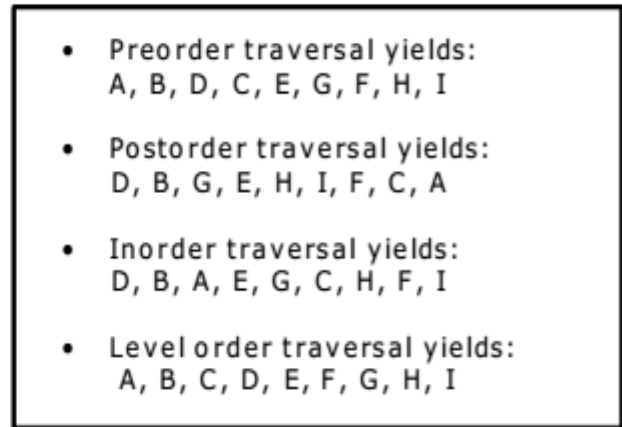
1. ptr = root
2. if (ptr != NULL) then
3. Postorder(ptr → LC)
4. Postorder(ptr → RC)
5. visit(ptr)
6. End if

4. Level Order Traversal:

- In this, the nodes are visited level by level starting from root, and going from left to right.
- The level order traversal requires queue data structure. So, it is not possible to develop recursive algorithm to traverse the binary tree in level order.

Example:

Binary Tree



Pre, Post, Inorder and level order Traversing

Q1.8. Construct Binary Tree from given In-order and Pre-order traversals:

In-order: { 4, 2, 1, 7, 5, 8, 3, 6 }

Pre-order: { 1, 2, 4, 3, 5, 7, 8, 6 }

Answer:

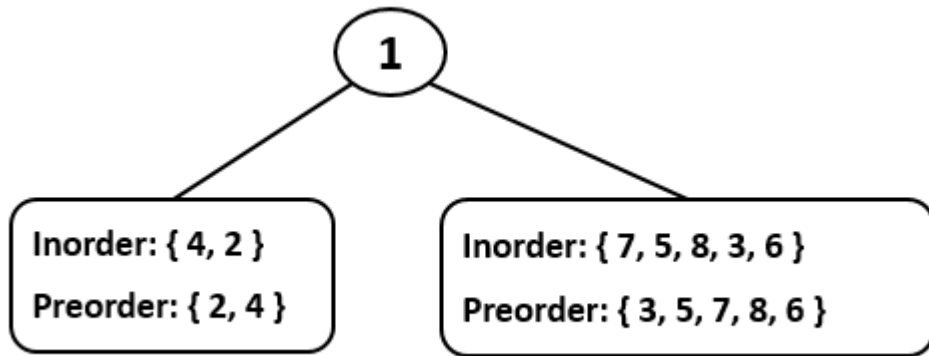
- Given Data,

In-order: { 4, 2, 1, 7, 5, 8, 3, 6 }

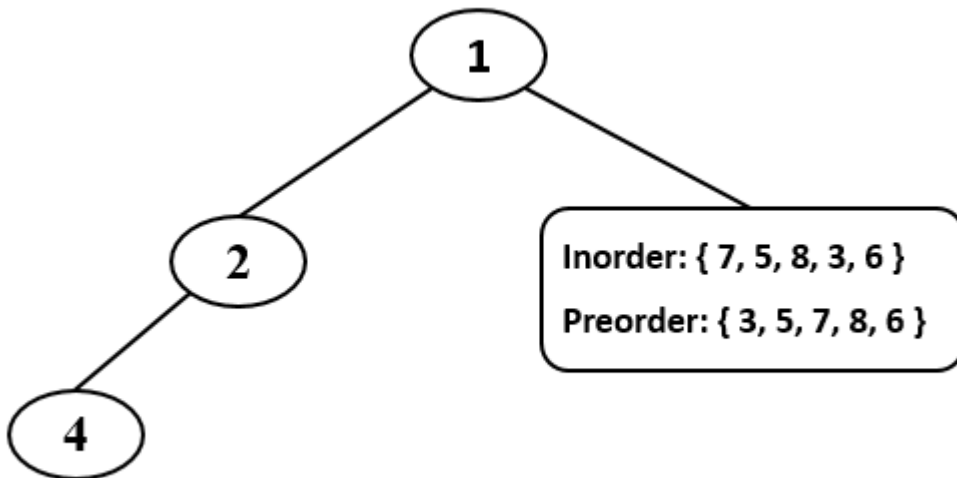
Pre-order: { 1, 2, 4, 3, 5, 7, 8, 6 }

- In a Pre-order sequence, leftmost element is the root of the tree. So we know '1' is root for given sequences.
- By searching '1' in In-order sequence, we can find out all elements on left side of '1' are in left subtree and elements on right are in right subtree.
- Then steps of constructing binary tree as follows:

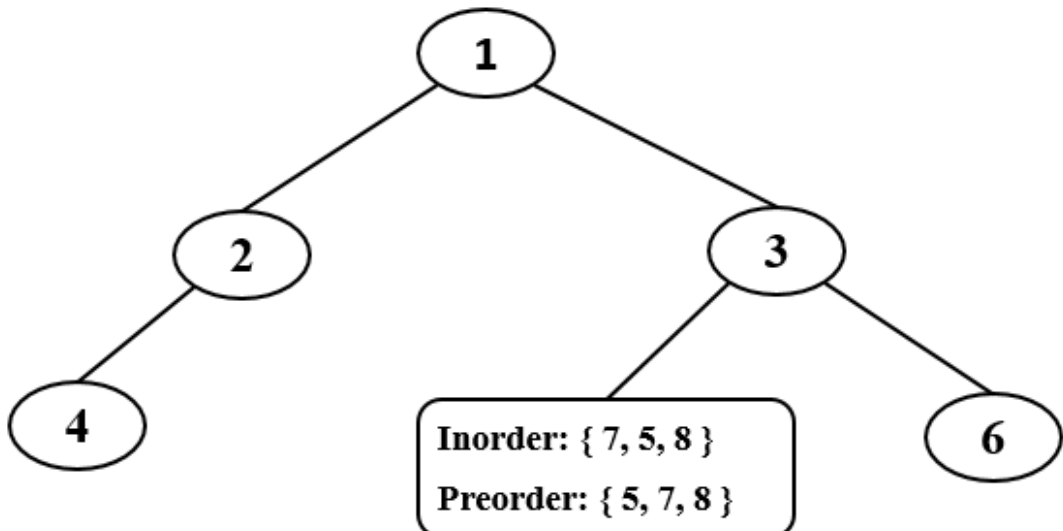
Step 1:

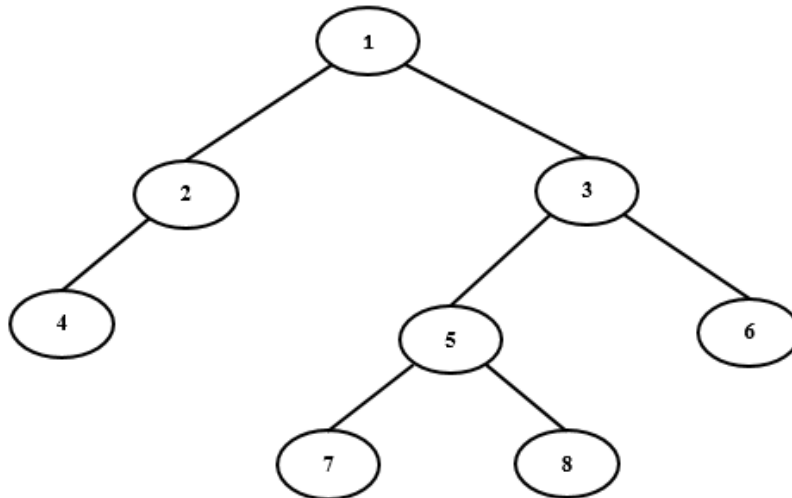


Step 2:



Step 3:



Step 4:

Q1.9. Construct Binary Tree from given In-order and Post-order traversals:

Inorder : { 4, 2, 1, 7, 5, 8, 3, 6 }

Postorder : { 4, 2, 7, 8, 5, 6, 3, 1 }

Answer:

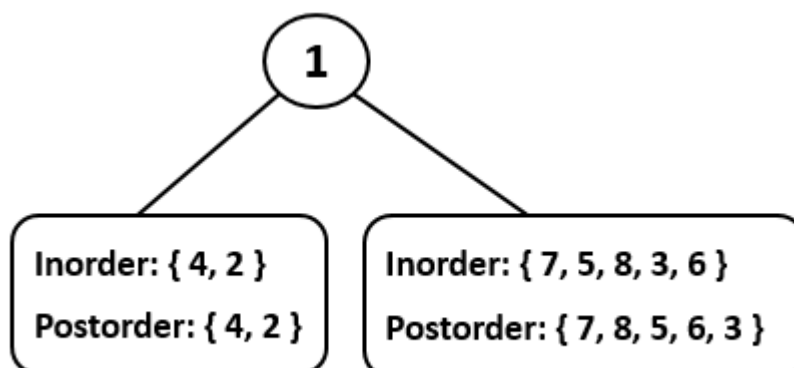
➤ Given Data,

In-order: { 4, 2, 1, 7, 5, 8, 3, 6 }

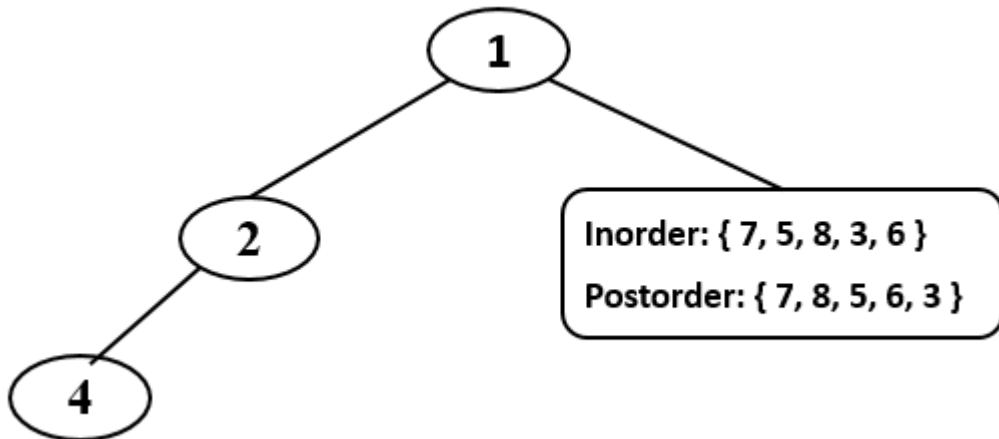
Post-order: { 4, 2, 7, 8, 5, 6, 3, 1 }

- In a Post-order sequence, rightmost element is the root of the tree. So we know '1' is root for given sequences.
- By searching '1' in In-order sequence, we can find out all elements on left side of '1' are in left subtree and elements on right are in right subtree.
- Then steps of constructing binary tree as follows:

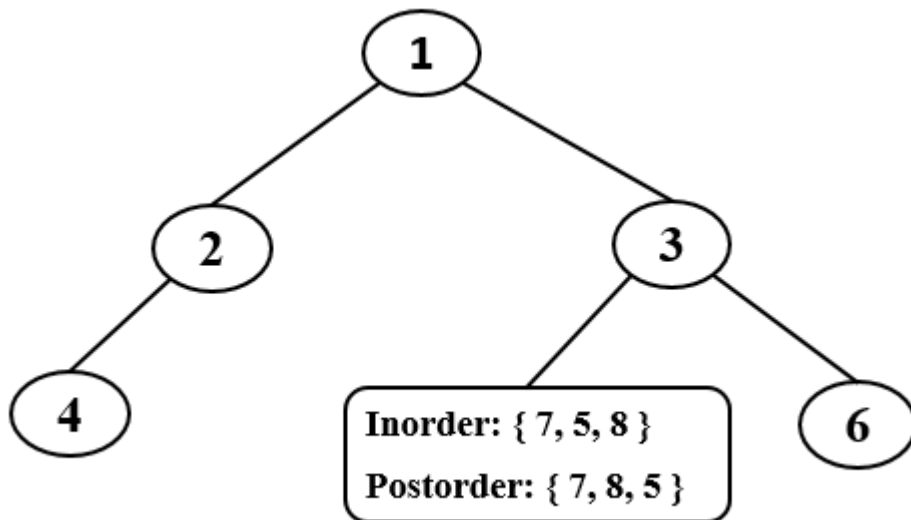
Step 1:



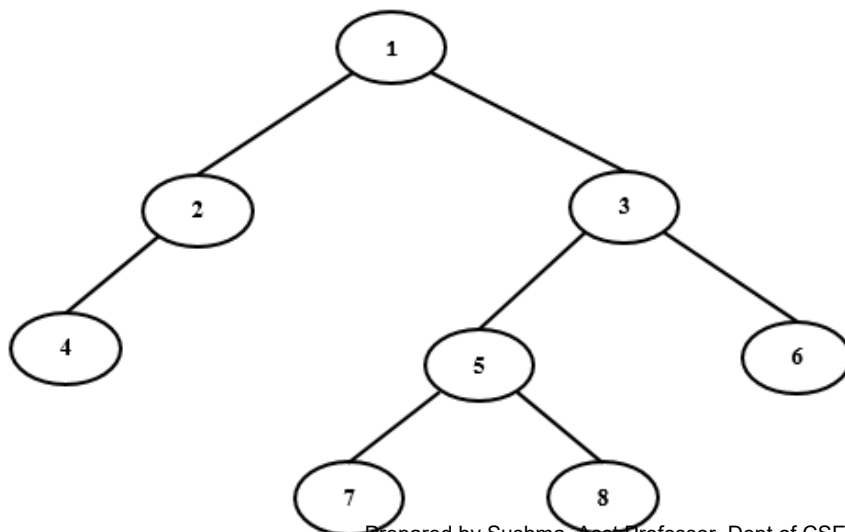
Step 2:



Step 3:



Step 4:



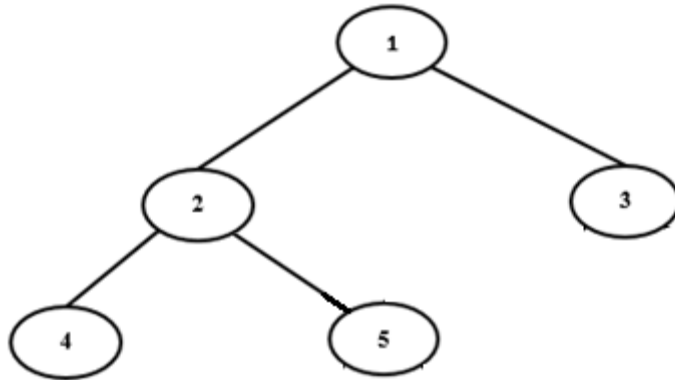
Q1.10. Write a C program that uses functions to perform the following:

i) Creating a Binary Tree of integers

ii) Traversing the above binary tree in pre-order, in-order and post-order.

Answer:

Let consider the below Binary Tree to print pre-order, in-order and post-order



Program:

```

#include <stdio.h>
#include <stdlib.h>
/* Structure for a node */
struct node
{
    int data;
    struct node *left;
    struct node *right;
};

/* Function to create a new node */
struct node *newNode(int data)
{
    struct node *temp = (struct node *) malloc(sizeof(struct node));
    temp -> data = data;
    temp -> left = NULL;
    temp -> right = NULL;
    return temp;
};
  
```

```
/* Function to print the inorder traversal of the tree */
void inorder(struct node *root)
{
    if(root == NULL)
        return;
    inorder(root -> left);
    printf( "%d ", root -> data);
    inorder(root -> right);
}

/* Function to print the preorder traversal of the tree */
void preorder(struct node *root)
{
    if(root == NULL)
        return;
    printf( "%d ", root -> data);
    preorder(root -> left);
    preorder(root -> right);
}

/* Function to print the postorder traversal of the tree */
void postorder(struct node *root)
{
    if(root == NULL)
        return;
    postorder(root -> left);
    postorder(root -> right);
    printf( "%d ", root -> data);
}
```



```
/* Main Function */  
int main()  
{  
    struct node *root = newNode(1);  
    root->left = newNode(2);  
    root->right = newNode(3);  
    root->left->left = newNode(4);  
    root->left->right = newNode(5);  
  
    printf("\nInorder Traversal : ");  
    inorder(root);  
    printf("\nPreorder Traversal : ");  
    preorder(root);  
    printf("\nPostorder Traversal : ");  
    postorder(root);  
    return 0;  
}
```

Output:

Inorder Traversal : 4 2 5 1 3

Preorder Traversal : 1 2 4 5 3

Postorder Traversal : 4 5 2 3 1

2. Graphs

Q2.1. Define a Graph. Give example?

Answer:

Graph:

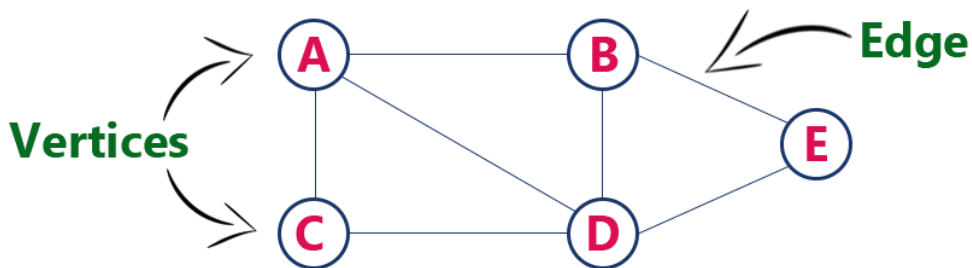
- A Graph is a non-linear data structure.
- A Graph 'G' is a set of vertices and edges in which nodes are connected with edges.
- A Graph 'G' is represented as

$$G = (V, E)$$

where **V** is set of vertices &

E is set of edges.

Example:



- The above graph has 5 vertices and 6 edges.
- This graph G can be defined as $G = (V, E)$
- Where $V = \{A, B, C, D, E\}$ &

$$E = \{(A, B), (A, C), (A, D), (B, D), (C, D), (B, E), (E, D)\}.$$

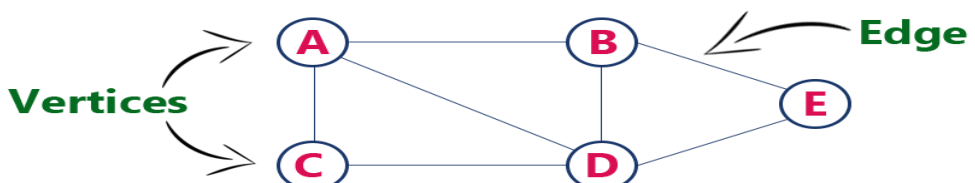
Note: A graph may have cycles where as a tree cannot. For example, in the above graph $A \rightarrow B \rightarrow D \rightarrow C \rightarrow A$ forms a cycle.

Q2.2. Define a vertex?

Vertex:

- Individual data element of a graph is called as Vertex.
- Vertex is also known as node.

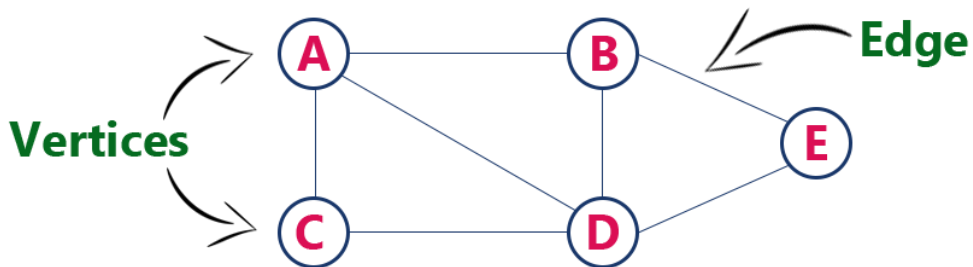
Example:



In the above graph A, B, C, D, E are the vertices.

Q2.3. Define an edge?**Edge:**

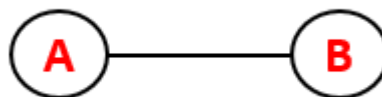
- An edge is a connecting link between two vertices.
- Edge is also known as Arc.
- An edge is represented as (starting_Vertex, ending_Vertex).

Example:

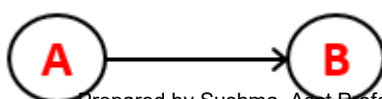
In above graph the link between vertices A and B is represented as (A, B). In above example graph, there are 7 edges (i.e., (A, B), (A, C), (A, D), (B, D), (B, E), (C, D), (D, E)).

Q2.4. Explain different types of edges of a graph?**Answer:****1. Undirected Edge:**

- An undirected edge is a bidirectional edge. If there is undirected edge between vertices A and B then edge (A, B) is equal to edge (B, A).

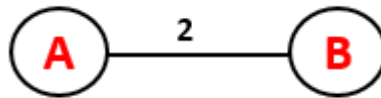
Example:**2. Directed Edge:**

- A directed edge is a unidirectional edge. If there is directed edge between vertices A and B then edge (A, B) is not equal to edge (B, A).

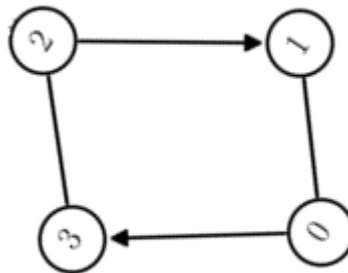
Example:

3. Weighted Edge:

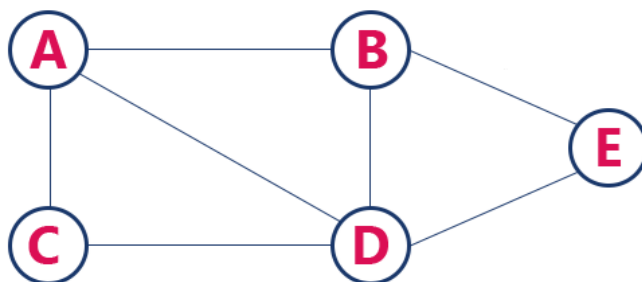
- A weighted edge is an edge with value (cost) on it.

Example:**Q2.5. Define a Mixed Graph?****Answer:****Mixed Graph:**

A graph with both undirected and directed edges is said to be mixed graph.

Example:**Q2.6. Define end vertices or Endpoints****Answer:****End vertices or Endpoints:**

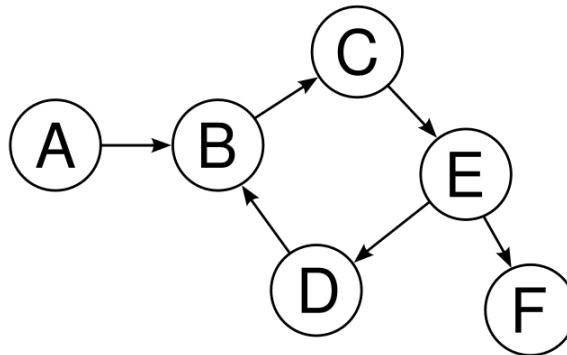
The two vertices joined by edge are called end vertices (or endpoints) of that edge.

Example:

In this graph, A and B are the end points for the edge (A,B)

Q2.7. Define origin and destination of an edge?**Answer:****Origin and Destination:**

If an edge is directed, its first endpoint is said to be the origin of it and the other endpoint is said to be the destination of that edge.

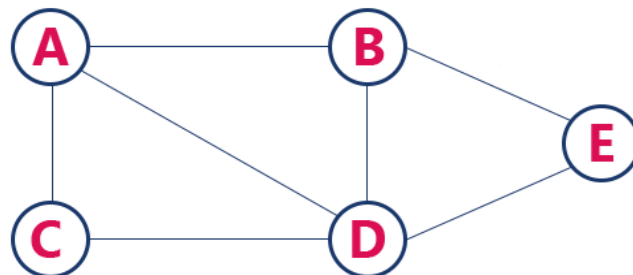
Example:

In this graph,

For the edge (A,B), A is the origin and B is the destination.

Q2.8. Define Adjacent vertices?**Answer:****Adjacent Vertices:**

Vertices A and B are said to be adjacent if there is an edge between them.

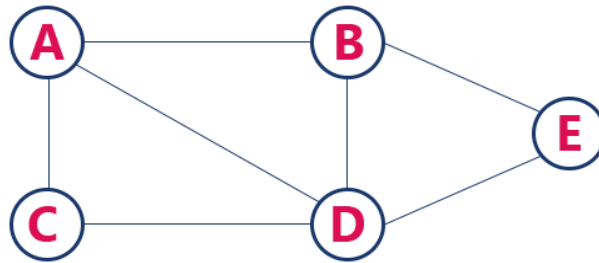
Example:

In this graph,

- B, C, D are adjacent to A
- A, D, E are adjacent to B
- A, D are adjacent to C
- A, B, C, E are adjacent to D
- B, D are adjacent to E

Q2.9. Define incident of a vertex.**Answer:****Incident:**

Edge is said to be incident on a vertex if the vertex is one of the endpoints of that edge.

Example:

In this graph,

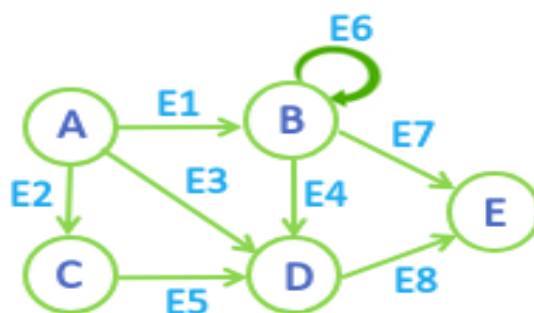
- A and B are incidents for edge (A, B).

Q2.10. Define incoming edge and outgoing edge.**Answer:****Incoming Edge:**

A directed edge is said to be incoming edge on its destination vertex.

Outgoing Edge:

A directed edge is said to be outgoing edge on its origin vertex.

Example:

Here **E1** is incoming edge for **B** &
E1 is outgoing edge for **A**

Q2.11. Define degree, in-degree and out-degree.**Answer:****Degree:**

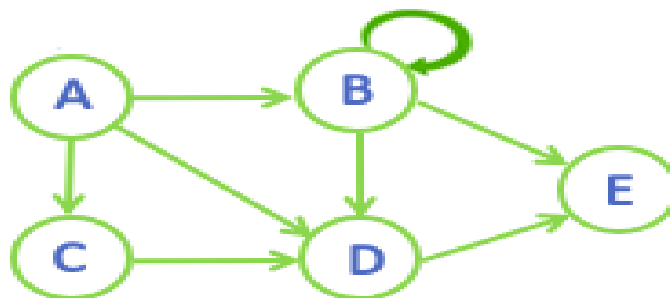
Total number of edges connected to a vertex is said to be degree of that vertex.

In-Degree:

Total number of incoming edges connected to a vertex is said to be in-degree of that vertex.

Out-Degree:

Total number of outgoing edges connected to a vertex is said to be out-degree of that vertex.

Example:

In-Degree of D is 3

Out-Degree of D is 1

Degree of D is 4

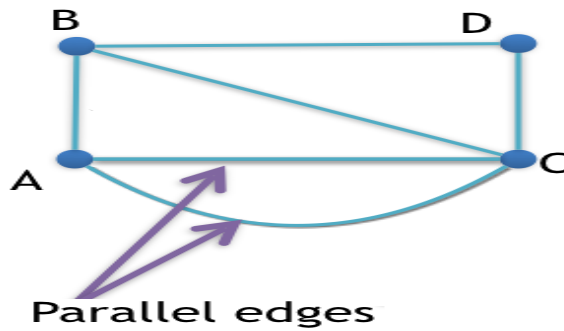
Q2.12. Define parallel edges?**Answer:****Parallel Edges (Multiple Edges):**

In undirected graph, if there are two undirected edges with same end vertices are called parallel edges or multiple edges.

(Or)

In directed graph two directed edges with same origin and destination, such edges are called parallel edges or multiple edges.

Example:



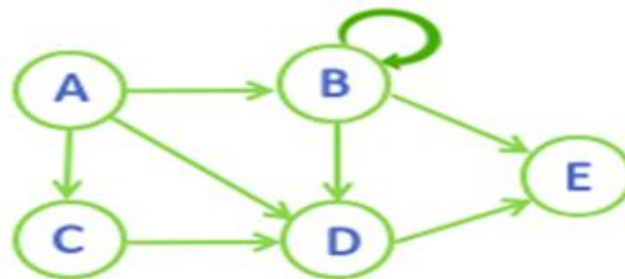
Q2.13. Define Self Loop?

Answer:

Self-Loop:

A directed or undirected edge is a self-loop if both of its endpoints are same. i.e., source equal to destination.

Example:



Here B has a self loop

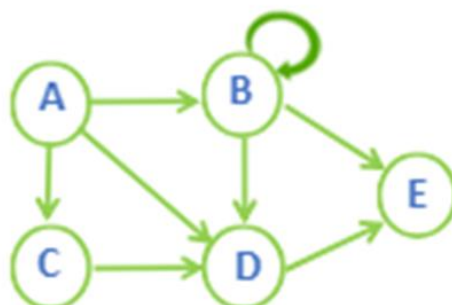
Q2.14. Define path and length of a path.

Answer:

Path:

Path represents a sequence of edges between the two vertices.

Example:



path from A to E is A-C-D-E, A-D-E, A-B-E, A-B-D-E

Length of a path:

The number of edges in a path is called the length of that path.

In the above graph, the length of the path A-C-D-E is 3.

Q2.15. Define simple graph?

Answer:

Simple Graph:

A graph is said to be simple if there are no parallel and self-loop edges.

Q2.16. Define sub-graph?

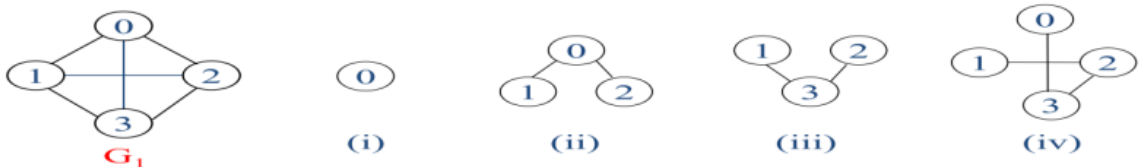
Answer:

Sub-Graph:

A graph S is said to be a sub graph of a graph G if all the vertices and all the edges of S are in G, and each edge of S has the same end vertices in S as in G.

A subgraph of G is a graph G' such that

$$V(G') \subseteq V(G) \text{ and } E(G') \subseteq E(G)$$

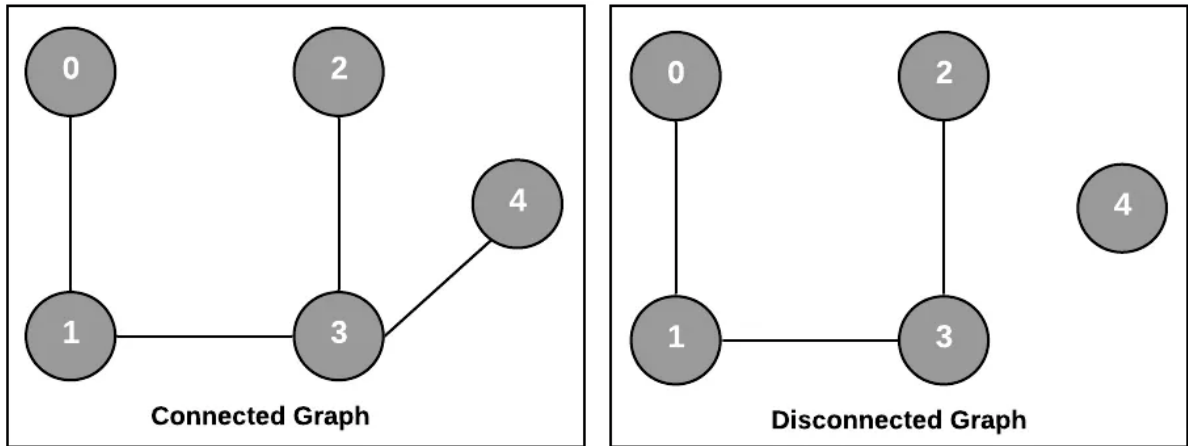
Example:**Q2.17. Define Connected and Disconnected Graphs?**

Answer:

Connected and Disconnected Graphs:

A graph G is said to be connected if there is at least one path between every pair of vertices in G. Otherwise G is disconnected or disjoint graph.

Example:



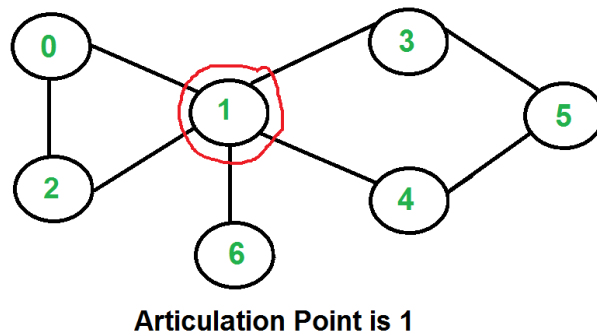
Q2.18. Define Articulation Point?

Answer:

Articulation Point:

If a vertex is removed from a graph which results in disjoint graphs, then this vertex is called Articulation point of that graph.

Example:



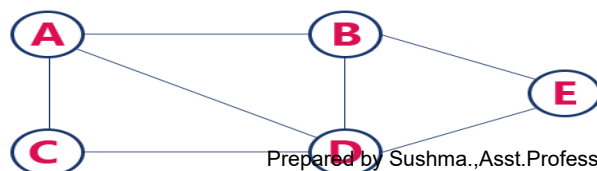
Q2.19. Define directed and undirected graphs?

Answer:

Directed Graph:

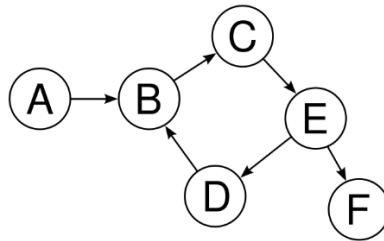
A graph with only undirected edges is said to be undirected graph.

Example:



Undirected Graph:

A graph with only directed edges is said to be directed graph.

Example:**Q2.20. Define a complete graph?**

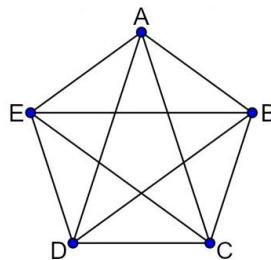
Answer:

Complete Graph:

A graph in which any 'V' vertices is adjacent to all other vertices present in the graph is known as a complete graph.

The number of edges in undirected complete graph is: $n(n-1)/2$

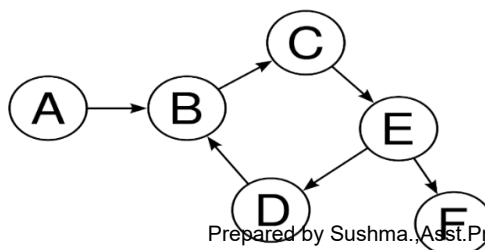
where n is the number of vertices

Example:**Q2.21. Define cyclic graph?**

Answer:

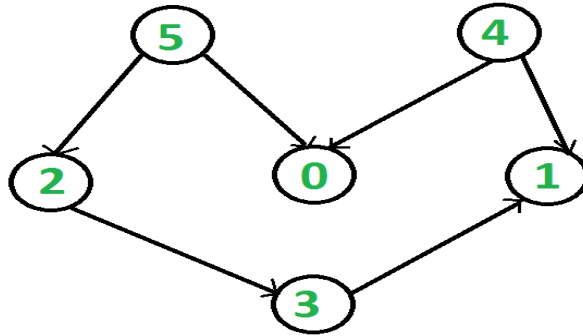
Cyclic Graph:

A graph having cycle is called cycle graph.

Example:

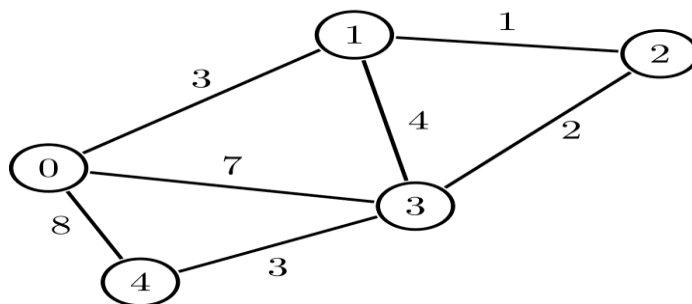
Q2.22. Define acyclic graph?**Answer:****Acyclic Graph:**

A graph without cycle is called acyclic graphs.

Example:**Q2.23. Define weighted graph?****Answer:****Weighted Graph:**

A graph is said to be weighted if there are some non-negative value assigned to each edges of the graph.

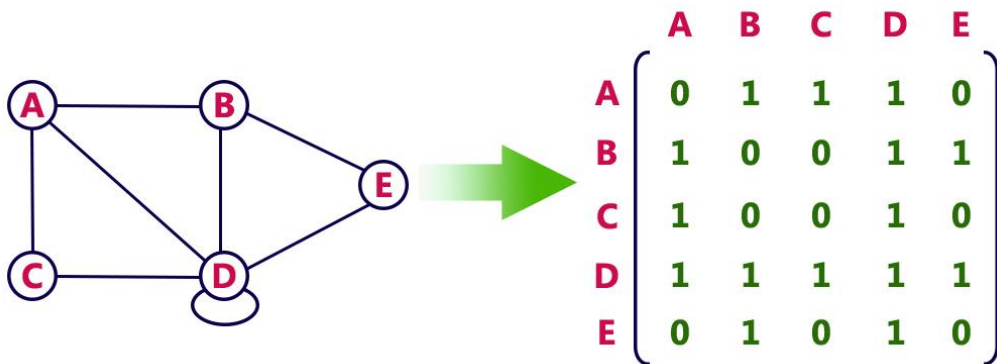
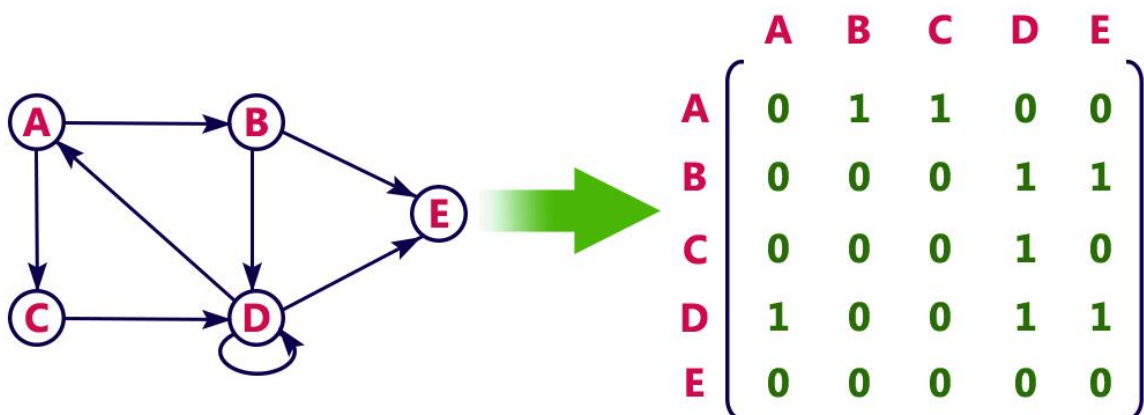
The value is equal to the length between two vertices.

Example:**Q2.24. List and explain the techniques used to represent a graph with example.****Answer:**

- Graph data structure is represented using following representations:
 1. Adjacency Matrix
 2. Incidence Matrix
 3. Adjacency List

1. Adjacency Matrix:

- In this representation, the graph is represented using a matrix of size $N \times N$, where N total number of vertices.
- That means a graph with 4 vertices is represented using a matrix of size 4×4 .
- In this matrix, both rows and columns represent vertices.
- This matrix is filled with either 1 or 0.
 - Here, 1 represents that there is an edge from row vertex to column vertex and 0 represents that there is no edge from row vertex to column vertex.

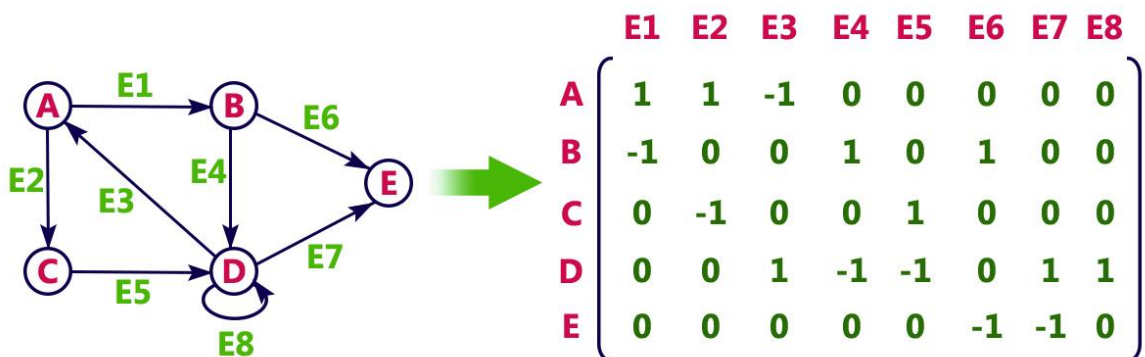
Example:**1. Undirected Graph Representation:****2. Directed Graph Representation:**

2. Incidence Matrix:

- In this representation, the graph is represented using a matrix of size $N \times M$.
Where N is the total number of vertices &
M is the total number of edges.
- That means graph with 4 vertices and 6 edges is represented using a matrix of size 4X6.
- In this matrix, rows represent vertices and columns represents edges.
- This matrix is filled with 0 or 1 or -1.
 - Here, 0 represents that the row vertex is not connected to column edge,
 - 1 represents that the row vertex is connected as the outgoing edge to column edge,
 - 1 represents that the row vertex is connected as the incoming edge to column edge.

Example:

1. Undirected Graph Representation:

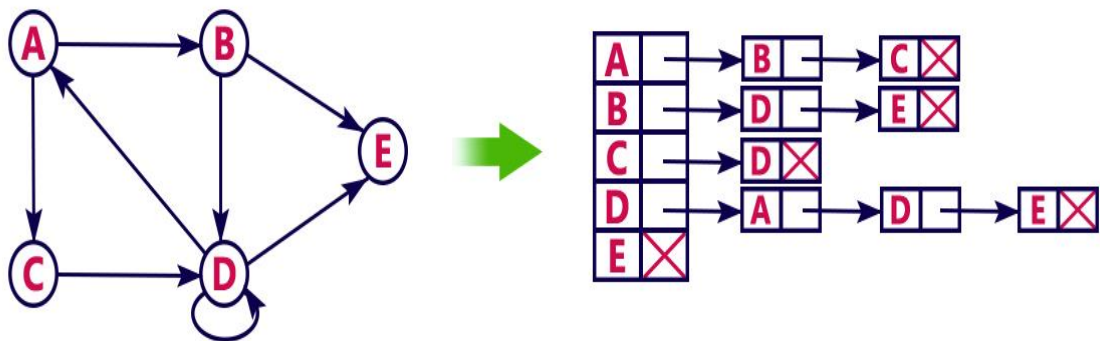


2. Directed Graph Representation:



3. Adjacency List:

- In this representation, every vertex of a graph contains list of its adjacent vertices.
- For example, consider the following directed graph representation implemented using linked list:

Example:

Q2.25. Define Graph ADT.

Answer:

Graph ADT:

Definition:

- Graph is a set of vertices and edges that are connected to the vertices.

Operations:

for all graph is a Graph, v, v1 and v2 Vertices

Graph Create() → return an empty graph

Graph InsertVertex(graph, v) → return a graph with v inserted. v has no edge.

Graph InsertEdge(graph, v1,v2) → return a graph with new edge between v1 and v2 Graph

DeleteVertex(graph, v) → return a graph in which v and all edges incident to it are removed

Graph DeleteEdge(graph, v1, v2) → return a graph in which the edge (v1, v2) is removed

Boolean IsEmpty(graph) → if (graph==empty graph) return TRUE else return FALSE

List Adjacent(graph,v) → return a list of all vertices that are adjacent to v

Q2.26. What is mean by graph traversal. List out graph traversal technique.

Answer:

- Graph traversal is a technique used for searching a vertex in a graph.
- Using graph traversal, we visit all the vertices of the graph without getting into looping path.
- There are two graph traversal techniques and they are as follows...
 1. DFS (Depth First Search)
 2. BFS (Breadth First Search)

Q2.27. Explain in detail about DFS (Depth First Search) traversal technique?

Answer:

DFS (Depth First Search):

- DFS traversal of a graph produces a **spanning tree** as final result.
- **Spanning Tree** is a graph without loops.
- We use **Stack data structure** with maximum size of total number of vertices in the graph to implement DFS traversal.
- We use the following steps to implement DFS traversal:

Step 1 - Define a Stack of size total number of vertices in the graph.

Step 2 - Select any vertex as **starting point** for traversal. Visit that vertex and push it on to the Stack.

Step 3 - **Visit any one** of the **non-visited adjacent** vertices of a vertex which is at the top of stack and push it on to the stack.

Step 4 - Repeat step 3 until there is no new vertex to be visited from the vertex which is at the top of the stack.

Step 5 - When there is no new vertex to visit then use **back tracking** and pop one vertex from the stack.

Step 6 - Repeat steps 3, 4 and 5 until stack becomes Empty.

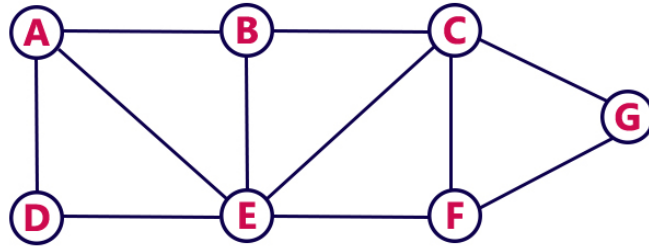
Step 7 - When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph

Note:

Back tracking is coming back to the vertex from which we reached the current vertex.

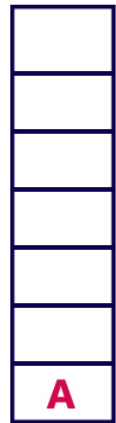
Example:

Consider the following example graph to perform DFS traversal



Step 1:

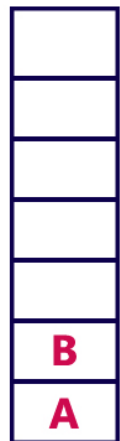
- Select the vertex **A** as starting point (visit **A**).
- Push **A** on to the Stack.



Stack

Step 2:

- Visit any adjacent vertex of **A** which is not visited (**B**).
- Push newly visited vertex B on to the Stack.



Stack

Step 3:

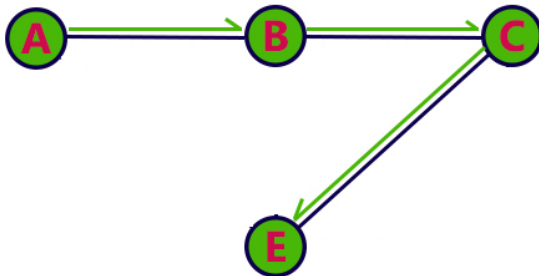
- Visit any adjacent vertex of **B** which is not visited (**C**).
- Push C on to the Stack.



Stack

Step 4:

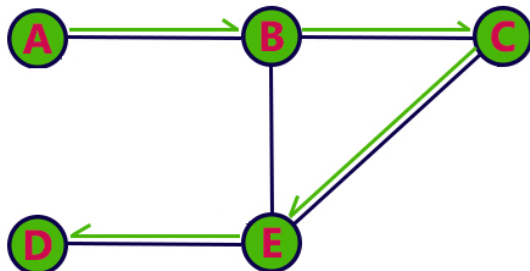
- Visit any adjacent vertex of **C** which is not visited (**E**).
- Push E on to the Stack



Stack

Step 5:

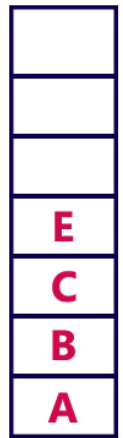
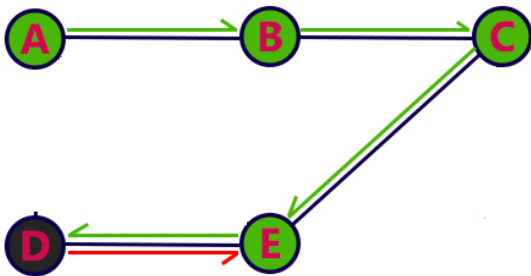
- Visit any adjacent vertex of **E** which is not visited (**D**).
- Push D on to the Stack



Stack

Step 6:

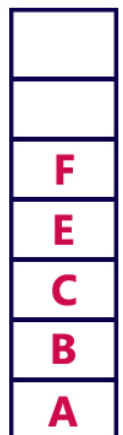
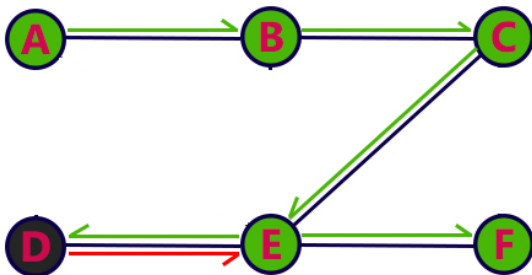
- There is no new vertex to be visited from D. So use back track.
- Pop D from the Stack.



Stack

Step 7:

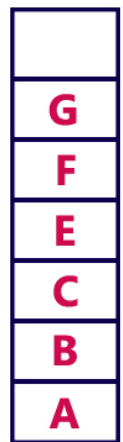
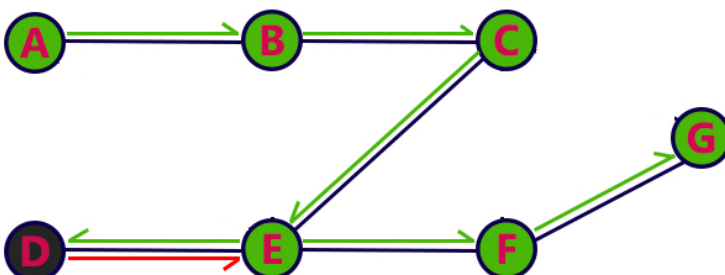
- Visit any adjacent vertex of E which is not visited (F).
- Push F on to the Stack.



Stack

Step 8:

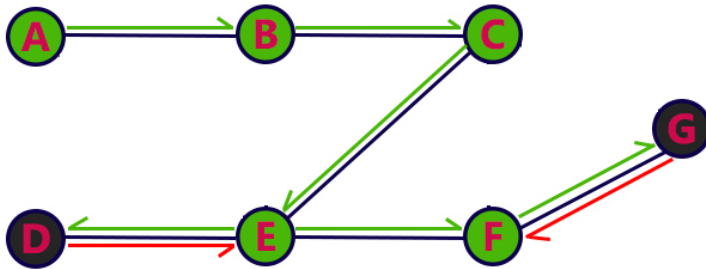
- Visit any adjacent vertex of F which is not visited (G).
- Push G on to the Stack.



Stack

Step 9:

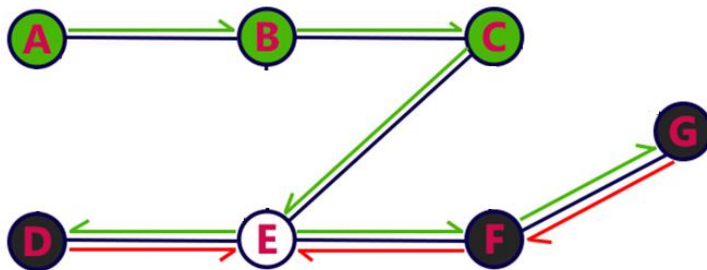
- There is no new vertex to be visited from G. So use back track.
- Pop G from the Stack.



Stack

Step 10:

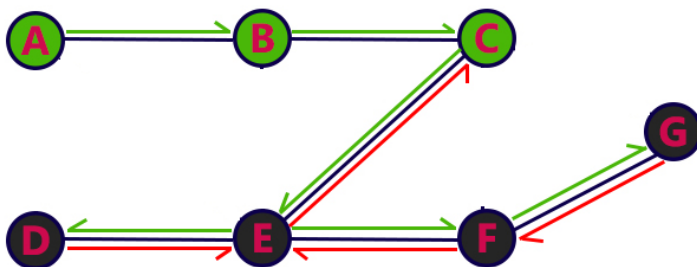
- There is no new vertex to be visited from F. So use back track.
- Pop F from the Stack.



Stack

Step 11:

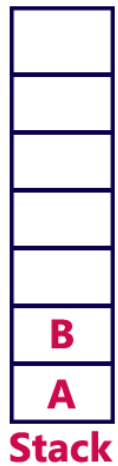
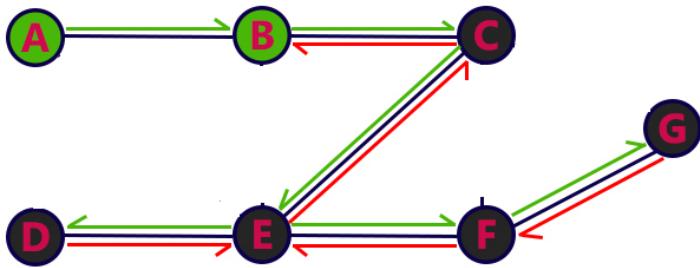
- There is no new vertex to be visited from E. So use back track.
- Pop E from the Stack.



Stack

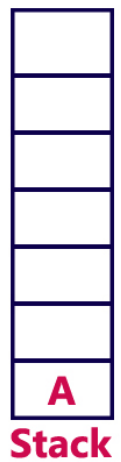
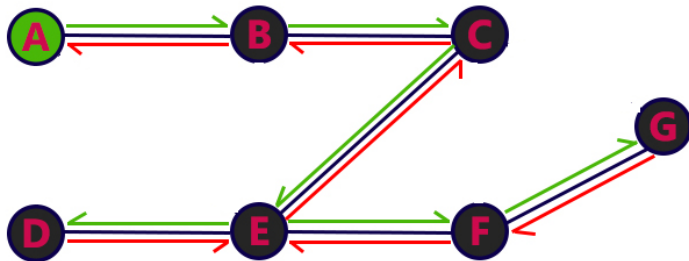
Step 12:

- There is no new vertiex to be visited from C. So use back track.
- Pop C from the Stack.



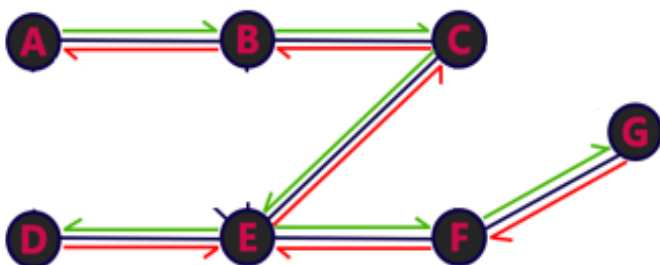
Step 13:

- There is no new vertiex to be visited from B. So use back track.
- Pop B from the Stack.

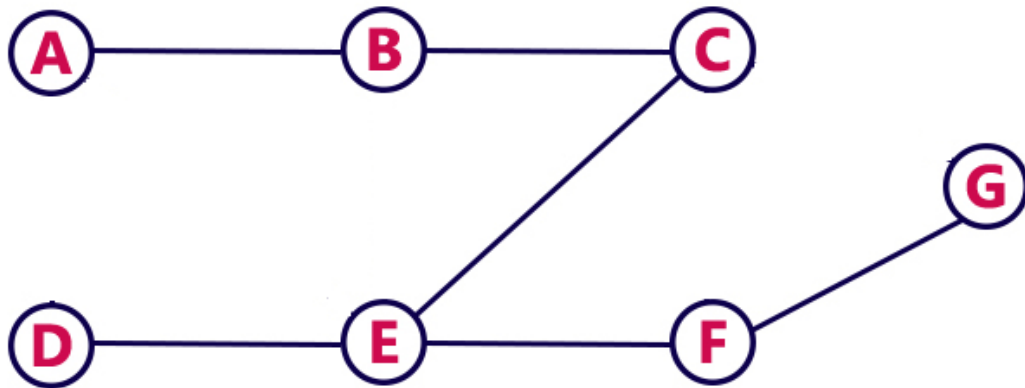


Step 14:

- There is no new vertiex to be visited from A. So use back track.
- Pop A from the Stack.



- Stack became Empty. So stop DFS Traversal.
- Final result of DFS traversal is following spanning tree.



Q2.28. Explain in detail about BFS (Breadth First Search) traversal technique?

Answer:

BFS (Breadth First Search):

- BFS traversal of a graph produces a spanning tree as final result.
- Spanning Tree is a graph without loops.
- We use Queue data structure with maximum size of total number of vertices in the graph to implement BFS traversal.
- We use the following steps to implement BFS traversal...

Step 1 - Define a Queue of size total number of vertices in the graph.

Step 2 - Select any vertex as **starting point** for traversal. Visit that vertex and insert it into the Queue.

Step 3 - Visit **all the non-visited adjacent** vertices of the vertex which is at front of the Queue and insert them into the Queue.

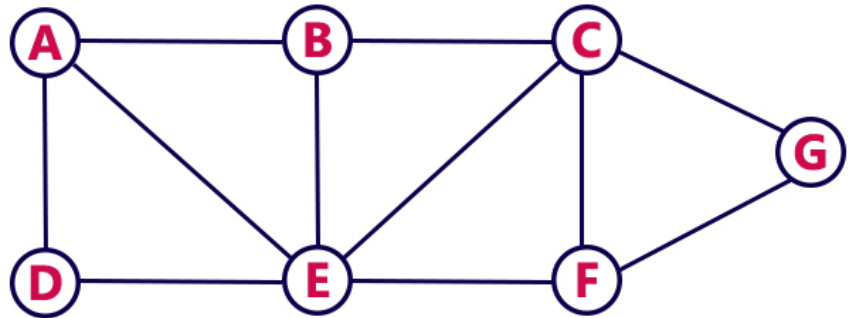
Step 4 - When there is no new vertex to be visited from the vertex which is at front of the Queue then delete that vertex.

Step 5 - Repeat steps 3 and 4 until queue becomes empty.

Step 6 - When queue becomes empty, then produce final spanning tree by removing unused edges from the graph

Example:

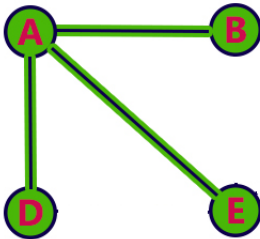
Consider the following example graph to perform BFS traversal

**Step 1:**

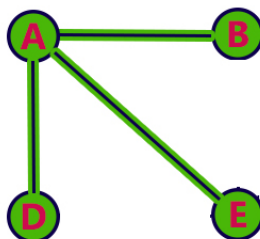
- Select the vertex **A** as starting point (visit **A**).
- Insert **A** into the Queue.

**Queue****Step 2:**

- Visit all adjacent vertices of **A** which are not visited (**D, E, B**).
- Insert newly visited vertices into the Queue and delete A from the Queue..

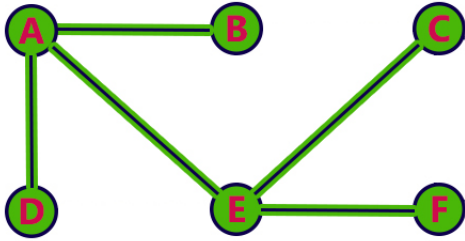
**Queue****Step 3:**

- Visit all adjacent vertices of **D** which are not visited (there is no vertex).
- Delete D from the Queue.

**Queue**

Step 4:

- Visit all adjacent vertices of **E** which are not visited (**C, F**).
- Insert newly visited vertices into the Queue and delete E from the Queue.

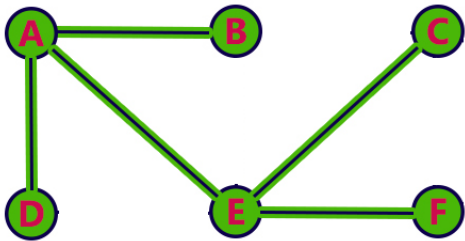


Queue



Step 5:

- Visit all adjacent vertices of **B** which are not visited (**there is no vertex**).
- Delete **B** from the Queue.

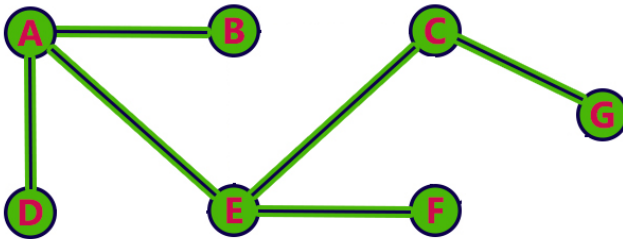


Queue



Step 6:

- Visit all adjacent vertices of **C** which are not visited (**G**).
- Insert newly visited vertex into the Queue and delete **C** from the Queue.

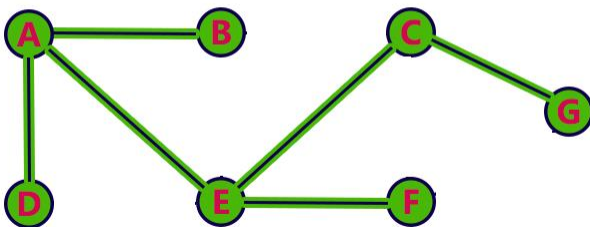


Queue



Step 7:

- Visit all adjacent vertices of **F** which are not visited (**there is no vertex**).
- Delete **F** from the Queue.

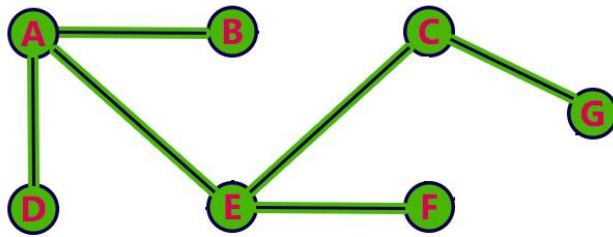


Queue



Step 8:

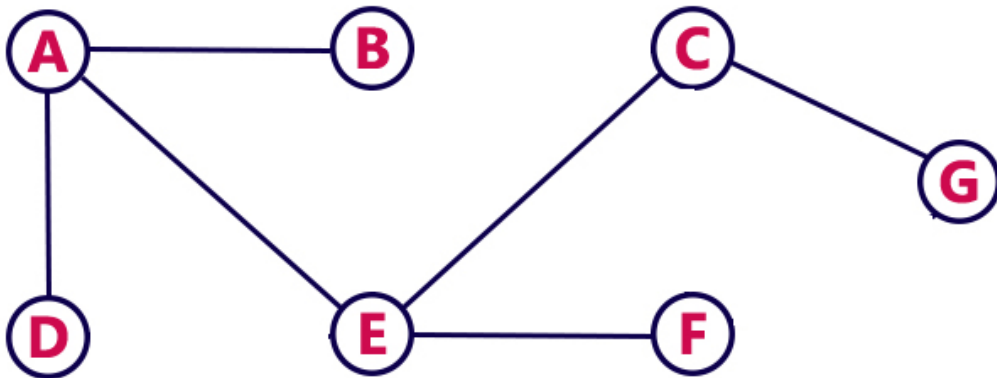
- Visit all adjacent vertices of **G** which are not visited (**there is no vertex**).
- Delete **G** from the Queue.



Queue



- Queue became Empty. So, stop the BFS process.
- Final result of BFS is a Spanning Tree as shown below...



Q2.29. Write the differences between BFS and DFS.**Answer:**

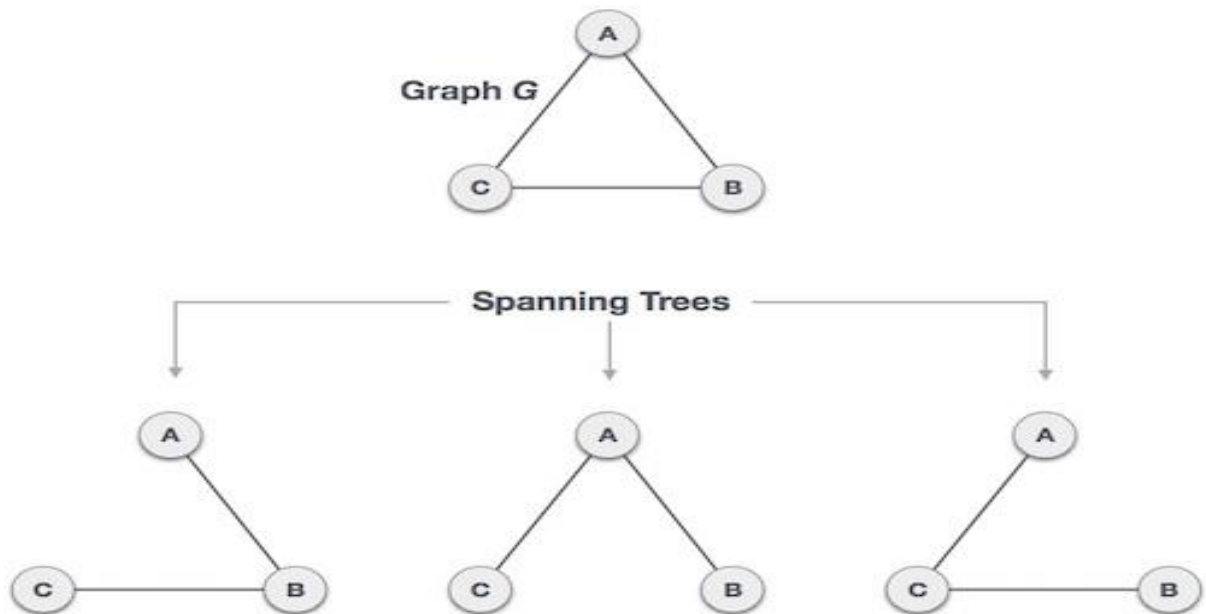
S. No.	Key	BFS	DFS
1	Definition	BFS, stands for Breadth First Search.	DFS, stands for Depth First Search.
2	Data structure	BFS uses Queue to find the shortest path.	DFS uses Stack to find the shortest path.
3	Source	BFS is better when target is closer to Source.	DFS is better when target is far from source.
4	Suitability for decision tree	As BFS considers all neighbour so it is not suitable for decision tree used in puzzle games.	DFS is more suitable for decision tree. As with one decision, we need to traverse further to augment the decision. If we reach the conclusion, we won.
5	Speed	BFS is slower than DFS.	DFS is faster than BFS.
6	Time Complexity	Time Complexity of BFS = $O(V+E)$ where V is vertices and E is edges.	Time Complexity of DFS is also $O(V+E)$ where V is vertices and E is edges.

Q2.30. Write short note on spanning trees.**Answer:****Spanning Tree:**

- A spanning tree is a subset of Graph G, which has all the vertices covered with minimum possible number of edges.
- Hence, a spanning tree does not have cycles and it cannot be disconnected.
- By this definition, we can draw a conclusion that every connected and undirected Graph G has at least one spanning tree.
- A disconnected graph does not have any spanning tree, as it cannot be spanned to all its vertices.

- A complete undirected graph can have maximum n^{n-2} number of spanning trees, where n is the number of nodes.

Example:



- In the above addressed example, n is 3, hence $3^{3-2} = 3$ spanning trees are possible.

Q2.31. what are the applications of spanning tree?

Answer:

Applications of Spanning Tree:

- Spanning tree is basically used to find a minimum path to connect all nodes in a graph.
- Common application of spanning trees is –
 1. Civil Network Planning
 2. Computer Network Routing Protocol
 3. Cluster Analysis

Q2.32. Write the properties of spanning tree?

Answer:

Properties of Spanning Tree:

1. A connected graph G can have more than one spanning tree.
2. All possible spanning trees of graph G , have the same number of edges and vertices.
3. The spanning tree does not have any cycle (loops).

4. Removing one edge from the spanning tree will make the graph disconnected, i.e. the spanning tree is **minimally connected**.
5. Adding one edge to the spanning tree will create a circuit or loop, i.e. the spanning tree is **maximally acyclic**.
6. Spanning tree has **$n-1$** edges, where **n** is the number of nodes (vertices).
7. From a complete graph, by removing maximum **$e - n + 1$** edges, we can construct a spanning tree.
8. A complete graph can have maximum **n^{n-2}** number of spanning trees.

Q2.33. Define Minimum Spanning Tree?

Answer:

Minimum Spanning Tree:

In a weighted graph, a minimum spanning tree is a spanning tree that has minimum weight than all other spanning trees of the same graph.

Q2.34. Define bi-connected graph and bi-connected component?

Answer:

Bi-Connected Graph:

A Bi-Connected graph is a connected graph that has no articulation points.

Bi-Connected Component:

A Bi-Connected component of a connected undirected graph is a maximal bi-connected subgraph.

3. Searching and Sorting

Q3.1. Define searching?

Answer:

Search is a process of finding a value in a list of values.

Q3.2. Explain linear search with example?

Answer:

Linear Search (Sequential Search):

Linear search algorithm finds a given element in a list of elements with **$O(n)$** time complexity where **n** is total number of elements in the list.

This search process starts comparing search element with the first element in the list. If both are matched, then result is element found otherwise search element is compared with the next element in the list.

Repeat the same until search element is compared with the last element in the list, if that last element also doesn't match, then the result is "Element not found in the list".

In linear search, search element is compared with element by element in the list.

Algorithm:

```
void linear_search(int list[], int size, int ele)
{
    int i;
    for(i = 0; i < size; i++)
    {
        if(ele == list[i])
        {
            printf("Element is found at %d index", i);
            break;
        }
    }
    if(i == size)
        printf("Given element is not found in the list!!!");
}
```

Time Complexity:

Best Case – $O(1)$

Average Case – $O(n)$

Worst Case – $O(n)$

Example:

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

search element **12**

Step 1:

search element (12) is compared with first element (65)

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

12

Both are not matching. So move to next element

Step 2:

search element (12) is compared with next element (20)

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

12

Both are not matching. So move to next element

Step 3:

search element (12) is compared with next element (10)

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

12

Both are not matching. So move to next element

Step 4:

search element (12) is compared with next element (55)

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

12

Both are not matching. So move to next element

Step 5:

search element (12) is compared with next element (32)

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

12

Both are not matching. So move to next element

Step 6:

search element (12) is compared with next element (12)

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

12

Both are matching. So we stop comparing and display element found at index 5.

Q3.3. Write a C program to implement linear search?**Answer:**

```
#include<stdio.h>

int linear_search(int list[], int size, int ele)
{
    int i;
    for(i = 0; i < size; i++)
    {
        if(ele == list[i])
            return i;
    }
    return -1;
}

int main()
{
    int a[20],size,ele, flag, i;
    printf("Enter size of the list: ");
    scanf("%d",&size);
    printf("Enter %d integer values:\n",size);
    for(i = 0; i < size; i++)
        scanf("%d",&a[i]);
    printf("Enter the element to be Search: ");
    scanf("%d",&ele);
    flag = linear_search(a, size, ele);
    if (flag != -1)
        printf("Element is found at %d index", flag);
    else
        printf("Given element is not found in the list!!!");
    return 0;
}
```

Output:

Enter size of the list: 5

Enter 5 integer values:

10 50 70 40 30

Enter the element to be Search: 70

Element is found at 2 index

Q3.4. Explain the process of binary search with example?**Answer:****Binary Search:**

Binary search algorithm finds a given element in a list of elements with **$O(\log n)$** time complexity where **n** is total number of elements in the list.

The binary search algorithm can be used with only a sorted list of elements. The binary search cannot be used for a list of elements arranged in random order.

This search process starts comparing the search element with the middle element in the list. If both are matched, then the result is "element found".

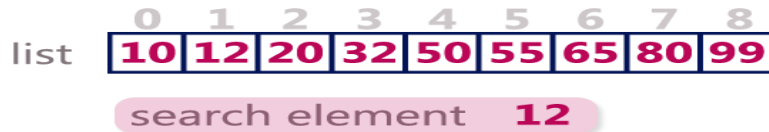
Otherwise, we check whether the search element is smaller or larger than the middle element in the list. If the search element is smaller, then we repeat the same process for the left sub list of the middle element. If the search element is larger, then we repeat the same process for the right sub list of the middle element.

We repeat this process until we find the search element in the list or until we left with a sub list of only one element. And if that element also doesn't match with the search element, then the result is "Element not found in the list".

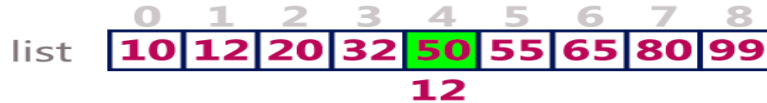
Recursive Algorithm:

```
int binary_Search(int list[], int low, int high, int ele)
{
    int mid;
    if (high >= low)
    {
        mid = (low + high) / 2;
        if (list[mid] == ele)
            return mid;
        if (list[mid] > ele)
            return binary_Search(list, low, mid-1, ele);
        return binary_Search(list, mid+1, high, ele);
    }
    return -1;
}
```

Time Complexity:**Best Case – $O(1)$** **Average Case – $O(\log n)$** **Worst Case – $O(\log n)$**

Example:**Step 1:**

search element (12) is compared with middle element (50)



Both are not matching. And 12 is smaller than 50. So we search only in the left sublist (i.e. 10, 12, 20 & 32).

**Step 2:**

search element (12) is compared with middle element (12)

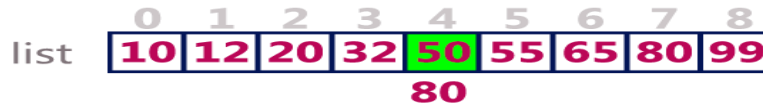


Both are matching. So the result is "Element found at index 1"

search element 80

Step 1:

search element (80) is compared with middle element (50)



Both are not matching. And 80 is larger than 50. So we search only in the right sublist (i.e. 55, 65, 80 & 99).

**Step 2:**

search element (80) is compared with middle element (65)



Both are not matching. And 80 is larger than 65. So we search only in the right sublist (i.e. 80 & 99).

**Step 3:**

search element (80) is compared with middle element (80)



Both are not matching. So the result is "Element found at index 7"

Q3.5. Write a C program to implement binary search?**Answer:**

```

#include<stdio.h>

int binary_Search(int list[], int low, int high, int ele)
{
    int mid;
    if (high >= low)
    {
        mid = (low + high) /2;
        if (list[mid] == ele)
            return mid;
        if (list[mid] > ele)
            return binary_Search(list, low, mid-1, ele);
        return binary_Search(list, mid+1, high, ele);
    }
    return -1;
}

int main()
{
    int a[20], size, ele, flag, i;
    printf("Enter the size of the list: ");
    scanf("%d",&size);
    printf("Enter %d integer values in Assending order\n", size);
    for (i = 0; i < size; i++)
        scanf("%d",&a[i]);
    printf("Enter value to be search: ");
    scanf("%d", &ele);
    flag = binary_Search(a,0,size-1,ele);
    if(flag != -1)
        printf("Element found at index %d.\n", flag);
    else
        printf("Element Not found in the list.");
}

```

Output:

Enter the size of the list: 5

Enter 5 integer values in Assending order

10 20 30 40 50

Enter value to be search: 10

Element found at index 0.

Q3.6. Define sorting?**Answer:**

Sorting is the process of arranging a list of elements in a particular order (Ascending or Descending).

Q3.7. Explain the process of bubble sort with example?**Answer:**

Bubble sort is an algorithm that compares the adjacent elements and swaps their positions if they are not in the intended order. The order can be ascending or descending.

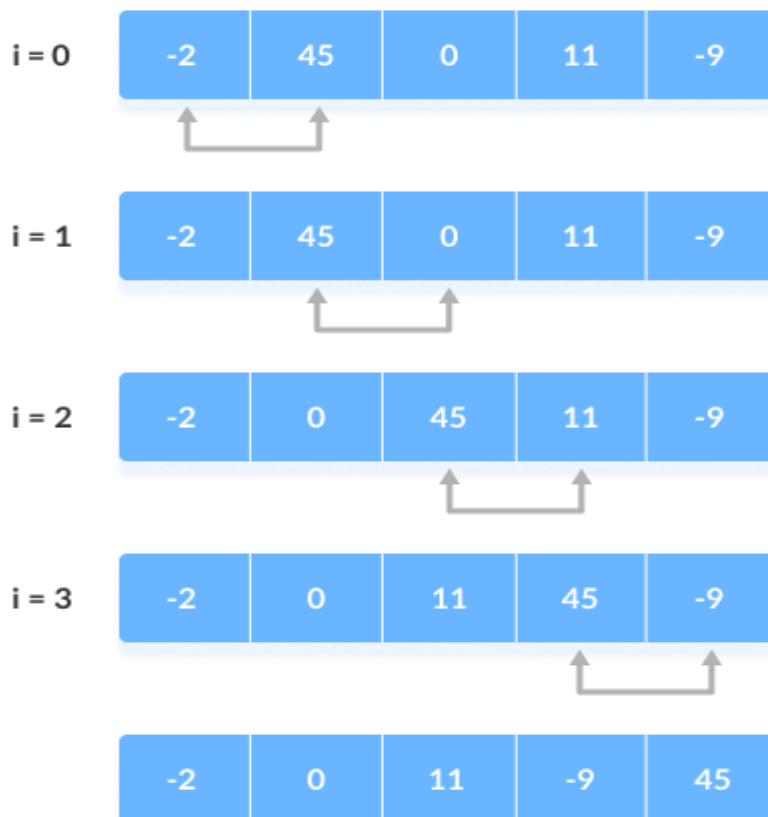
How Bubble Sort Works?

1. Starting from the first index, compare the first and the second elements. If the first element is greater than the second element, they are swapped.

Now, compare the second and the third elements. Swap them if they are not in order.

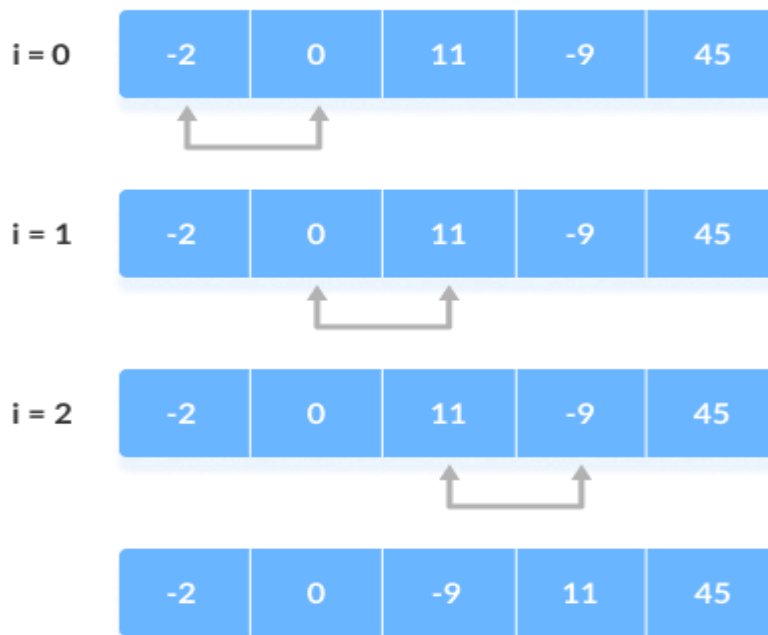
The above process goes on until the last element.

step = 0

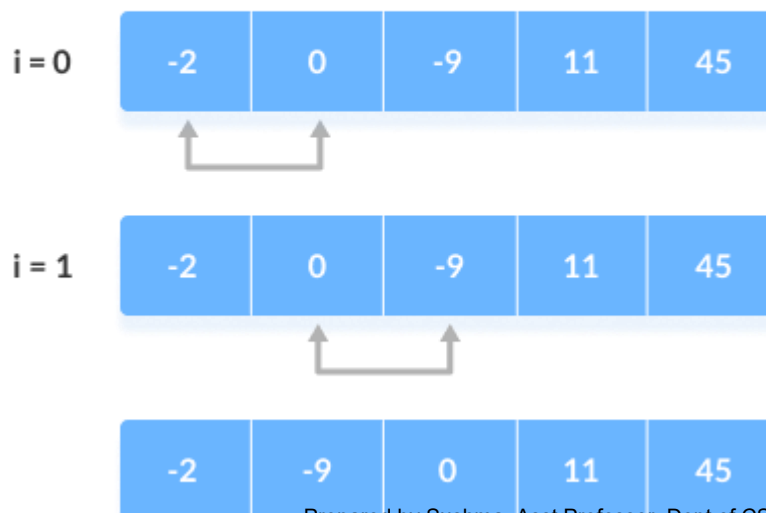


2. The same process goes on for the remaining iterations. After each iteration, the largest element among the unsorted elements is placed at the end.
- In each iteration, the comparison takes place up to the last unsorted element.
- The array is sorted when all the unsorted elements are placed at their correct positions.

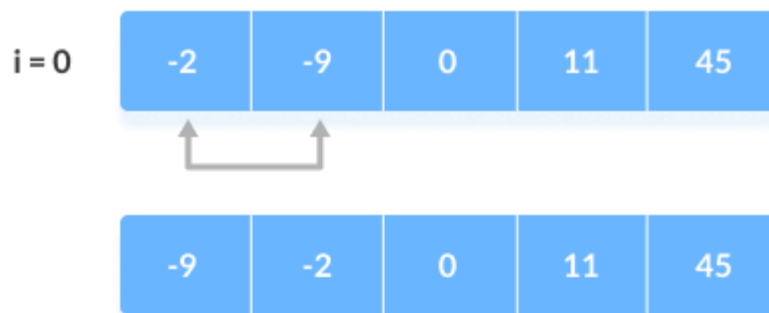
step = 1



step = 2



step = 3

**Algorithm:**

```
void bubble_sort(int list[], int size)
{
    int i,j,temp;
    for (i = 0; i < size - 1; i++)
    {
        for (j = 0; j < size - i - 1; j++)
        {
            if (list[j] > list[j + 1])
            {
                temp = list[j];
                list[j] = list[j + 1];
                list[j + 1] = temp;
            }
        }
    }
}
```

Q3.8. Write a C program to implement bubble sort?**Answer:**

```
#include <stdio.h>

void bubble_sort(int list[], int size)
{
    int i,j,temp;
    for (i = 0; i < size - 1; i++)
    {
        for (j = 0; j < size - i - 1; j++)
        {
            if (list[j] > list[j + 1])
            {
                temp = list[j];
                list[j] = list[j + 1];
                list[j + 1] = temp;
            }
        }
    }
}

int main()
{
    int a[20],i,j,size;
    printf("Enter the size of the list: ");
    scanf("%d", &size);
    printf("Enter %d integer values:\n", size);
    for (i = 0; i < size; i++)
        scanf("%d", &a[i]);
    bubble_sort(a, size);
    printf("List after Sorting is: ");
    for (i = 0; i < size; i++)
        printf("%d ", a[i]);
    return 0;
}
```

Output:

```
Enter the size of the list: 5
Enter 5 integer values:
50 40 30 20 10
List after Sorting is: 10 20 30 40 50
```

Q3.9. Explain the process of selection sort with example?**Answer:****Selection Sort:**

Selection Sort algorithm is used to arrange a list of elements in a particular order (Ascending or Descending).

In selection sort, the first element in the list is selected and it is compared repeatedly with all the remaining elements in the list. If any element is smaller than the selected element (for Ascending order), then both are swapped so that first position is filled with the smallest element in the sorted order.

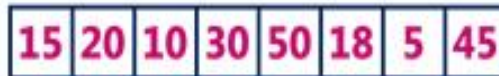
Next, we select the element at a second position in the list and it is compared with all the remaining elements in the list. If any element is smaller than the selected element, then both are swapped. This procedure is repeated until the entire list is sorted.

Algorithm:

```
for(i=0; i<size; i++)
{
    for(j=i+1; j<size; j++)
    {
        if(list[i] > list[j])
        {
            temp=list[i];
            list[i]=list[j];
            list[j]=temp;
        }
    }
}
```

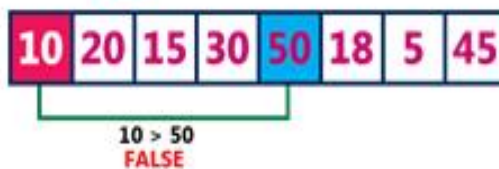
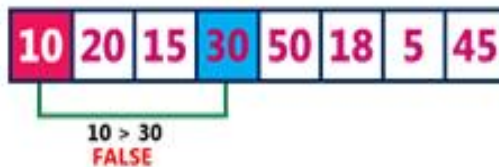
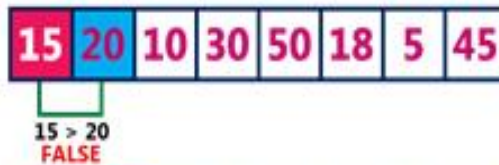
Example:

Consider the following unsorted list of elements...



Iteration #1

Select the first position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.



List after 1st iteration



Iteration #2

Select the second position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

List after 2nd iteration

5	10	20	30	50	18	15	45
---	----	----	----	----	----	----	----

Iteration #3

Select the third position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

List after 3rd iteration

5	10	15	30	50	20	18	45
---	----	----	----	----	----	----	----

Iteration #4

Select the fourth position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

List after 4th iteration

5	10	15	18	50	30	20	45
---	----	----	----	----	----	----	----

Iteration #5

Select the fifth position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

List after 5th iteration

5	10	15	18	20	50	30	45
---	----	----	----	----	----	----	----

Iteration #6

Select the sixth position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

List after 6th iteration

5	10	15	18	20	30	50	45
---	----	----	----	----	----	----	----

Iteration #7

Select the seventh position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

List after 7th iteration

5	10	15	18	20	30	45	50
---	----	----	----	----	----	----	----

Final sorted list

Q3.10. Write a C program to implement selection sort?**Answer:**

```
#include<stdio.h>

void selection_sort(int list[], int size)
{
    int i,j,temp;
    for(i=0; i<size; i++)
    {
        for(j=i+1; j<size; j++)
        {
            if(list[i]> list[j])
            {
                temp = list[i];
                list[i] = list[j];
                list[j] = temp;
            }
        }
    }
}

int main()
{
    int size,a[100],i,j;
    printf("Enter the size of the List: ");
    scanf("%d",&size);
    printf("Enter %d integer values:\n",size);
    for(i=0; i<size; i++)
        scanf("%d",&a[i]);
    selection_sort(a,size);
    printf("List after sorting is: ");
    for(i=0; i<size; i++)
        printf(" %d",a[i]);
    return 0;
}
```

Output:

Enter the size of the List: 5

Enter 5 integer values:

20 50 10 40 30

List after sorting is: 10 20 30 40 50

Q3.11. Explain the process of insertion sort with example?**Answer:****Insertion Sort:**

Insertion sort algorithm arranges a list of elements in a particular order. In insertion sort algorithm, every iteration moves an element from unsorted portion to sorted portion until all the elements are sorted in the list.

Algorithm:

```

for i = 1 to size-1
{
    temp = list[i];
    j = i-1;
    while ((temp < list[j]) && (j > 0))
    {
        list[j] = list[j-1];
        j = j - 1;
    }
    list[j] = temp;
}

```

Example:

Consider the following unsorted list of elements...



Assume that sorted portion of the list empty and all elements in the list are in unsorted portion of the list as shown in the figure below...



Move the first element 15 from unsorted portion to sorted portion of the list.

Sorted	Unsorted						
15	20	10	30	50	18	5	45

To move element 20 from unsorted to sorted portion, Compare 20 with 15 and insert it at correct position

Sorted	Unsorted						
15	20	10	30	50	18	5	45

To move element 10 from unsorted to sorted portion, Compare 10 with 20 and it is smaller so swap. Then compare 10 with 15 again smaller swap. And 10 is insert at its correct position in sorted portion of the list.

Sorted	Unsorted						
10	15	20	30	50	18	5	45

To move element 30 from unsorted to sorted portion, Compare 30 with 20, 15 and 10. And it is larger than all these so 30 is directly inserted at last position in sorted portion of the list.

Sorted	Unsorted						
10	15	20	30	50	18	5	45

To move element 50 from unsorted to sorted portion, Compare 50 with 30, 20, 15 and 10. And it is larger than all these so 50 is directly inserted at last position in sorted portion of the list.

Sorted	Unsorted						
10	15	20	30	50	18	5	45

To move element 18 from unsorted to sorted portion, Compare 18 with 30, 20 and 15. Since 18 is larger than 15, move 20, 30 and 50 one position to the right in the list and insert 18 after 15 in the sorted portion.

Sorted	Unsorted						
10	15	18	20	30	50	5	45

To move element 5 from unsorted to sorted portion, Compare 5 with 50, 30, 20, 18, 15 and 10. Since 5 is smaller than all these element, move 10, 15, 18, 20, 30 and 50 one position to the right in the list and insert 5 at first position in the sorted list.

Sorted	Unsorted						
5	10	15	18	20	30	50	45

To move element 45 from unsorted to sorted portion, Compare 45 with 50 and 30. Since 45 is larger than 30, move 50 one position to the right in the list and insert 45 after 30 in the sorted list.

Sorted	Unsorted						
5	10	15	18	20	30	45	50

Unsorted portion of the list has become empty. So we stop the process. And the final sorted list of elements is as follows...

5	10	15	18	20	30	45	50
---	----	----	----	----	----	----	----

Q3.12. Write a C program to implement insertion sort?**Answer:**

```
#include<stdio.h>
void insertion_sort(int list[], int size)
{
    int i,j,temp;
    for (i = 1; i < size; i++)
    {
        temp = list[i];
        j = i - 1;
        while ((temp < list[j]) && (j >= 0))
        {
            list[j + 1] = list[j];
            j = j - 1;
        }
        list[j + 1] = temp;
    }
}

int main()
{
    int size,i,j,a[100];
    printf("Enter the size of the list: ");
    scanf("%d", &size);
    printf("Enter %d integer values:\n", size);
    for (i = 0; i < size; i++)
        scanf("%d", &a[i]);
    insertion_sort(a, size);
    printf("List after Sorting is: ");
    for (i = 0; i < size; i++)
        printf(" %d", a[i]);
    return 0;
}
```

Output:

Enter the size of the list: 5

Enter 5 integer values:

50 20 40 10 30

List after Sorting is: 10 20 30 40 50

Q3.13. Compare the bubble, selection and insertion sorting techniques.

Answer:

Algorithm	Time Complexity		
	Best	Average	Worst
Selection Sort	$\Omega(n^2)$	$\theta(n^2)$	$O(n^2)$
Bubble Sort	$\Omega(n)$	$\theta(n^2)$	$O(n^2)$
Insertion Sort	$\Omega(n)$	$\theta(n^2)$	$O(n^2)$

2 Marks Questions**Trees:**

1. Define a binary tree and give an example.
2. List any four operations on binary tree.
3. Suppose the numbers 7, 5, 1, 8, 3, 6, 0, 9, 4, 2 are inserted in that order into an initially empty binary search tree. The binary search tree uses the usual ordering on natural numbers. What is the in-order traversal sequence of the resultant tree?
4. What is the difference between full binary tree and complete binary tree?
5. What is binary search tree?
6. What is the minimum spanning tree?
7. There are 8, 15, 13, 14 nodes, were there in 4 different trees. Which of them could have formed a full binary tree?
- 8.

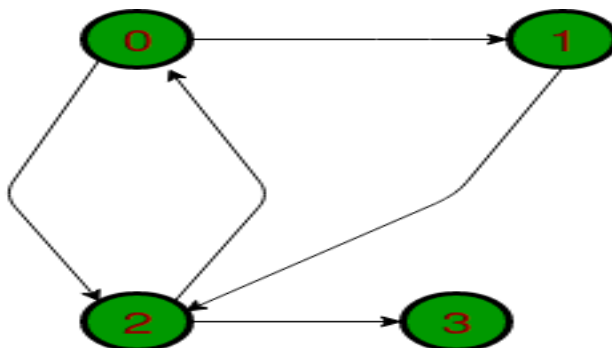
Graphs:

1. What is an articulation point in a graph?
2. Write the applications of graph data structure.
3. Define transitive closure of a graph.

Answer:

Given a directed graph, find out if a vertex j is reachable from another vertex i for all vertex pairs (i, j) in the given graph. Here reachable mean that there is a path from vertex i to j . The reach-ability matrix is called the transitive closure of a graph.

For example, consider below graph



Transitive closure of above graphs is

```

1 1 1 1
1 1 1 1
1 1 1 1
0 0 0 1
  
```

Searching and Sorting:

1. What is sorting by insertion?
2. Differentiate sorting by insertion and sorting by selection.
3. Derive the time complexity of sequential search.
4. Write non recursive pseudo code for binary search.
5. Write worst case and best case time complexity of the bubble sort algorithm.
6. Write the time complexity of linear search and binary search techniques.
7. Distinguish between linear search Vs binary search.
8. Consider a file sorted in the reverse order. Calculate the total number of comparisons when the file is sorted using insertion sort.
9. Define connected and disjoint graphs.
10. Define sort efficiency.