Programming and Data Structures

This Chapter "Programming and Data Structures" **is taken from our:**

ISBN : 9789386629029

3

Page - 119 - 200

PROGRAMMING AND DATA STRUCTURES

CONTENTS

PROGRAMMING AND DATA STRUCTURES PROGRAMMING IN C

C is a general-purpose high level language that was originally developed by Dennis Ritchie for the Unix operating system. It was first implemented on the Digital Equipment Corporation PDP-11 computer in 1972.

The Unix operating system and virtually all Unix applications are written in the C language. C has now become a widely used professional language for various reasons.

- Easy to learn
- Structured language
- It produces efficient programs.
- It can handle low-level activities.

C Program File

All the C programs are writen into text files with extension ".c" for example ***hello.c***. You can use "vi" editor to write your C program into a file.

This tutorial assumes that you know how to edit a text file and how to write programming insturctions inside a program file.

C Compilers

When you write any program in C language then to run that program you need to compile that program using a C Compiler which converts your program into a language understandable by a computer. This is called machine language (ie. binary format). So before proceeding, make sure you have C Compiler available at your computer. Preprocessor Commands

- Functions
- Variables
- Statements & Expressions
- Comments

The following program is written in the C programming language. Open a text file First.c using vi editor and put the following lines inside that file.

```
#include <stdio.h>

int main()
{
   /* My first program */
   printf("This is a C program! \n");

   return 0;
}
```

Preprocessor Commands: These commands tells the compiler to do preprocessing before doing actual compilation. Like *#include <stdio.h>* is a preprocessor command which tells a C compiler to include stdio.h file before going to actual compilation.

Functions: are main building blocks of any C Program. Every C Program will have one or more functions and there is one mandatory function which is called *main()* function. This function is prefixed with keyword int which means this function returns an integer value when it exits. This integer value is returned using return statement.The C Programming language provides a set of built-in functions. In the above example printf() is a C built-in function which is used to print anything on the screen.

Variables: are used to hold numbers, strings and complex data for manipulation.

Statements & Expressions : Expressions combine variables and constants to create new values. Statements are expressions, assignments, function calls, or control flow statements which make up C programs.

Comments: are used to give additional useful information inside a C Program. All the comments will be put inside /*...*/ as given in the example above.

Note the followings

- C is a case sensitive programming language. It means in C printf and Printf will have different meanings.
- C has a free-form line structure. End of each C statement must be marked with a semicolon.
- Multiple statements can be one the same line.
- White Spaces (ie tab space and space bar) are ignored.
- Statements can continue over multiple lines.

C - Reserved Keywords

The following names are reserved by the C language. Their meaning is already defined, and they cannot be re-defined to mean anything else.

Auto	else	Long	switch
Break	enum	register	typedef
Case	extern	return	union
Char	float	Short	unsigned
Const	for	signed	void
Continue	goto	Sizeof	volatile
Default	if	Static	while
Do	int	Struct	Packed
Double			

While naming your functions and variables, other than these names, you can choose any names of reasonable length for variables, functions etc.

C Preprocessor, Directives and Header Files:

C Preprocessor :

As part of compilation, the C compiler runs a program called the C preprocessor. The preprocessor is able to add and remove code from your source file. One of the major functions of C preprocessor is Tokenizing. The final step of the preprocessor is to link the resulting program with necessary programs and library.

This directive is used for text substitution. Every occurrence of the identifier is substituted by the substitution_token. The primary advantage is that it increases the readability of the program.

Syntax :
#define directive:
#define identifier substitution_token
For Example:
#define TRUE 1
#define FALSE
#define MAX 100
#define PI 3.14
Syntax :
#include <filename>

include directive:

This directive searches for a header or source file, which can be processed by the implementation, to be include in the program.

Header files:

Header files are a collection of macros, types, functions and constants. Any program that needs those functions can include the corresponding header files.

List of some commonly used Header file and their purposes:

Header Files	Purpose	Functions Declared
stdio.h	Used for standard input and output (I/O) operations.	printf(), scanf(), getchar(), putchar(), gets(), puts(), getc(), putc(), fopen, fclose(), feof()
conio.h	Contains declaration for console I/O functions	clrscr(); exit()
ctype.h	Used for character-handling or testing characters.	isupper(), is lower, isalpha()
math.h	Declares mathematical functions and macros	pow(), squr(), cos(), tan(), sin(), log()
stdlib.h	Used for number conversions, storage allocations.	rand(), srand()
string.h	Used for manipulating strings	streln(), strcpy(), strcmp(), stract(), strlwr(), strupr(), strrev()

Data Type

Data type determines the kind of data which is going to be stored in the variable declared to of that type. It plays major role to declare variables. It prevents invalid data entry in the program. The biggest advantage of data types is efficient use of memory space. Data type can be classified into following sub-heads.

Data type	Range of values	Memory Requirement	Properties
char	-128 to 127	1 byte	Holds a character
Int	-32,768 to 32,767	2 bytes	Holds an integer value
Float	3.4e-38b to 3.4e + e38	4 bytes	Holds single precision value
double	1.7e-308 to 1.7e + 308	8 bytes	Holds double precision value
Void		0 byte	Holds nothing

C Variable types

A variable is just a named area of storage that can hold a single value (numeric or character). The C language demands that you declare the name of each variable that you are going to use and its type, or class, before you actually try to do anything with it.

The Programming language C has two main variable types

- Local Variables
- Global Variables

Local Variables

- Local variables scope is confined within the block or function where it is defined. Local variables must always be defined at the top of a block.
- When a local variable is defined - it is not initalised by the system, you must initalise it yourself.
- When execution of the block starts the variable is available, and when the block ends the variable 'dies'.

Global Variables

Global variable is defined at the top of the program file and it can be visible and modified by any function that may reference it.

Data Type	Initialser
int	0
char	'\0'
float	0
pointer	NULL

If same variable name is being used for global and local variable then local variable takes preference in its scope. But it is not a good practice to use global variables and local variables with the same name.

C - Storage Classes

A storage class defines the scope (visibility) and life time of variables and/or functions within a C Program.

There are following storage classes which can be used in a C Program

- auto
- register
- static
- extern

auto - Storage Class

auto is the default storage class for all local variables.

```
{
    int Count;
    auto int Month;
}
```

The example above defines two variables with the same storage class. auto can only be used within functions, i.e. local variables.

register - Storage Class

register is used to define local variables that should be stored in a register instead of RAM. This means that the variable has a maximum size equal to the register size (usually one word) and cant have the unary '&' operator applied to it (as it does not have a memory location).

```
{
register int  Miles;
}
```

Register should only be used for variables that require quick access - such as counters. It should also be noted that defining 'register' goes not mean that the variable will be stored in a register. It means that it MIGHT be stored in a register - depending on hardware and implementation restrictions.

static - Storage Class

static is the default storage class for global variables. The two variables below (count and road) both have a static storage class.

```
static int Count;
int Road;

{
    printf("%d\n", Road);
}
```

static variables can be 'seen' within all functions in this source file. At link time, the static variables defined here will not be seen by the object modules that are brought in.Static can also be defined within a function. If this is done the variable is initalised at run time but is not reinitalized when the function is called. This inside a function static variable retains its value during various calls.

NOTE : Here keyword void means function does not return anything and it does not take any parameter. You can memoriese void as nothing. Static variables are initialized to 0 automatically.

Definition vs Declaration : Before proceeding, let us understand the difference between defintion and declaration of a variable or function. Definition means where a variable or function is defined in realityand actual memory is allocated for variable or function. Declaration means just giving a reference of a variable and function. Through declaration we assure to the complier that this variable or function has been defined somewhere else in the program and will be provided at the time of linking. In the above examples char *func(void) has been put at the top which is a declaration of this function where as this function has been defined below to main() function. There is one more very important use for 'static'. Consider this bit of code.

```
char *func(void);
main()
{
   char *Text1;
   Text1 = func();
}

char *func(void)
{
  char Text2[10]="martin";
  return(Text2);
}
```

Now, 'func' returns a pointer to the memory location where 'text2' starts But text2 has a storage class of 'auto' and will disappear when we exit the function and could be overwritten but something else. The storage assigned to 'text2' will remain reserved for the duration if the program.

extern - Storage Class

extern is used to give a reference of a global variable that is visible to ALL the program files. When you use 'extern' the variable cannot be initalized as all it does is point the variable name at a storage location that has been previously defined.

File 1: main.c

```
 int count=5;

main()
 {
  write_extern();
 }
```

File 2: write.c

```
 void write_extern(void);

 extern int count;

 void write_extern(void)
 {
   printf("count is %i\n", count);
 }
```

Here extern keyword is being used to declare count in another file.

Now compile these two files as follows

```
 gcc main.c write.c -o write
```

This fill produce write program which can be executed to produce result.

Count in 'main.c' will have a value of 5. If main.c changes the value of count - write.c will see the new value

The same program in C ++

First File: main.cpp

```
#include <iostream>

int count ;
extern void write_extern();

main()
{
  count = 5;
 write_extern();
}
```

Second File: support.cpp

```
#include <iostream>

extern int count;

void write_extern(void)
{
  std::cout << "Count is " << count << std::endl;
}
```

Here, extern keyword is being used to declare count in another file. Now compile these two files as follows:

```
$g++ main.cpp support.cpp -o write
```

The mutable Storage Class

The **mutable** specifier applies only to class objects That is, a mutable member can be modified by a const member function.

C - Using Constants

A C constant is usually just the written version of a number. For example 1, 0, 5.73, 12.5e9. We can specify our constants in octal or hexadecimal, or force them to be treated as long integers.

- Octal constants are written with a leading zero - 015.
- Hexadecimal constants are written with a leading 0x - 0x1ae.
- Long constants are written with a trailing L - 890L.

Character constants are usually just the character enclosed in single quotes; 'a', 'b', 'c'. Some characters can't be represented in this way, so we use a 2 character sequence as follows.

'\n' newline
'\t' tab
'\\' backslash
'\"' single quote
'\0' null (Usedautomatically to terminate character string)

In addition, a required bit pattern can be specified using its octal equivalent.
'\044' produces bit pattern 00100100.

Character constants are rarely used, since string constants are more convenient. A string constant is surrounded by double quotes eg "Brian and Dennis". The string is actually stored as an array of characters. The null character '\0' is automatically placed at the end of such a string to act as a string terminator.

A character is a different type to a single character string. This is important poing to note.

Defining Constants

ANSI C allows you to declare constants. When you declare a constant it is a bit like a variable declaration except the value cannot be changed.

The const keyword is to declare a constant, as shown below:

```
int const a = 1;
const int a =2;
```

The enum Data type

enum is the abbreviation for ENUMERATE, and we can use this keyword to declare and initialize a sequence of integer constants. Here's an example:

```
enum colors {RED, YELLOW, GREEN, BLUE};
```

I've made the constant names uppercase, but you can name them which ever way you want.

Here, colors is the name given to the set of constants - the name is optional. Now, if you don't assign a value to a constant, the default value for the first one in the list - RED in our case, has the value of 0. The rest of the undefined constants have a value 1 more than the one before, so in our case, YELLOW is 1, GREEN is 2 and BLUE is 3.

But you can assign values if you wanted to:

```
enum colors {RED=1, YELLOW, GREEN=6, BLUE };
```

Now RED=1, YELLOW=2, GREEN=6 and BLUE=7.

The main advantage of enum is that if you don't initialize your constants, each one would have a unique value. The first would be zero and the rest would then count upwards.

```
#include <stdio.h>
int main() {
 enum {RED=5, YELLOW, GREEN=4, BLUE};

  printf("RED = %d\n", RED);
 printf("YELLOW = %d\n", YELLOW);
 printf("GREEN = %d\n", GREEN);
  printf("BLUE = %d\n", BLUE);
   return 0;
}
```

This will produce following results

```
 RED = 5
 YELLOW = 6
 GREEN = 4
 BLUE = 5
```

C - Operator Types

What is Operator? Simple answer can be given using expression 4 + 5 is equal to 9. Here 4 and 5 are called operands and + is called operator. C language supports following type of operators.
C++ is rich in built-in operators and provides the following types of operators:

- Arithmetic Operators
- Logical (or Relational) Operators
- Bitwise Operators
- Assignment Operators
- Misc Operators

Lets have a look on all operators one by one.
Arithmetic Operators:
There are following arithmetic operators supported by C language:
Assume variable A holds 10 and variable B holds 20 then:
Show Examples

Operator	Description	Example
+	Adds two operands	A + B will give 30
-	Subtracts second operand from the first	A - B will give -10
*	Multiply both operands	A * B will give 200
/	Divide numerator by denumerator	B / A will give 2
%	Modulus Operator and remainder of after an integer division	B % A will give 0

Logical (or Relational) Operators:
There are following logical operators supported by C language
Assume variable A holds 10 and variable B holds 20 then:
Show Examples

Operator	Description	Example
==	Checks if the value of two operands is equal or not, if yes then condition becomes true.	(A == B) is not true.
!=	Checks if the value of two operands is equal or not, if values are not equal then condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.
&&	Called Logical AND operator. If both the operands are non zero then then condition becomes true.	(A && B) is true.
||	Called Logical OR Operator. If any of the two operands is non zero then then condition becomes true.	(A || B) is true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(A && B) is false.

Bitwise Operators:

Bitwise operator works on bits and perform bit by bit operation.

Assume if A = 60; and B = 13; Now in binary format they will be as follows:

A = 0011 1100

B = 0000 1101

A&B = 0000 1100

A|B = 0011 1101

A^B = 0011 0001

~A = 1100 0011

Show Examples

There are following Bitwise operators supported by C language

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12 which is 0000 1100
|	Binary OR Operator copies a bit if it exists in eather operand.	(A | B) will give 61 which is 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) will give 49 which is 0011 0001
~	Binary Ones Complement Operator is unary and has the efect of 'flipping' bits.	(~A) will give -60 which is 1100 0011
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 will give 240 which is 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15 which is 0000 1111

Assignment Operators:

There are following assignment operators supported by C language:

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand	C = A + B will assigne value of A + B into C
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator	C &= 2 is same as C = C & 2
^=	bitwise exclusive OR and assignment operator	C ^= 2 is same as C = C ^ 2
|=	bitwise inclusive OR and assignment operator	C |= 2 is same as C = C | 2

Misc Operators

There are few other operators supported by C Language.

Operator	Description	Example
sizeof()	Returns the size of an variable.	sizeof(a), where a is interger, will return 4.
&	Returns the address of an variable.	&a; will give actaul address of the variable.
*	Pointer to a variable.	*a; will pointer to a variable.
? :	Conditional Expression	If Condition is true ? Then value X : Otherwise value Y

Operators Categories:

All the operators we have discussed above can be categorised into following categories:

- Postfix operators, which follow a single operand.
- Unary prefix operators, which precede a single operand.
- Binary operators, which take two operands and perform a variety of arithmetic and logical operations.
- The conditional operator (a ternary operator), which takes three operands and evaluates either the second or third expression, depending on the evaluation of the first expression.
- Assignment operators, which assign a value to a variable.
- The comma operator, which guarantees left-to-right evaluation of comma-separated expressions.

Precedence of C Operators:

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator:

For example x = 7 + 3 * 2; Here x is assigned 13, not 20 because operator * has higher precedenace than + so it first get multiplied with 3*2 and then adds into 7.

Here operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedenace operators will be evaluated first.

Category	Operator	Associativity
Postfix	() [] -> . ++ - -	Left to right
Unary	+ - ! ~ ++ - - (type) * & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<<>>	Left to right
Relational	<<= >>=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %= >>= <<= &= ^= |=	Right to left
Comma	,	Left to right

All above operators are similar to C++.

Operators Precedence in C++:

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator:

For example x = 7 + 3 * 2; here, x is assigned 13, not 20 because operator * has higher precedence than +, so it first gets multiplied with 3*2 and then adds into 7.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

Category	Operator	Associativity
Postfix	() [] -> . ++ - -	Left to right
Unary	+ - ! ~ ++ - - (type)* & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<<>>	Left to right
Relational	<<= >>=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %=>>= <<= &= ^= |=	Right to left
Comma	,	Left to right

C - Flow Control Statements

C provides two sytles of flow control:

- Branching
- Looping

Branching is deciding what actions to take and looping is deciding how many times to take a certain action.

Branching:

Branching is so called because the program chooses to follow one branch or another.

if statement

This is the most simple form of the branching statements.

It takes an expression in parenthesis and an statement or block of statements. If the expression is true then the statement or block of statements gets executed otherwise these statements are skipped.

NOTE: Expression will be assumed to be true if its evaluated values is non-zeroif statements take the following form.

Syntax:

```
if (expression)
  statement;
or
if (expression)
  {
    Block of statements;
  }
if condition is true ? then X return value : otherwise Y value;
```

? : Operator

The ? : operator is just like an if ... else statement except that because it is an operator you can use it within expressions.

? : is a ternary operator in that it takes three values, this is the only ternary operator C has.

? : takes the following form:

switch statement:

The switch statement is much like a nested if .. else statement. Its mostly a matter of preference which you use, switch statement can be slightly more efficient and easier to read.

Syntax:

```
switch( expression )
    {
       case constant-expression1: statements1;
       [case constant-expression2: statements2;]
       [case constant-expression3: statements3;]
       [default : statements4;]
}
```

Using break keyword:

If a condition is met in switch case then execution continues on into the next case clause also if it is not explicitly specified that the execution should exit the switch statement. This is achieved by using break keyword.

Looping

Loops provide a way to repeat commands and control how many times they are repeated. C provides a number of looping way.

while loop

The most basic loop in C is the while loop.A while statement is like a repeating if statement. Like an If statement, if the test condition is true: the statments get executed. The difference is that after the statements have been executed, the test condition is checked again. If it is still true the statements get executed again.This cycle repeats until the test condition evaluates to false.

Basic syntax of while loop is as follows:

```
while ( expression )
{
   Single statement
   or
   Block of statements;
}
```

for loop

for loop is similar to while, it's just written differently. for statements are often used to proccess lists such a range of numbers:

Basic syntax of for loop is as follows:

```
for( expression1; expression2; expression3)
{
   Single statement
   or
   Block of statements;
}
```

In the above syntax:

- expression1 - Initialise variables.
- expression2 - Condtional expression, as long as this condition is true, loop will keep executing.
- expression3 - expression3 is the modifier which may be simple increment of a variable.

do...while loop

do ... while is just like a while loop except that the test condition is checked at the end of the loop rather than the start. This has the effect that the content of the loop are always executed at least once.

Basic syntax of do...while loop is as follows:

Syntax

```
Do
{
   Single statement
   or
   Block of statements;
}while(expression);
```

break and continue statements

C provides two commands to control how we loop:

- break -- exit form loop or switch.
- continue -- skip 1 iteration of loop.

There may be a situation, when you need to execute a block of code several number of times. In general statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.

Programming languages provide various control structures that allow for more complicated execution paths.

C++ programming language provides the following types of loop to handle looping requirements.

Loop Type	Description
while loop	Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.
for loop	Execute a sequence of statements multiple times and abbreviates the code that manages the loop variable.
do...while loop	Like a while statement, except that it tests the condition at the end of the loop body
nested loops	You can use one or more loop inside any another while, for or do..while loop.

Loop Control Statements:

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

C++ supports the following control statements.

Control Statement	Description
break statement	Terminates the loop or switch statement and transfers execution to the statement immediately following the loop or switch.
continue statement	Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.
goto statement	Transfers control to the labeled statement. Though it is not advised to use goto statement in your program.

C - Input and Output

Input : In any programming language input means to feed some data into program. This can be given in the form of file or from command line. C programming language provides a set of built-in functions to read given input and feed it to the program as per requirement.

Output : In any programming language output means to display some data on screen, printer or in any file. C programming language provides a set of built-in functions to output required data.

printf() function

This is one of the most frequently used functions in C for output.

scanf() function

This is the function which can be used to to read an input from the command line.

C - Pointing to Data

A pointer is a special kind of variable. Pointers are designed for storing memory address i.e. the address of another variable. Declaring a pointer is the same as declaring a normal variable except you stick an asterisk '*' in front of the variables identifier.

- There are two new operators you will need to know to work with pointers. The "address of" operator '&' and the "dereferencing" operator '*'. Both are prefix unary operators.
- When you place an ampersand in front of a variable you will get it's address, this can be stored in a pointer variable.
- When you place an asterisk in front of a pointer you will get the value at the memory address pointed to.

Pointers and Arrays

The most frequent use of pointers in C is for walking efficiently along arrays. In fact, in the implementation of an array, the array name represents the address of the zeroth element of the array, so you can't use it on the left side of an expression. For example:

```
 char *y;
char x[100];
```

y is of type pointer to character (although it doesn't yet point anywhere). We can make y point to an element of x by either of

```
y = &x[0];
y = x;
```

Since x is the address of x[0] this is legal and consistent. Now `*y' gives x[0]. More importantly notice the following:

```
*(y+1)  gives x[1]
*(y+i)  gives x[i]
```

and the sequence

```
y = &x[0];
y++;
```

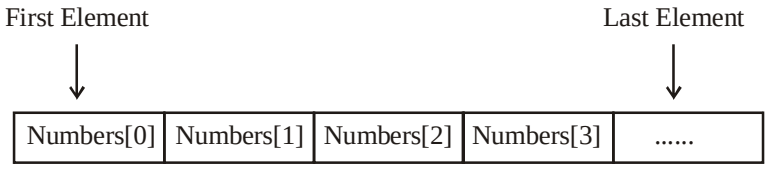
leaves y pointing at x[1].

Arrays

C programming language provides a data structure called **the array**, which can store a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables. A specific element in an array is accessed by an index.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.



int - data type

int is used to define integer numbers.

```
{
    int Count;
    Count = 5;
}
```

float - data type

float is used to define floating point numbers.

```
{
    float Miles;
    Miles = 5.6;
}
```

double - data type

double is used to define BIG floating point numbers. It reserves twice the storage for the number. On PCs this is likely to be 8 bytes.

```
{
    double Atoms;
    Atoms = 2500000;
}
```

char - data type

char defines characters.

```
{
    char Letter;
    Letter = 'x';
}
```

Modifiers

The data types explained above have the following modifiers.

- short
- long
- signed
- unsigned

The modifiers define the amount of storage allocated to the variable. The amount of storage allocated is not cast in stone.

Qualifiers

A type qualifier is used to refine the declaration of a variable, a function, and parameters, by specifying whether:

- The value of a variable can be changed.
- The value of a variable must always be read from memory rather than from a register

Standard C language recognizes the following two qualifiers:

- const
- volatile

The *const* qualifier is used to tell C that the variable value can not change after initialisation.

const float pi=3.14159;

Now pi cannot be changed at a later time within the program.

ARRAYS:

We have seen all basic data types. In C language it is possible to make arrays whose elements are basic types. Thus we can make an array of 10 integers with the declaration.

int x[10];

The square brackets mean subscripting; parentheses are used only for function references. Array indexes begin at zero, so the elements of x are:

Thus Array are special type of variables which can be used to store multiple values of same data type. Those values are stored and accessed using subscript or index. Arrays occupy consecutive memory slots in the computer's memory.

x[0], x[1], x[2], ..., x[9]

If an array has n elements, the largest subscript is n-1.

Multiple-dimension arrays are provided. The declaration and use look like:

```
int name[10] [20];
n = name[i+j] [1] + name[k] [2];
```

Subscripts can be arbitrary integer expressions. Multi-dimension arrays are stored by row so the rightmost subscript varies fastest. In above example name has 10 rows and 20 columns.

Same way, arrays can be defined for any data type. Text is usually kept as an array of characters. By convention in C, the last character in a character array should be a `\0' because most programs that manipulate character arrays expect it. For example, printf uses the `\0' to detect the end of a character array when printing it out with a `%s'.

Array Initialization

- As with other declarations, array declarations can include an optional initialization
- Scalar variables are initialized with a single value
- Arrays are initialized with a list of values
- The list is enclosed in curly braces

int array [8] = {2, 4, 6, 8, 10, 12, 14, 16};

The number of initializers cannot be more than the number of elements in the array but it can be less in which case, the remaining elements are initialized to 0.if you like, the array size can be inferred from the number of initializers by leaving the square brackets empty so these are identical declarations:

int array1 [8] = {2, 4, 6, 8, 10, 12, 14, 16};
int array2 [] = {2, 4, 6, 8, 10, 12, 14, 16};

An array of characters ie string can be initialized as follows:

char string[10] = "Hello";

To declare an array in C or C++, the programmer specifies the type of the elements and the number of elements required by an array as follows:

type arrayName [arraySize];

This is called a single-dimension array. The **arraySize** must be an integer constant greater than zero and type can be any valid C++ data type. For example, to declare a 10-element array called balance of type double, use this statement:

double balance[10];

Initializing Arrays

You can initialize array in C either one by one or using a single statement as follows:

double balance[5]={1000.0,2.0,3.4,17.0,50.0};

The number of values between braces { } can not be larger than the number of elements that we declare for the array between square brackets [ ].

If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write:

double balance[]={1000.0,2.0,3.4,17.0,50.0};

You will create exactly the same array as you did in the previous example. Following is an example to assign a single element of the array:

balance[4]=50.0;

The above statement assigns element number 5th in the array with a value of 50.0. All arrays have 0 as the index of their first element which is also called base index and last index of an array will be total size of the array minus 1. Following is the pictorial representation of the same array we discussed above:

	0	1	2	3	4
balance	1000.0	2.0	3.4	7.0	50.0

Accessing Array Elements

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. For example:

double salary = balance[9];

The above statement will take 10th element from the array and assign the value to salary variable.

Dynamic Memory Allocation:

It enables us to create data types and structures of any size and length to suit our programs need within the program. We use dynamic memory allocation concept when we don't know how in advance about memory requirement.

There are following functions to use for dynamic memory manipulation:

- void *calloc(size_t num elems, size_t elem_size) - Allocate an array and initialise all elements to zero .
- void free(void *mem address) - Free a block of memory.
- void *malloc(size_t num bytes) - Allocate a block of memory.
- void *realloc(void *mem address, size_t newsize) - Reallocate (adjust size) a block of memory.

Command Line Arguments:

It is possible to pass arguments to C programs when they are executed. The brackets which follow main are used for this purpose. argc refers to the number of arguments passed, and argv[] is a pointer array which points to each argument which is passed to main.

```
#include <stdio.>h
main( int argc, char *argv[] )
{
   if( argc == 2 )
      printf("The argument supplied is %s\n", argv[1]);
   else if( argc > 2 )
      printf("Too many arguments supplied.\n");
    else
      printf("One argument expected.\n");
}
```

Note that *argv[0] is the name of the program invoked, which means that *argv[1] is a pointer to the first argument supplied, and *argv[n] is the last argument. If no arguments are supplied, argc will be one. Thus for n arguments, argc will be equal to n + 1. The program is called by the command line:

```
$myprog  argument1
```

More clearly, Suppose a program is compiled to an executable program myecho and that the program is executed with the following command.

```
$myeprog aaa bbb ccc
```

When this command is executed, the command interpreter calls the main() function of the myprog program with 4 passed as the argc argument and an array of 4 strings as the argv argument.

```
argv[0]    -    "myprog"
argv[1]    -    "aaa"
argv[2]    -    "bbb"
argv[3]    -    "ccc"
```

Multidimensional Arrays:

The array we used in the last example was a one dimensional array. Arrays can have more than one dimension, these arrays-of-arrays are called multidimensional arrays. They are very similar to standard arrays with the exception that they have multiple sets of square brackets after the array identifier. The above array has two dimensions and can be called a doubly subscripted array.

C/C++ Arrays in Detail

Arrays are important to C and should need lots of more details. There are following few important concepts related to array which should be clear to a C programmer:

Concept	Description
Multi-dimensional arrays	It supports multidimensional arrays. The simplest form of the multidimensional array is the two-dimensional array.

Passing arrays to functions	You can pass to the function a pointer to an array by specifying the array's name without an index.
Return array from a function	It allows a function to return an array.
Pointer to an array	You can generate a pointer to the first element of an array by simply specifying the array name, without any index.

POINTER ARITHMETIC

C is one of the few languages that allows pointer arithmetic. In other words, you actually move the pointer reference by an arithmetic operation. For example:

```
int x = 5, *ip = &x;

ip++;
```

On a typical 32-bit machine, *ip would be pointing to 5 after initialization. But ip++; increments the pointer 32-bits or 4-bytes. So whatever was in the next 4-bytes, *ip would be pointing at it.

Pointer arithmetic is very useful when dealing with arrays, because arrays and pointers share a special relationship in C.

Generic Pointers: (void Pointer)

When a variable is declared as being a pointer to type void it is known as a generic pointer. Since you cannot have a variable of type void, the pointer will not point to any data and therefore cannot be dereferenced. It is still a pointer though, to use it you just have to cast it to another kind of pointer first. Hence the term Generic pointer. This is very useful when you want a pointer to point to data of different types at different times.

NOTE-1 : Here in first print statement, the_data is prefixed by ***(int*)**. This is called type casting in C language.Type is used to caste a variable from one data type to another datatype to make it compatible to the lvalue.

NOTE-2 : lvalue is something which is used to left side of a statement and in which we can assign some value. A constant can't be an lvalue because we can not assign any value in contact. For example x = y, here x is lvalue and y is rvalue.

However, above example will produce following result:

```
the_data points to the integer value 6
the_data now points to the character a
```

POINTERS

A **pointer** is a variable whose value is the address of another variable. Like any variable or constant, you must declare a pointer before you can work with it. The general form of a pointer variable declaration is:

```
type *var-name;
```

Here, **type** is the pointer's base type; it must be a valid C++ type and var-name is the name of the pointer variable. The asterisk you used to declare a pointer is the same asterisk that you use for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer. Following are the valid pointer declaration:

```
int*ip;// pointer to an integer
double*dp;// pointer to a double
float*fp;// pointer to a float
char*ch      // pointer to character
```

The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

C - Using Functions

A function is a module or block of program code which deals with a particular task. Making functions is a way of isolating one block of code from other independent blocks of code.

Functions serve two purposes.

- They allow a programmer to say: `this piece of code does a specific job which stands by itself and should not be mixed up with anyting else',
- Second they make a block of code reusable since a function can be reused in many different contexts without repeating parts of the program text.

A function can take a number of parameters, do required processing and then return a value. There may be a function which does not return any value.

Consider the following code

```
int total = 10;
printf("Hello World");
total = total + l;
```

To turn it into a function you simply wrap the code in a pair of curly brackets to convert it into a single compound statement and write the name that you want to give it in front of the brackets:

```
Demo()
{
   int total = 10;
   printf("Hello World");
   total = total + l;
}
```

curved brackets after the function's name are required. You can pass one or more paramenters to a function as follows:

```
Demo( int par1, int par2)
{
   int total = 10;
   printf("Hello World");
   total = total + l;
}
```

By default function does not return anything. But you can make a function to return any value as follows:

```
int Demo( int par1, int par2)
{
   int total = 10;
   printf("Hello World");
   total = total + l;

   return total;
}
```

A return keyword is used to return a value and datatype of the returned value is specified before the name of function. In this case function returns total which is int type. If a function does not return a value then void keyword can be used as return value.

Once you have defined your function you can use it within a program:

```
main()
{
  Demo();
}
```

Functions and Variables:

Each function behaves the same way as C language standard function main(). So a function will have its own local variables defined. In the above example total variable is local to the function Demo.

A global variable can be accessed in any function in similar way it is accessed in main() function.

Declaration and Definition

When a function is defined at any place in the program then it is called function definition. At the time of definition of a function actual logic is implemented with-in the function.

A function declaration does not have any body and they just have their interfaces. A function declaration is usually declared at the top of a C source file, or in a separate header file.

A function declaration is sometime called function prototype or function signature. For the above Demo() function which returns an integer, and takes two parameters a function declaration will be as follows:

int Demo(int par1, int par2);

Passing Parameters to a Function

There are two ways to pass parameters to a function:

- **Pass by Value:** mechanism is used when you don't want to change the value of passed paramters. When parameters are passed by value then functions in C create copies of the passed in variables and do required processing on these copied variables.
- **Pass by Reference :** mechanism is used when you want a function to do the changes in passed parameters and reflect those changes back to the calling function. In this case only addresses of the variables are passed to a function so that function can work directly over the addresses.

By default, C++ uses **call by value** to pass arguments.

While calling a function, there are two ways that arguments can be passed to a function:

Call Type	**Description**
Call by value	This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.
Call by pointer	This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.
Call by reference	This method copies the reference of an argument into the formal parameter. Inside the function, the reference is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

Here are two programs to understand the difference: First example is for Pass by value:

```
#include <stdio.h>

/* function declaration goes here.*/
void swap( int p1, int p2 );

int main()
{
   int a = 10;
   int b = 20;

   printf("Before: Value of a = %d and value of b = %d\n", a, b );
   swap( a, b );
   printf("After: Value of a = %d and value of b = %d\n", a, b );
}
```

```
void swap( int p1, int p2 )
{
    int t;

    t = p2;
    p2 = p1;
    p1 = t;
   printf("Value of a (p1) = %d and value of b(p2) = %d\n", p1, p2 );
}
```

Here is the result produced by the above example. Here the values of a and b remain unchanged before calling swap function and after calling swap function.

```
Before: Value of a = 10 and value of b = 20
Value of a (p1) = 20 and value of b(p2) = 10
After: Value of a = 10 and value of b = 20
```

Following is the example which demonstrate the concept of **pass by reference**

```
#include <stdio.h>
/* function declaration goes here.*/
void swap( int *p1, int *p2 );

int main()
{
   int a = 10;
   int b = 20;

   printf("Before: Value of a = %d and value of b = %d\n", a, b );
   swap( &a, &b );
   printf("After: Value of a = %d and value of b = %d\n", a, b );
}

void swap( int *p1, int *p2 )
{
    int t;

    t = *p2;
    *p2 = *p1;
    *p1 = t;
   printf("Value of a (p1) = %d and value of b(p2) = %d\n", *p1, *p2 );
}
```

Here is the result produced by the above example. Here the values of a and b are changes after calling swap function.

```
Before: Value of a = 10 and value of b = 20
Value of a (p1) = 20 and value of b(p2) = 10
After: Value of a = 20 and value of b = 10
```

Recursion

What is recursion? The simple answer is, it's when a function calls itself. But how does this happen? Why would this happen, and what are its uses?

When we talk about recursion, we are really talking about creating a loop. Let's start by looking at a basic loop.

```
1 for(int i=0;  i<10; i++) {
2   cout << "The number is: " << i << endl;
3 }
```

For those who don't yet know, this basic loop displays the sentence, "The number is: " followed by the value of 'i'. Like this.

The number is: 0
The number is: 1
The number is: 2
The number is: 3
The number is: 4
The number is: 5
The number is: 6
The number is: 7
The number is: 8
The number is: 9

STRINGS

- In C language Strings are defined as an array of characters or a pointer to a portion of memory containing ASCII characters. A string in C is a sequence of zero or more characters followed by a NULL '\0' character:
- It is important to preserve the NULL terminating character as it is how C defines and manages variable length strings. All the C standard library functions require this for successful operation.
- All the string handling functions are prototyped in: string.h or stdio.h standard header file. So while using any string related function, don't forget to include either stdio.h or string.h. May be your compiler differes so please check before going ahead.
- If you were to have an array of characters WITHOUT the null character as the last element, you'd have an ordinary character array, rather than a string constant.
- String constants have double quote marks around them, and can be assigned to char pointers as shown below. Alternatively, you can assign a string constant to a char array - either with no size specified, or you can specify a size, but don't forget to leave a space for the null character!

```
char *string_1 = "Hello";
char string_2[] = "Hello";
char string_3[6] = "Hello";
```

Reading and Writing Strings:

One possible way to read in a string is by using scanf. However, the problem with this, is that if you were to enter a string which contains one or more spaces, scanf would finish reading when it reaches a space, or if return is pressed. As a result, the string would get cut off. So we could use the gets function

A gets takes just one argument - a char pointer, or the name of a char array.A puts function is similar to gets function in the way that it takes one argument - a char pointer. This also automatically adds a newline character after printing out the string.

```
#include <stdio.h>

int main() {
  char array1[50];
   char *array2;

   printf("Now enter another string less than 50");
   printf(" characters with spaces: \n");
   gets(array1);

   printf("\nYou entered: ");
   puts(array1);

   printf("\nTry entering a string less than 50");
   printf(" characters, with spaces: \n");
```

```
  scanf("%s", array2);

  printf("\nYou entered: %s\n", array2);

  return 0;
}
```

This will produce following result:

```
Now enter another string less than 50 characters with spaces:
hello world

You entered: hello world

Try entering a string less than 50 characters, with spaces:
hello world

You entered: hello
```

String Manipulation Functions

- char *strcpy (char *dest, char *src) - Copy src string string into dest string.
- char *strncpy(char *string1, char *string2, int n) - Copy first n characters of string2 to stringl .
- int strcmp(char *string1, char *string2) - Compare string1 and string2 to determine alphabetic order.
- int strncmp(char *string1, char *string2, int n) - Compare first n characters of two strings.
- int strlen(char *string) - Determine the length of a string.
- char *strcat(char *dest, const char *src); - Concatenate string src to the string dest.
- char *strncat(char *dest, const char *src, int n); - Concatenate n chracters from string src to the string dest.
- char *strchr(char *string, int c) - Find first occurrence of character c in string.
- char *strrchr(char *string, int c) - Find last occurrence of character c in string.
- char *strstr(char *string2, char string*1) - Find first occurrence of string string1 in string2.
- char *strtok(char *s, const char *delim) - Parse the string s into tokens using delim as delimiter.

C++ supports a wide range of functions that manipulate null-terminated strings:

S.N.	Function & Purpose
1	**strcpy(s1, s2);** Copies string s2 into string s1.
2	**strcat(s1, s2);** Concatenates string s2 onto the end of string s1.
3	**strlen(s1);** Returns the length of string s1.
4	**strcmp(s1, s2);** Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.
5	**strchr(s1, ch);** Returns a pointer to the first occurrence of character ch in string s1.
6	**strstr(s1, s2);**Returns a pointer to the first occurrence of string s2 in string s1.

C - Structured Datatypes

- A structure in C is a collection of items of different types. You can think of a structure as a "record" is in Pascal or a class in Java without methods.

- Structures, or structs, are very useful in creating data structures larger and more complex than the ones we have discussed so far.
- Simply you can group various built-in data types into a structure.

Following is the example how to define a structure.

```
struct student {
    char firstName[20];
    char lastName[20];
    char SSN[9];
     float gpa;
  };
```

Now you have a new datatype called student and you can use this datatype define your variables of student type:

```
struct student student_a, student_b;
```

or an array of students as

```
struct student students[50];
```

Another way to declare the same thing is:

```
struct {
    char firstName[20];
    char lastName[20];
    char SSN[10];
     float gpa;
  } student_a, student_b;
```

All the variables inside an structure will be accessed using these values as student_a.firstName will give value of firstName variable. Similarly we can aqccess other variables.

Pointers to Structs:

Sometimes it is useful to assign pointers to structures (this will be evident in the next section with self-referential structures). Declaring pointers to structures is basically the same as declaring a normal pointer:

```
struct student *student_a;
```

To dereference, you can use the infix operator: ->.

```
printf("%s\n", student_a->SSN);
```

typedef Keyword

There is an easier way to define structs or you could "alias" types you create. For example:

```
typedef struct{
    char firstName[20];
    char lastName[20];
    char SSN[10];
     float gpa;
  }student;
```

Now you can use student directly to define variables of student type without using struct keyword. Following is the example:

```
student student_a;
```

You can use typedef for non-structs:

```
typedef long int *pint32;

pint32 x, y, z;
```

x, y and z are all pointers to long ints

Unions Datatype

Unions are declared in the same fashion as structs, but have a fundamental difference. Only one item within the union can be used at any time, because the memory allocated for each item inside the union is in a shared memory location.

Here is how we define a Union

```
union Shape {
      int circle;
      int triangle;
       int ovel;
};
```

We use union in such case where only one condition will be applied and only one variable will be used.

C - Working with Files

When accessing files through C, the first necessity is to have a way to access the files. For C File I/O you need to use a FILE pointer, which will let the program keep track of the file being accessed. For Example:

```
FILE *fp;
```

To open a file you need to use the **fopen** function, which returns a FILE pointer. Once you've opened a file, you can use the FILE pointer to let the compiler perform input and output functions on the file.

```
FILE *fopen(const char *filename, const char *mode);
```

Here filename is string literal which you will use to name your file and mode can have one of the following values

w - open for writing (file need not exist)
a - open for appending (file need not exist)
r+ - open for reading and writing, start at beginning
w+ - open for reading and writing (overwrite file)
a+ - open for reading and writing (append if file exists)

Note that it's possible for fopen to fail even if your program is perfectly correct: you might try to open a file specified by the user, and that file might not exist (or it might be write-protected). In those cases, fopen will return 0, the NULL pointer.

Here's a simple example of using fopen:

```
FILE *fp;

fp=fopen("/home/tutorialspoint/test.txt", "r");
```

This code will open test.txt for reading in text mode. To open a file in a binary mode you must add a b to the end of the mode string; for example, "rb" (for the reading and writing modes, you can add the b either after the plus sign - "r+b" - or before - "rb+")

To close a function you can use the function:

```
int fclose(FILE *a_file);
```

fclose returns zero if the file is closed successfully.

An example of fclose is:

```
fclose(fp);
```

To work with text input and output, you use fprintf and fscanf, both of which are similar to their friends printf and scanf except that you must pass the FILE pointer as first argument.

Try out following example:

```
#include <stdio.h>

main()
{
   FILE *fp;

    fp = fopen("/tmp/test.txt", "w");
    fprintf(fp, "This is testing...\n");
    fclose(fp;);

}
```

This will create a file test.txt in /tmp directory and will write This is testing in that file.

Here is an example which will be used to read lines from a file:

```
#include <stdio.h>

main()
{
   FILE *fp;
   char buffer[20];

    fp = fopen("/tmp/test.txt", "r");
    fscanf(fp, "%s", buffer);
   printf("Read Buffer: %s\n", %buffer );
    fclose(fp;);

}
```

It is also possible to read (or write) a single character at a time--this can be useful if you wish to perform character-by-character input. The fgetc function, which takes a file pointer, and returns an int, will let you read a single character from a file:

```
int fgetc (FILE *fp);
```

The *fgetc* returns an int. What this actually means is that when it reads a normal character in the file, it will return a value suitable for storing in an unsigned char (basically, a number in the range 0 to 255). On the other hand, when you're at the very end of the file, you can't get a character value--in this case, fgetc will return "EOF", which is a constnat that indicates that you've reached the end of the file.

The fputc function allows you to write a character at a time--you might find this useful if you wanted to copy a file character by character. It looks like this:

```
int fputc( int c, FILE *fp );
```

Note that the first argument should be in the range of an unsigned char so that it is a valid character. The second argument is the file to write to. On success, fputc will return the value c, and on failure, it will return EOF.

Binary I/O

There are following two functions which will be used for binary input and output:

```
size_t fread(void *ptr, size_t size_of_elements,
                        size_t number_of_elements, FILE *a_file);

size_t fwrite(const void *ptr, size_t size_of_elements,
                        size_t number_of_elements, FILE *a_file);
```

Both of these functions deal with blocks of memories - usually arrays. Because they accept pointers, you can also use these functions with other data structures; you can even write structs to a file or a read struct into memory.

Abstract Data types

C is not object-oriented, but we can still manage to inject some object-orientedprinciples into the design of C code. For example, a data structure and its operationscan be packaged together into an entity called an Abstract data type. There's a clean,simple interface between the abstract data type and the program(s) that use it. The lower-level implementation details of the data structure are hidden from view of therest of the Program.

Example:-

stack: operations are "push an item onto the stack", "pop an item from the stack", "ask if the stack is empty"; implementation may be as array or linked list queue: operations are "add to the end of the queue", "delete from the beginning of the queue", "ask if the queue is empty"; implementation may be as array or linked list or heap.

search structure: operations are "insert an item", "ask if an item is in the structure", and "delete an item"; implementation may be as array, linked list, tree, hash table.

Data abstraction refers to, providing only essential information to the outside world and hiding their background details, i.e., to represent the needed information in program without presenting the details.
Data abstraction is a programming (and design) technique that relies on the separation of interface and implementation.

Now, if we talk in terms of C++ Programming, C++ classes provides great level of **data abstraction**. They provide sufficient public methods to the outside world to play with the functionality of the object and to manipulate object data, i.e., state without actually knowing how class has been implemented internally.
In C++, we use **classes** to define our own abstract data types (ADT). You can use the **cout** object of class **ostream** to stream data to standard output like this:

```
#include<iostream>
usingnamespace  std;

int main()
{
   cout <<"Hello C++"<<endl;
return0;
}
```

Here, you don't need to understand how **cout** displays the text on the user's screen. You need to only know the public interface and the underlying implementation of cout is free to change.

Access Labels Enforce Abstraction:

In C++, we use access labels to define the abstract interface to the class. A class may contain zero or more access labels:

- Members defined with a public label are accessible to all parts of the program. The data-abstraction view of a type is defined by its public members.
- Members defined with a private label are not accessible to code that uses the class. The private sections hide the implementation from code that uses the type.

There are no restrictions on how often an access label may appear. Each access label specifies the access level of the succeeding member definitions. The specified access level remains in effect until the next access label is encountered or the closing right brace of the class body is seen.

Benefits of Data Abstraction:

Data abstraction provides two important advantages:

- Class internals are protected from inadvertent user-level errors, which might corrupt the state of the object.
- The class implementation may evolve over time in response to changing requirements or bug reports without requiring change in user-level code.

By defining data members only in the private section of the class, the class author is free to make changes in the data. If the implementation changes, only the class code needs to be examined to see what affect the change may have. If data are public, then any function that directly accesses the data members of the old representation might be broken.

Designing Strategy:

Abstraction separates code into interface and implementation. So while designing your component, you must keep interface independent of the implementation so that if you change underlying implementation then interface would remain intact.
In this case whatever programs are using these interfaces, they would not be impacted and would just need a recompilation with the latest implementation

OOPS

Introduction

The object-oriented programming (OOP) is a different approach to programming. Object oriented technology supported by C++ is considered the latest technology in software development. It is regarded as the ultimate paradigm for the modelling of information, be that data or logic.

Objectives

After going through this lesson, you would be able to:

l learn the basic concepts used in OOP

l describe the various benefits provided by OOP

l explain the programming applications of OOP.

Object-Oriented Programming

The object-oriented programming is a different approach to programming. It has been created with a view to increase programmer's productivity by overcoming the weaknesses found in procedural programming approach. Over the years many object-oriented programming languages such as C++ and smalltalk have come up and are becoming quite popular in the market. The major need for developing such languages was to manage the ever-increasing size and complexity of programs.

Basic Concepts

The following are the basic concepts used in object-oriented programming.

l Objects

l Classes

l Data abstraction

l Modularity

l Inheritance

l Polymorphism

Objects

It can represent a person, a bank account or any item that a program can handle. When a program is executed, the objects interact by sending messages to one another. For example, if 'customer' and 'account' are two objects in a program, then the customer object may send message to account object requesting for a bank balance. Each object contains data and code to manipulate data. Objects can interact without having to know details of each other's data or code. It is sufficient to know the type of massage accepted and the type of response returned by the objects.

Classes

We have just mentioned that objects contain data and function or code to manipulate that data. The entire set of data and code of an object can be made a user-defined data type with the help of a class. In fact objects are variables of type class. Once a class has been defined, we can create any number of objects associated with that class. For example, mango, apple and orange are members of class fruit. If fruit has been defined as a class, then the statement fruit mango, will create an object mango belonging to the class fruit.

Data Abstraction

Abstraction refers to the act of representing essential features without including the background details. To understand this concept more clearly, take an example of 'switch board'. You only press particular switches as per your requirement. You need not know the internal working of these switches. What is happening inside is hidden from you. This is abstraction, where you only know the essential things to operate on switch board without knowing the background details of switch board.

Data Encapsulation

Encapsulation is the most basic concept of OOP. It is the way of combining both data and the functions that operate on that data under a single unit. The only way to access the data is provided by the functions (that are combined along with the data). These functions are considered as member functions in C++. It is not possible

to access the data directly. If you want to reach the data item in an object, you call a member function in the object. It will read the data item and return the value to you. The data is hidden, so it is considered as safe and far away from accidental alternation. Data and its functions are said to be encapsulated into a single entity.

Modularity

The act of partitioning a program into individual components is called modularity. It gives the following benefits.

l It reduces its complexity to some extent.

l It creates a number of well-defined, documented boundaries within the program.

Module is a separate unit in itself. It can be compiled independently though it has links with other modules. Modules work quite closely in order to achieve the program's goal.

Inheritance

It is the capability to define a new class in terms of an existing class. An existing class is known as a base class and the new class is known as derived class. Number of examples can be given on this aspect. For example, a motor cycle is a class in itself. It is also a member of two wheelers class. Two wheelers class in turn is a member of automotive class as shown in Fig. 8.1. The automotive is an example of base class and two wheelers is its derived class. In simple words, we can say a motor cycle is a two wheeler automotive.

C++ supports such hierarchical classification of classes. The main benefit from inheritance is that we can build a generic base class, i.e., obtain a new class by adding some new features to an existing class and so on. Every new class defined in that way consists of features of both the classes. Inheritance allows existing classes to be adapted to new application without the need for modification.

Polymorphism

Polymorphism is a key to the power of OOP. It is the concept that supports the capability of data to be processed in more than one form. For example, an operation may exhibit different behaviour in different instances. The behaviour depends upon the types of data used in the operation. Let us consider the operation of addition. For two numbers, the operation will generate a sum. If the operands are strings then the operation would produce a third string by concatenation.

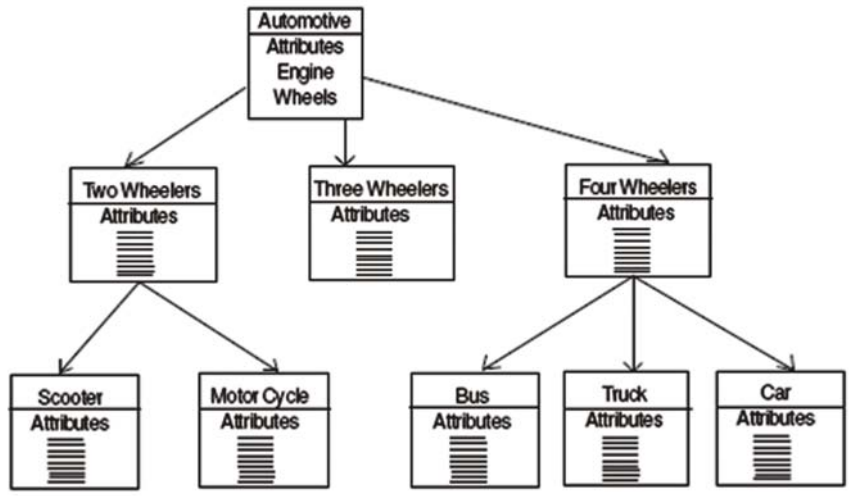


Fig.

Benefits of OOP

OOP provides lot of benefits to both the program designer and the user. Objectoriented approach helps in solving many problems related to software development and quality of software product. The new technology assures greater programmer productivity, better quality of software and lesser maintenance cost. The major benefits are :

l Software complexity can be easily managed
l Object-oriented systems can be easily upgraded
l It is quite easy to partition the work in a project based on objects.

Programming Applications of OOP

OOP has become one of the programming buzzwords today. There appears to be a great deal of excitement and interest among software programmers in using OOP. Applications of OOP are gaining importance in many areas. OOP has been extensively used in the development of windows and word based systems such as MS-Windows, x-Windows etc. The promising application areas of OOP are:

(i) Multiple data structure: This is an application where the same data structure is used many times. For example a window data structure is used multiple-times in a windowing system.

(ii) Data in multiple programs: This is an application where the same operations are performed on a data structure in different programs. For example, record validation in an accounting system.

The other application areas of OOP are parallel programming, simulation and modelling, AI and Expert systems, Neural Networks and CAD systems.

Linked Lists

A linked list contains a list of data .The Data can be anything: number, character, array,structure, etc.Each element of the list must also link with the next element therefore, a structure containing data and link is created.

The link is a pointer to the same type of structure

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

This is called a self-referential pointer

Uses and Operations on Linked Lists

Linear linked list: last element is not connected to anything
Circular linked list: last element is connected to the first
Dynamic: Size of a linked list grows or shrinks during the execution of a program and is just right
Advantage: It provides flexibility in inserting and deleting elements by just re-arranging the links
Disadvantage: Accessing a particular element is not easy
There are three major operations on linked lists

1 Insertion
2 Deletion
3 Searching

Linked list: chain of nodes

A linked list is simply a linear chain of such nodes.The beginning of the list is maintained as a pointer to the firstelement (generally called head)

Space for an element is created using a pointer (say q)

```
q = (struct Node *) malloc (size of (struct Node) );
```

q->data is the desired value
q->next is NULL
A list element's members are accessed using the pointer (q) to the list element
data using q->data
next element pointer using q->next
Moving to next element is done using pointers
q = q?next;

Linked List: element definition and creation

```
#include <stdio.h>
#include <stdlib.h>
typedef struct Node
{
int data; // data of a node: list is made of these elements
struct Node *next; // link to the next node
} node;
node *create_node(int val)
{
node *n;

n = malloc(sizeof(node));
n->data = val;
n->next = NULL;
return n;
}
```

Insertion at the beginning of the list
Create a new node (say q)
Make q->next point to head
Make head equal to q
list is empty, i.e., head is NULL
Make head equal to q

Insertion at end of list
Create a new node (say q)
Find the last element (say p)
Make p->next point to q
if list is empty, i.e., head is NULL
Make head equal to q

Deletion at the beginning of the list
Make p equal to head
Make head equal to head->next
Delete p (by using free)
If list is empty, i.e., head is NULL
Nothing to do
If list contains only one element
Delete head
head is now NULL

Deletion from the end of the list
Find the last element (say p)
While finding p, maintain q that points to p
q is the node just before p, i.e., q->next is p

Make q->next NULL
Delete p (by using free)
If list is empty, i.e., head is NULL
Nothing to do
If list contains only one element
Delete head
head is now NULL

Searching a node (insert after, delete after)
Make p equal to head
While p->data not equal to the data that is being searched, make p equal to p->next
Using search, insert after and delete after operations can be implemented
Insert after p
Create a new node q
Make q->next equal to p->next
Make p->next equal to q
)

Delete after p
Call the next node, i.e., p->next as q
Make p->next equal to q->next
Delete q
Linked List: element definition and creation

```
#include <stdio.h>
#include <stdlib.h>
typedef struct Node
{
int data; // data of a node: list is made of these elements
struct Node *next; // link to the next node
} node;
node *create_node(int val)
{
node *n;

n = malloc(sizeof(node));
n->data = val;
n->next = NULL;
return n;
}
```

Displaying the data in the linked list

```
void print_list(node *h)
{ /*Display data in each element of the linked list*/
node *p;
p = h;
while (p != NULL)
{
printf("%d --> ", p->data);
p = p->next;
}

}
```

Inserting at end

```
int main()
{
```

```
node *head = NULL; // head maintains the entry to the list
node *p = NULL, *q = NULL;
int v = -1, a;
printf("Inserting at end: Enter the data value:\n");
scanf("%d", &v);
while (v != -1)
{
q = create node(v);

create_if (head == NULL)
head = q;
else /*non empty list*/
{
p = head;
while (p->next != NULL)
p = p->next;
p->next = q;
}
scanf("%d", &v);
}

print_list(head); /*Display the data in the list*/
```

Inserting at the beginning

```
printf("Inserting at beginning\n");
scanf("%d", &v);
while (v != -1)
{
q = create_node(v);
q->next = head;
head = q;
scanf("%d", &v);
}

print_list(head); /*Display the data in the list*/
```

Inserting after an element

```
printf("Inserting after\n");
scanf("%d", &v);
while (v != -1)
{
q = create_node(v);
scanf("%d", &a);
p = head;
while ((p != NULL) && (p->data != a))
p = p->next;
if (p != NULL)
{
q->next = p->next;
p->next = q;
}
scanf("%d", &v);
}
print_list(head); /*Display the data in the list*/
```

Deleting from the end

```
printf("Deleting from end\n");
if (head != NULL)
{
p = head;
while (p->next != NULL)
{
q = p;
p = p->next;
}

q->next = NULL;
free (P);
}
Print_list(head);/*Display the data in the list */
```

Deleting from the beginning

```
printf("Deleting from beginning\n");
if (head != NULL)
{
p = head;
head = head->next;
free(p);
}
/*Empty list: i.e. head==NULL, do nothing*/
print_list(head); /*Display the data in the list*/
```

Deleting after an element

```
printf("Deleting after\n");
scanf("%d", &a);
p = head;
while ((p != NULL) && (p->data != a))
p = p->next;
if (p != NULL)
{
q = p->next;
if (q != NULL)
{
p->next = q->next;
free(q);
}
}
print_list(head); /*Display the data in the list*/
}
```

STACKS

A stack is simply a list of elements with insertions and deletions permitted at one endcalled
the stack top. That means that it is possible to remove elements from a stack in reverse order from the insertion of elements into the stack. Thus, a stack data structure exhibits the LIFO (last in first out) property. Push and pop are the operations that are provided for insertion of an element into the stack and the removal of an element from the stack, respectively.

Operations on stack:

The insertion of elements into stack is called PUSH operation.

The deletion of elements from stack is called POP operation.

POP operation:

Following actions taken place in POP:

Check the stack empty or not.

Remove the top element from the stack.

Return this element to the calling function or program.

Stack Push

stack top/head has the address of the first element

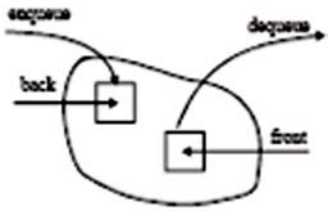
Function needs the address to the stack top/head to make changes to head

```
void push(node **head_address, int top)
{
node *q;
q = create_node(top); /*New element storing the new data*/
q->next = *head address; /*New element pointing to head*/
q >head_/ head /
*head_address = q; /*head pointing to new element*/
return;
}
```

Stack Pop

```
int pop(node **head_address)
{
node *p, *head;
int top;
head = *head_address; /*head has address of the first element*/
if (head != NULL)
{
p = head; //p: address of stack top element in stack
top = p->data; //data in stack top/first element
head = head->next; //head now has address of 2nd element in stack
free(p); //remove the first element in stack
}
else
top = -1; //-1 denotes invalid value or empty list
*head_address = head; /*reflect the changes to head outside*/
return top;
}
```

QUEUE

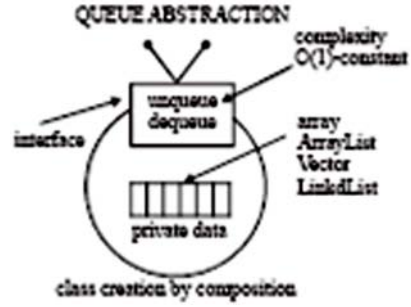


A queue is a container of objects (a linear collection) that are inserted and removed according to the first-in first-out (FIFO) principle. An excellent example of a queue is a line of students in the food court of the UC. New additions to a line made to the back of the queue, while removal (or serving) happens in the front. In the queue only two operations are allowed enqueue and dequeue. Enqueue means to insert an item into the back of the queue, dequeue means removing the front item. The picture demonstrates the FIFO access.

The difference between stacks and queues is in removing. In a stack we remove the item the most recently added; in a queue, we remove the itemthe least recently added.

Implementation

In the standard library of classes, the data type queue is an *adapter* class, meaning that a queue is built on top of other data structures. The underlying structure for a queue could be an array, a Vector, an ArrayList, a LinkedList, or any other collection. Regardless of the type
of the underlying data structure, a queue must implement the same functionality. This is achieved by providing a unique interface.

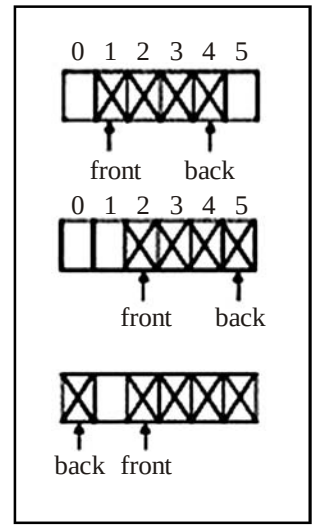



```
interface QueueInterface‹AnyType>
{
public boolean isEmpty();
public AnyType getFront();
public AnyType dequeue();
public void enqueue(AnyType e);
public void clear();
}
```

Each of the above basic operations must run at constant time O(1). The following picture demonstrates the idea of implementation by composition.

Circular Queue

Given an array A of a default size (? 1) with two references back and front, originally set to -1 and 0 respectively. Each time we insert (enqueue) a new item, we increase the back index; when we remove (dequeue) an item - we increase the front index. Here is a picture that illustrates the model after a few steps:



As you see from the picture, the queue logically moves in the array from left to right. After several moves back reaches the end, leaving no space for adding new elements However, there is a free space before the front index. We shall use that space for enqueueing new items, i.e. the next entry will be stored at index 0, then 1, until front. Such a model is called a **wrap around queue or a circular queue**

Finally, when back reaches front, the queue is full. There are two choices to handle a full queue:a) throw an exception; b) double the array size.

The circular queue implementation is done by using the modulo operator (denoted %), which is computed by taking the remainder of division (for example, 8%5 is 3). By using the modulo operator, we can view the queue as a circular array, where the "wrapped around" can be simulated as "back % array_size". In addition to the back and front indexes, we maintain another index: cur - for counting the number of elements in a queue. Having this index simplifies a logic of implementation.

APPLICATIONS

The simplest two search techniques are known as Depth-First Search(DFS) and Breadth-First Search (BFS). These two searches are described by looking at how the search tree (representing all the possible paths from the start) will be traversed.

Deapth-First Search with a Stack

In depth-first search we go down a path until we get to a dead end; then we backtrack or back up (by popping a stack) to get an alternative path.

Create a stack
Create a new choice point
Push the choice point onto the stack

- Pop the stack
- Find all possible choices after the last one tried
- Push these choices onto the stack

Return

Breadth-First Search with a Queue

In breadth-first search we explore all the nearest possibilities by finding all possible successors and enqueue them to a queue.

Create a queue
Create a new choice point
Enqueue the choice point onto the queue
while (not found and queue is not empty)

- Dequeue the queue
- Find all possible choices after the last one tried
- Enqueue these choices onto the queue

Return

Queues

Queue operations are also called First-in first-out Operations

Enqueue: insert at the end of queue

Dequeue: delete from the beginning of queue

Code: similar to previous code on linked lists

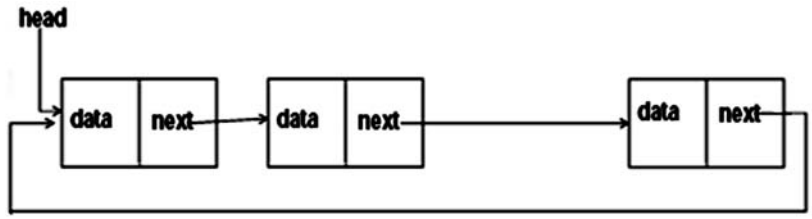
Queue Application: Executing processes by operating system

Operating System puts new processes at the end of a queue.System executes processes at the beginning of the queue

Circular Lists

The last element of a linked list points to the first element.

A reference pointer is required to access the list: head



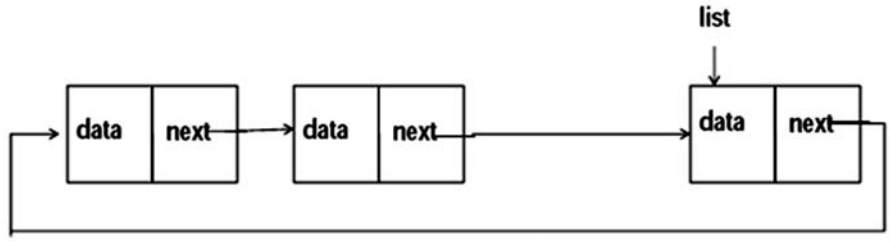
Circular Lists

The list pointer can have the address of the last element.The tail/last element can be accessed by the list pointer.The head/first element can be accessed from the tail/ lastelement (by list->next)

Provides flexibility in accessing first and last elements

Circular lists can be used for queues.

Useful in enqueue/dequeue operations without needing to traverse the list.



ARITHMETIC EXPRESSION EVALUATION

An important application of stacks is in parsing. For example, a compiler must parse arithmetic expressions written using infix notation:

1 + ((2 + 3) * 4 + 5)*6

We break the problem of parsing infix expressions into two stages. First, we convert from infix to a different representation called postfix. Then we parse the postfix expression, which is a somewhat easier problem than directly parsing infix.

Converting from Infix to Postfix. Typically, we deal with expressions in infix notation
2 + 5
where the operators (e.g. +, *) are written between the operands (e.q, 2 and 5). Writing the operators after the operands gives a postfix expression 2 and 5 are called operands, and the '+' is operator. The above arithmetic expression is called infix, since the operator is in between operands. The expression 2 5 + Writing the operators before the operands gives a prefix expression +2 5 Suppose you want to compute the cost of your shopping trip. To do so, you add a list of numbers and multiply them by the local sales tax (7.25%): 70 + 150 * 1.0725
Depending on the calculator, the answer would be either 235.95 or 230.875. To avoid this confusion we shall use a postfix notation
70 150 + 1.0725 *
Postfix has the nice property that parentheses are unnecessary.
Now, we describe how to convert from infix to postfix.

1. Read in the tokens one at a time
2. If a token is an integer, write it into the output
3. If a token is an operator, push it to the stack, if the stack is empty. If the stack is not empty, you pop entries with higher or equal priority and only then you push that token to the stack.
4. If a token is a left parentheses '(' , push it to the stack
5. If a token is a right parentheses ')', you pop entries until you meet '(' .
6. When you finish reading the string, you pop up all tokens which are left there.
7. Arithmetic precedence is in increasing order: '+', '-', '*', '/';

Evaluating a Postfix Expression. We describe how to parse and evaluate a postfix expression.

1. We read the tokens in one at a time.
2. If it is an integer, push it on the stack
3. If it is a binary operator, pop the top two elements from the stack, apply the operator, and push the result back on the stack.

Consider the following postfix expression
5 9 3 + 4 2 * * 7 + *
Here is a chain of operations

Stack Operations Output

push(5); 5
push(9); 5 9
push(3); 5 9 3
push(pop( ) + pop( )) 5 12
push(4); 5 12 4
push(2); 5 12 4 2
push(pop( ) * pop( )) 5 12 8
push(pop( ) * pop( )) 5 96
push(7) 5 96 7
push(pop( ) + pop( )) 5 103
push(pop( ) * pop( )) 515
Note, that division is not a commutative operation, so 2/3 is not the same as 3/2.

Suppose we have an infix expression:2+(4+3*2+1)/3. We read the string by characters.
'2' - send to the output.
'+' - push on the stack.
'(' - push on the stack.
'4' - send to the output.
'+' - push on the stack.
'3' - send to the output.
'*' - push on the stack.
'2' - send to the output.

TREES

A **tree** is a data structure that is made of nodes and pointers, much like a linked list. The difference between them lies in how they are organized:

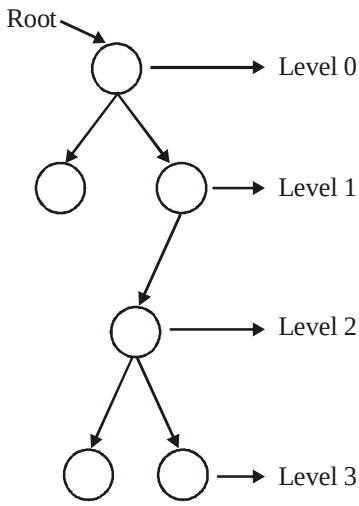
In a linked list each node is connected to one "successor" node (via next pointer), that is, it is linear.

In a tree, the nodes can have several next pointers and thus are not linear.

The top node in the tree is called the **root** and all other nodes branch off from this one.

Every node in the tree can have some number of children. Each child node can in turn be the parent node to its children and so on.

A common example of a tree structure is the binary tree.



A **binary tree** is a tree that is limited such that each node has only two children.

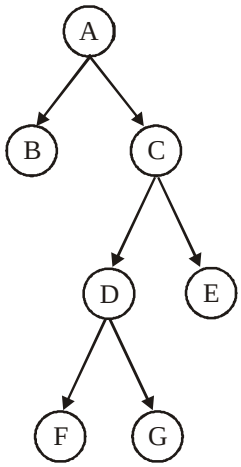
If n1 is the root of a binary tree and n2 is the root of its left or right tree, then n1 is the **parent** of n2 and n2 is the **left** or **right child** of n1.

A node that has no children is called a **leaf**.

The nodes are **siblings** if they are left and right children of the same parent.

The level of a node in a binary tree:

- The root of the tree has level 0
- The level of any other node in the tree is one more than the level of its parent.



Types of Binary Tree:-

1. Strictly Binary Tree

- If every non leaf node in a binary tree has non empty left and right sub trees, the tree is termed as strictly binary tree

Every non leaf node has degree 2.

A strictly binary tree with n leaves has (2n-1) nodes.

Thus a strictly binary tree has odd number of nodes.

2. Complete Binary Tree

A complete binary tree of depth d is the binary tree of depth d that contains exactly 2^l at each level l between 0 and d.

Thus the total number of nodes in complete binary tree are 2^{d+1}-1 where leaf nodes are 2^d and non leaf are 2^d-1.

Because a complete binary tree is also a strictly binary tree , thus if it has n leaves then it has 2n-1 nodes. Also follows from this is the previous assertion that if $n=2^d$ then total nodes are

$$2*2^d-1=2^{d+1}-1$$

3. Atmost complete Binary Tree

A binary tree of depth d is an almost complete binary tree if:

- At any node in the tree with a right descendent at level d, node must have a left son and every left descendent of node is either a leaf at level d or has two sons.

  i.e. the tree must be left filled.

Note : ACBT property says that if there are n nodes in the tree then leaf node have numbers

[(n/2)+1][(n/2)+2]———n

ACBT with n leaves has 2n nodes if it is not a SBT

An ACBT which is also an SBT has 2n-1 nodes for n leaves

Also an ACBT is an SBT if the number of node are odd

An ACBT of depth d is intermediate between the complete binary tree of depth, t d-1 that contains 2^{d-1} nodes , and the complete binary tree of depth d, which contains 2^{d+1}-1 nodes

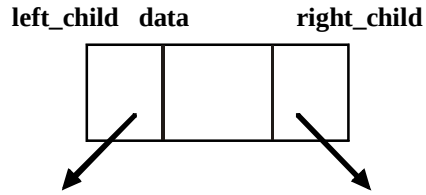
Implementation

A binary tree has a natural implementation in linked storage. A separate pointer is used to point the tree (e.g. root)

```
root = NULL;                    // empty tree
```

Each node of a binary tree has both left and right subtrees which can be reached with pointers:

```
struct tree_node{
int data;
struct tree_node *left_child;
struct tree_node *right_child;
};
```



Traversal of Binary Trees:

Linked lists are traversed from first to last sequentially. However, there is no such natural linear order for the nodes of a tree. Different orderings are possible for traversing a binary tree. Three of these traversal orderings are:

Preorder traversal (also known as depth – first order)
Inorder traversal (a.k.a. symmetric order)
Postorder traversal (a.k.a end order)

These names are chosen according to the step at which the root node is visited.

- With **preorder** traversal the node is visited *before* its left and right subtrees,
- With **inorder** traversal the root is visited *between* the subtrees,
- With **postorder** traversal the root is visited *after* both subtrees.

Binary Tree Traversal >> Preorder Traversal

Preorder traversal of a binary tree consists of following three recursive operations.

a. Visit the root.
b. Traverse the left sub-tree in preorder.
c. Traverse the left sub-tree in preorder.

```
void doPreOrder(nodeptr &tree){
nodeptr p = tree;
if(p!= null){
printf("%d", p->key);
doPreOrder(p->left);
doPreOrder(p->right);
}
}
```

Binary Tree Traversal >> Inorder Traversal

Inorder traversal of a binary tree consists of following three recursive operations.

a. Traverse the left sub-tree in inorder.
b. Visit the root.
c. Traverse the left sub-tree in inorder.

```
void doInOrder(nodeptr &tree){
nodeptr p = tree;
if(p!= null){
doInOrder(p->left);
printf("%d", p->key);
doInOrder(p->right);
}
}
```

Binary Tree Traversal >> Postorder Traversal

Postorder traversal of a binary tree consists of following three recursive operations.

a. Traverse the left sub-tree in postorder.
b. Traverse the left sub-tree in postorder.
c. Visit the root.

```
void doPostOrder(nodeptr &tree){
nodeptr p = tree;
if(p!= null){
doPostOrder(p->left);
doPostOrder(p->right);
printf("%d", p->key);
}
}
```

BINARY SEARCH TREE:-

BST is a binary tree. For each node in a BST, the left subtree is smaller than it; and the right subtree is greater than it. A **binary search tree** (**BST**), sometimes also called an **ordered** or **sorted binary tree**, is a node-based binary tree data structure which has the following properties:

The left subtree of a node contains only nodes with keys less than the node's key.

The right subtree of a node contains only nodes with keys greater than the node's key.

The left and right subtree must each also be a binary search tree.

There must be no duplicate nodes.

Implementation

We implement a binary search tree using a private inner class BSTNode. In order to support the *binary search tree property*, we require that data stored in each node is Comparable:

```
public class BST <AnyType extends Comparable<AnyType>>
{
   private Node<AnyType> root;

   private class Node<AnyType>
   {
      private AnyType data;
      private Node<AnyType> left, right;

      public Node(AnyType data)
      {
         left = right = null;
         this.data = data;
      }
   }
   ...
```

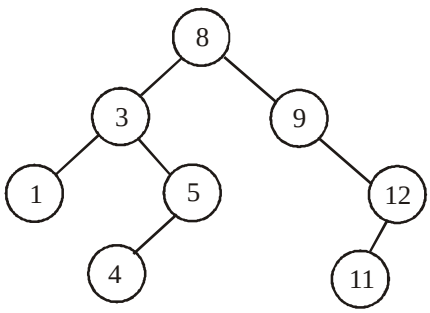
Insertion

The insertion procedure is quite similar to searching. We start at the root and recursively go down the tree searching for a location in a BST to insert a new node. If the element to be inserted is already in the tree, we are done (we do not insert duplicates). The new node will always replace a NULL reference.

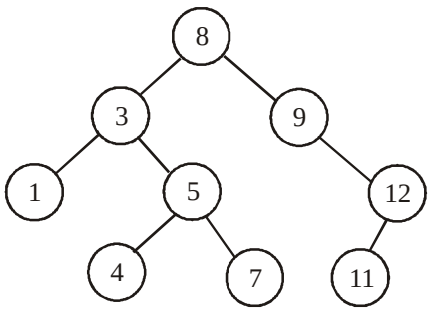
Example: Given a sequence of numbers:

11, 6, 8, 19, 4, 10, 5, 17, 43, 49, 31

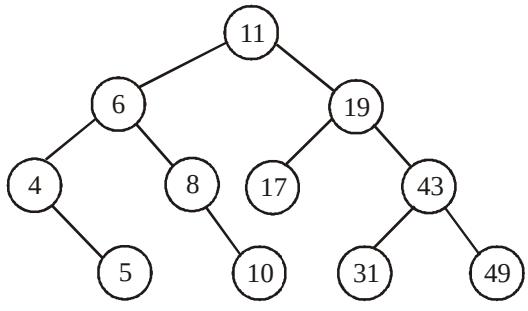
Draw a binary search tree by inserting the above numbers from left to right.



before insertion



after insertion



Searching

Searching in a BST always starts at the root. We compare a data stored at the root with the key we are searching for (let us call it as **to Search**). If the node does not contain the key we proceed either to the left or right child depending upon comparison. If the result of comparison is negative we go to the left child, otherwise - to the right child. The recursive structure of a BST yields a recursive algorithm.

Deletion

Deletion is somewhat more tricky than insertion. There are several cases to consider. A node to be deleted (let us call it as **to Delete**)

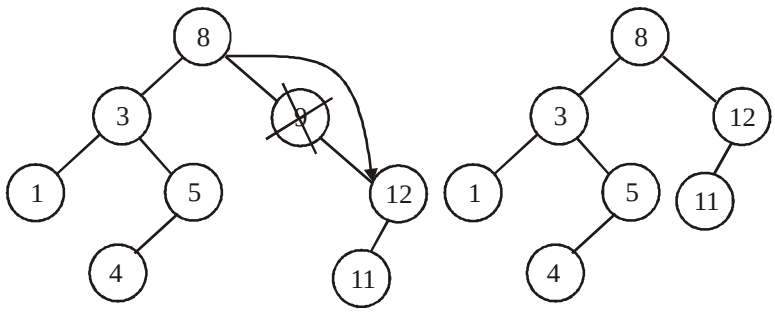
is not in a tree;

is a leaf;

has only one child;

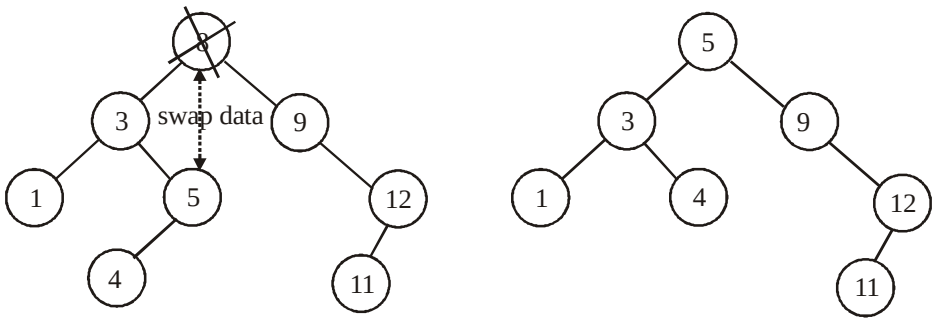
has two children.

If **to Delete** is not in the tree, there is nothing to delete. If **to Delete** node has only one child the procedure of deletion is identical to deleting a node from a linked list - we just bypass that node being deleted



before deletion after deletion

Deletion of an internal node with two children is less straightforward. If we delete such a node, we split a tree into two subtrees and therefore, some children of the internal node won't be accessible after deletion. In the picture below we delete 8:



before deletion after deletion

Deletion starategy is the following: replace the node being deleted with the largest node in the left subtree and then delete that largest node. By symmetry, the node being deleted can be swapped with the smallest node is the right subtree.

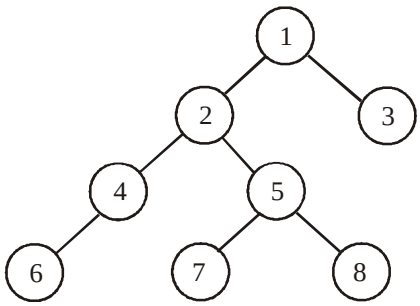
Non-Recursive Traversals

Depth-first traversals can be easily implemented recursively.A non-recursive implementation is a bit more difficult. In this section we implement a pre-order traversal as a tree iterator

```
public Iterator<AnyType> iterator()
{
  return new PreOrderIterator();
}
```

Example

```
public AnyType next()
{
   Node cur = stk.peek();
  if(cur.left != null)
   {
      stk.push(cur.left);
   }
   else
   {
      Node tmp = stk.pop();
     while(tmp.right == null)
      {
if (stk.isEmpty()) return cur.data;
         tmp = stk.pop();
      }
     stk.push(tmp.right);
   }
   return cur.data;
}
```



where the PreOrderIterator class is implemented as an inner private class of the BST class

```
private class PreOrderIterator implements Iterator<AnyType>
{
  ...
}
```

The main difficulty is with next() method, which requires the implicit recursive stack implemented explicitly. We will be using Java's Stack. The algorithm starts with the root and push it on a stack. When a user calls for the **next**() method, we check if the top element has a left child. If it has a left child, we push that child on a stack and return a parent node. If there is no a left child, we check for a right child. If it has a right child, we push that child on a stack and return a parent node. If there is no right child, we move back up the tree (by popping up elements from a stack) until we find a node with a right child. Here is the **next**() implementation as shown in example

This example showed the output and the state of the stack during each call to **next**(). Note, the algorithm works on any binary trees, not necessarily binary search trees..

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

A non-recursive preorder traversal can be eloquently implemented in just three lines of code. If you understand next()'s implementation above, it should be no problem to grasp this one:

```
public AnyType next()
{
  if (stk.isEmpty()) throw new java.util.NoSuchElementException();
   Node cur = stk.pop();
  if(cur.right != null) stk.push(cur.right);
  if(cur.left != null) stk.push(cur.left);
   return cur.data;
}
```

Note, we push the right child before the left child.

Level Order Traversal

Level order traversal processes the nodes level by level. It first processes the root, and then its children, then its grandchildren, and so on. Unlike the other traversal methods, a recursive version does not exist.

A traversal algorithm is similar to the non-recursive preorder traversal algorithm. The only difference is that a stack is replaced with a FIFO queue.

BINARY HEAPS

A binary tree is **completely full** if it is of height, *h*, and has 2^{h+1}-1 nodes.
A binary tree of height, *h*, is **complete** *iff*

a. it is empty *or*
b. its left sub tree is complete of height *h*-1 and its right sub tree is completely full of height *h*-2 *or*
c. Its left sub tree is completely full of height *h*-1 and its right sub tree is complete of height *h*-1.

A complete tree is filled from the left:

all the leaves are on

- the same level *or*
- two adjacent ones *and*

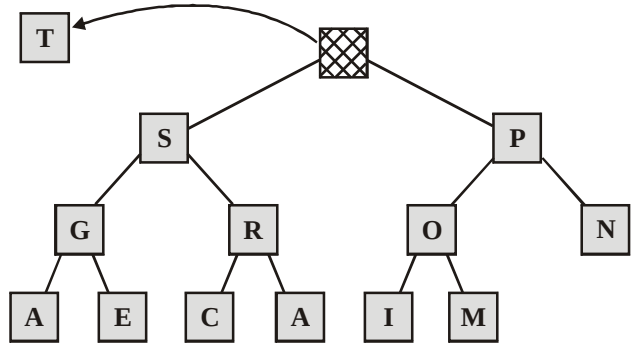
All nodes at the lowest level are as far to the left as possible.

Heaps

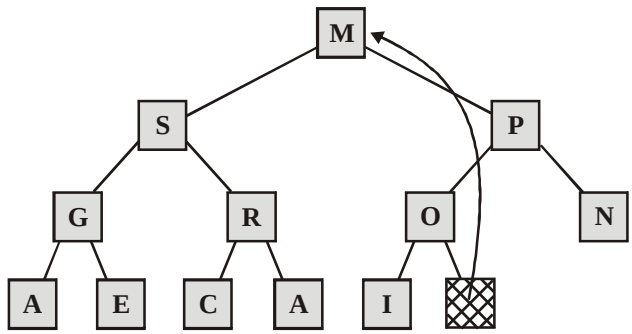
A binary tree has the **heap property** *iff*

a. it is empty *or*
b. The key in the root is larger than that in either child and both sub trees have the heap property.

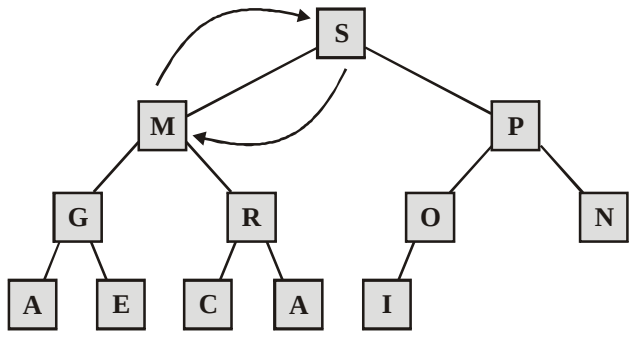
A heap can be used as a priority queue: the highest priority item is at the root and is trivially extracted. But if the root is deleted, we are left with two sub-trees and we must *efficiently* re-create a single tree with the heap property.



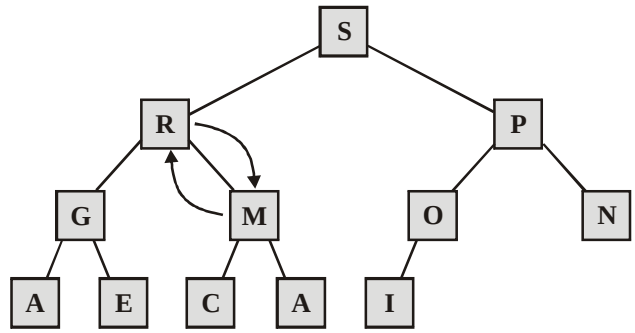
The value of the heap structure is that we can both extract the highest priority item and insert a new one in **O(log*n*)** time.
How do we do this?
Let's start with this heap.
A deletion will remove the T at the root.
To work out how we're going to maintain the heap property, use the fact that a complete tree is filled from the left. So that the position which must become empty is the one occupied by the M. Put it in the vacant root position.



This has violated the condition that the root must be greater than each of its children. So interchange the M with the larger of its children.



The left subtree has now lost the heap property.
So again interchange the M with the larger of its children.



This tree is now a heap again, so we're finished.
We need to make at most *h* interchanges of a root of a subtree with one of its children to fully restore the heap property. Thus deletion from a heap is **O(*h*)** or **O(log*n*)**.

Heap operations :

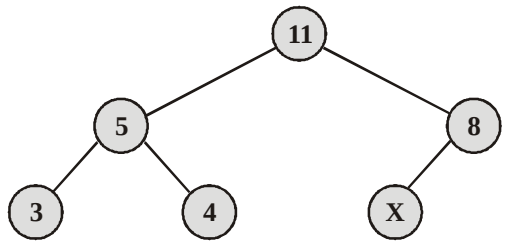
Both the insert and remove operations modify the heap to conform to the shape property first, by adding or removing from the end of the heap. Then the heap property is restored by traversing up or down the heap. Both operations take O(log *n*) time.

Insert :

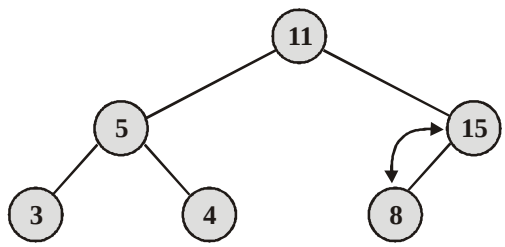
To add an element to a heap we must perform an *up-heap* operation (also known as *bubble-up*, *percolate-up*, *sift-up*, *trickle up*, *heapify-up*, or *cascade-up*), by following this algorithm:

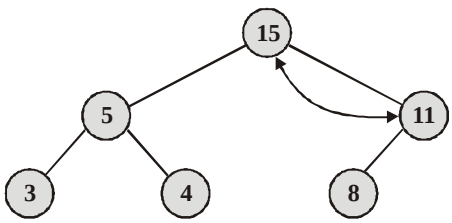
1. Add the element to the bottom level of the heap.
2. Compare the added element with its parent; if they are in the correct order, stop.
3. If not, swap the element with its parent and return to the previous step.

The number of operations required is dependent on the number of levels the new element must rise to satisfy the heap property, thus the insertion operation has a time complexity of O(log *n*).
As an example, say we have a max-heap



and we want to add the number 15 to the heap. We first place the 15 in the position marked by the X. However, the heap property is violated since 15 is greater than 8, so we need to swap the 15 and the 8. So, we have the heap looking as follows after the first swap:





However the heap property is still violated since 15 is greater than 11, so we need to swap again:

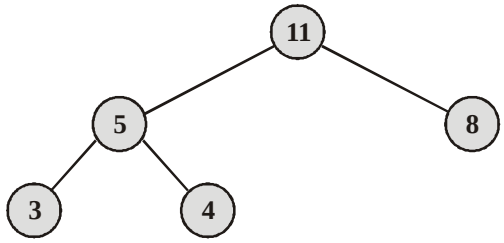
which is a valid max-heap. There is no need to check the children after this. Before we placed 15 on X, the heap was valid, meaning 11 is greater than 5. If 15 is greater than 11, and 11 is greater than 5, then 15 must be greater than 5, because of the transitive relation.

Delete :

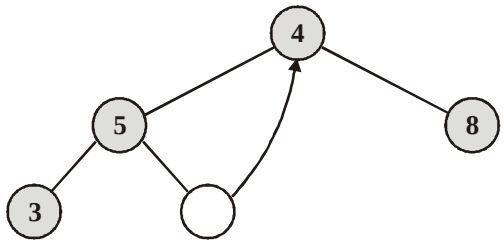
The procedure for deleting the root from the heap (effectively extracting the maximum element in a max-heap or the minimum element in a min-heap) and restoring the properties is called *down-heap* (also known as *bubble-down*, *percolate-down*, *sift-down*, *trickle down*, *heapify-down*, *cascade-down* and *extract-min/max*).

1. Replace the root of the heap with the last element on the last level.
2. Compare the new root with its children; if they are in the correct order, stop.
3. If not, swap the element with one of its children and return to the previous step. (Swap with its smaller child in a min-heap and its larger child in a max-heap.)

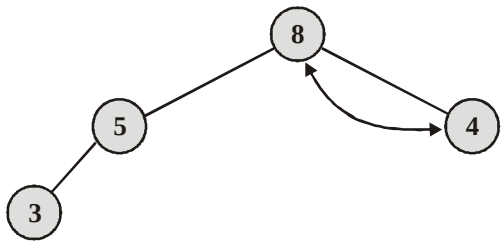
So, if we have the same max-heap as before



We remove the 11 and replace it with the 4.



Now the heap property is violated since 8 is greater than 4. In this case, swapping the two elements, 4 and 8, is enough to restore the heap property and we need not swap elements further:



The downward-moving node is swapped with the *larger* of its children in a max-heap (in a min-heap it would be swapped with its smaller child), until it satisfies the heap property in its new position. This functionality is achieved by the **Max-Heapify** function as defined below in **pseudo code** for an array-backed heap *A* of length *heap_length*[*A*]. Note that "A" is indexed starting at 1, not 0 as is common in many real programming languages.

Max-Heapify (*A*, *i*):
left ← 2*i*
right ← 2*i* + 1
largest ← *i*
if *left* ≤ *heap_length*[*A*] **and** *A*[*left*] > A[*largest*] **then**:
largest ← *left*
if *right* ≤ *heap_length*[*A*] **and** *A*[*right*] > *A*[*largest*] **then**:
largest ← *right*
if *largest* ≠ *i* **then**:
swap *A*[i] ↔ A[largest]
Max-Heapify(*A*, *largest*)

For the above algorithm to correctly re-heapify the array, the node at index *i* and its two direct children must violate the heap property. If they do not, the algorithm will fall through with no change to the array. The down-heap operation (without the preceding swap) can also be used to modify the value of the root, even when an element is not being deleted.

In the worst case, the new root has to be swapped with its child on each level until it reaches the bottom level of the heap, meaning that the delete operation has a time complexity relative to the height of the tree, or $O(\log n)$.

Building a heap :

A heap could be built by successive insertions. This approach requires $O(n \log n)$ time because each insertion takes $O(\log n)$ time and there are n elements. However this is not the optimal method. The optimal method starts by arbitrarily putting the elements on a binary tree, respecting the shape property (the tree could be represented by an array, see below). Then starting from the lowest level and moving upwards, shift the root of each subtree downward as in the deletion algorithm until the heap property is restored. More specifically if all the subtrees starting at some height *h* (measured from the bottom) have already been "heapified", the trees at height $h + 1$ can be heapified by sending their root down along the path of maximum valued children when building a max-heap, or minimum valued children when building a min-heap. This process takes $O(h)$ operations (swaps) per node. In this method most of the heapification takes place in the lower levels. Since the height of the heap is $[\lg(n)]$, the number of nodes at height is *h*.

$$\leq [2^{(\lg n - h)-1}] = \left\lceil \frac{2^{\lg n}}{2^{h+1}} \right\rceil = \left\lceil \frac{n}{2^{h+1}} \right\rceil$$

Therefore, the cost of heapifying all subtrees is:

$$\sum_{h=0}^{[\lg n]} \frac{n}{2^{h+1}} O(\mathrm{h}) = O\left(n \sum_{h=0}^{\lg n} \frac{h}{2^{h+1}} \right)$$

$$\leq O\left(n \sum_{h=0}^{\infty} \frac{h}{2^{h}} \right)$$

$$= O(n)$$

This uses the fact that the given infinite series $h / 2^h$ converges to 2.

The exact value of the above (the worst-case number of comparisons during the heap construction) is known to be equal to:

$$2\mathrm{n} - 2\mathrm{s}_2(n) - \mathrm{e}_2(n),$$

where $\mathrm{s}_2(\mathrm{n})$ is the sum of all digits of the binary representation of n and $\mathrm{e}_2(\mathrm{n})$ is the exponent of 2 in the prime factorization of n.

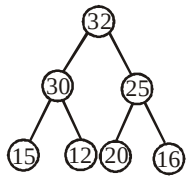
Past GATE Questions Exercise

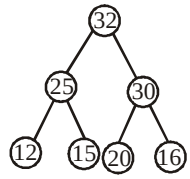
1. Assume that the operators +, –, ×, are left associative and ^ is right associative. The order of precedence (from highest to lowest) is ^, ×, +, –. The postfix expression corresponding to the infix expression a + b × c – d ^ e ^ f is **[2005, 2 marks]**

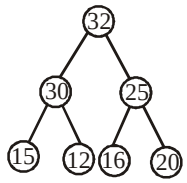
(a) abc × + def ^^– (b) abc × + de^f^–

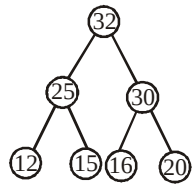
(c) ab + c × d – e^f^ (d) – + a × bc ^^ def

2. The elements 32, 15, 20, 30, 12, 25, 16, are inserted one by one in the given order into a max-heap. The resultant max-heap is **[2005, 2 marks]**

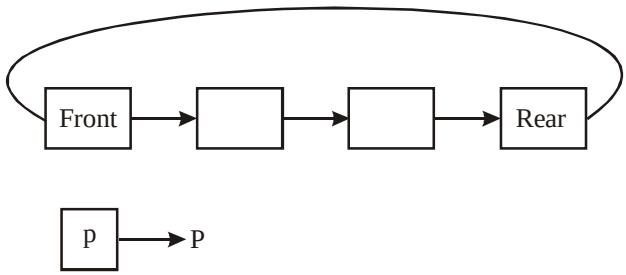
(a) 

(b) 

(c) 

(d) 

3. A circularly linked list is used to represent a queue. A single variable p is used to access the queue. To which node should p point such that both the operations en-queue and de-queue can be performed in constant time? **[2005, 2 marks]**



(a) Rear node

(b) Front node

(c) Not possible with a single pointer (d)

(d) Node next to front

4. Consider the following C program segment: **[2005, 2 marks]**

```
char p [20]
char *s = "string";
int length = strlen (s);
for (i = 0; i < length, i++)
 p[i] = s [length – i];
print f("%s", p);
```

The output of the program is

(a) gnirts (b) string

(c) gnirt (d) no output is printed

5. Consider the following C function: **[2005, 2 marks]**

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

The value returned by f(1) is

(a) 5 (b) 6

(c) 7 (d) 8

6. Postorder traversal of a given binary search tree, T produces the following sequence of keys

10, 9, 23, 22, 27, 25, 15, 50, 95, 60, 40, 29

Which one of the following sequences of keys can be the result of an in-order traversal of the tree T? **[2005, 2 marks]**

(a) 9, 10, 15, 22, 23, 25, 27, 29, 40, 50, 60, 95

(b) 9, 10, 15, 22, 40, 50, 60, 95, 23, 25, 27, 29

(c) 29, 15, 9, 10, 25, 22, 23, 27, 40, 60, 50, 95

(d) 95, 50, 60, 40, 27, 23, 22, 25, 10, 9, 15, 29

7. Consider the following C-program **[2005, 2 marks]**

```
double foo (double);          /* Line 1 */

int main () {
      double da, db;
      // input da
      db = foo (da);
}
      double foo (double a) {
      return a;
```

The above code compiled without any error or warning. If Line 1 is deleted, the above code will show

(a) no compile warning or error

(b) some compiler-warnings not leading to unintended results

(c) some compiler-warnings due to type-mismatch eventually leading to unintended results

(d) compiler errors

8. Consider the following C-program: **[2005, 2 marks]**

```
void foo (int n, int sum) {
  int k = 0,  j = 0
  if (n = = 0) return;
  k = n% 10, j = n/10;
  sum = sum + k;
  foo (j, sum);
  printf ("%d", k);
}
int main () {
  int a = 2048, sum = 0;
  foo (a, sum);
  printf ("%d\n", sum);
```

What does the above program print?

(a) 8, 4, 0, 2, 14 (b) 8, 4, 0, 2, 0

(b) 2, 0, 4, 8, 14 (d) 2, 0, 4, 8, 0

9. A program P reads in 500 integers in the range [0, 100] representing the scores of 500 students. It then prints the frequency of each score above 50. What would be the best way for P to store the frequencies? **[2005, 1 mark]**

(a) An array of 50 numbers

(b) An array of 100 numbers

(c) An array of 500 numbers

(d) A dynamically allocated array of 550 numbers

10. Which one of the following are essential features of an object-oriented programming language? **[2005, 1 mark]**

1. Abstraction and encapsulation
2. Strictly - typedness
3. Type-safe property coupled with sub-type rule
4. Polymorphism in the presence of inheritance

(a) 1 and 2 (b) 1 and 4

(c) 1, 2 and 4 (d) 1, 3 and 4

11. A common property of logic programming languages and functional languages is **[2005, 1 mark]**

(a) both are procedural languages

(b) both are based on λ-calculus

(c) both are declarative

(d) both use Horn-clauses

12. An Abstract Data Type (ADT) is **[2005, 1 mark]**

(a) same as an abstract class

(b) a data type that cannot be instantiated

(c) a data type for which only the operations defined on it can be used, but none else

(d) All of the above

13. What does the following C-statement declare?
int (*f) int*); **[2005, 1 mark]**

(a) A function that takes an integer pointer as argument and returns an integer

(b) A function that takes as argument and returns an integer pointer

(c) A pointer to a function that takes an integer pointer as argument and returns an integer

(d) A function that takes an integer pointer as argument and returns a function pointer

14. Consider this code to swap integers and these five statements:
The code **[2006, 2 marks]**

```
void swap (int *px, int *py) {
    *px = *px – *py;
    *py = *px + *py;
    *px = *py – *px;
}
```

S1 : will generate a compilation error

S2 : may generate a segmentation fault by runtime depending on the arguments passed

S3 : correctly implements the swap procedure for all input pointers referring to integers stored in memory locations accessible to the process

S4: implements the swap procedure correctly for some but not all valid input pointers

S5 : may add or subtract integers and pointers

(a) S1 only (b) S2 and S3

(c) S2 and S4 (d) S2 and S5

15. Consider these two functions and two statements S1 and S2 about them. **[2006, 2 marks]**

```
int work1 (int*a, int i, int j)
{
    int x = a[i + 2];
    a[j] = x + 1;
    return a[i + 2] – 3;
}
```

```
int work2 {int *a, int i, int j)
{
    int t1 = i + 2
    int t2 = a[t1]
    a[j] = t2 + 1;
    return t2 – 3;
}
```

S1: The transformation form work1 to work2 is valid, i.e., for any program state and input arguments, work2 will compute the same output and have the same effect on program state as work 1.

S2: All the transformations applied to work1 to get work2 will always improve the performance (i.e., reduce CPU time) of work2 compared to work1

(a) S1 is false and S2 is false
(b) S1 is false and S2 is true
(c) S1 is true and S2 is false
(d) S1 is true and S2 is true

16. Consider the following C-function in which a[n] and b[m] are two sorted integer arrays and c[n + m] be another array. **[2006, 2 marks]**

```
void xyz (int a [ ], int b [ ], int c [ ])
    {
    int i, j, k;
    i = j = k = 0;
    while ((i < n) && (j < m))
      if (a [i] < b [j] c [k++] = a[i++];
      else c[k++] = b[j++];
    }
```

Which of the following conditions hold(s) after the termination of the while loop?

(i) $j < m, k = n + j - 1$ and $a[n-1] < b[j]$, if $i = n$
(ii) $i < n, k = m + i - 1$ and $b[m-1] \le a[i]$, if $j = m$
(a) only (i)
(b) only (ii)
(c) either (i) or (ii) but not both
(d) neither (i) nor (ii)

17. An implementation of a queue Q, using two stacks S1 and S2 is given below **[2006, 2 marks]**

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

Let n insert and m(≤ n) delete operations be performed in an arbitrary order on an empty queue Q. Let x and y be the number of push and pop operations performed respectively in the process. Which one of the following is true for all m and n?

(a) $n + m \le x < 2n$ and $2m \le y \le n + m$
(b) $n + m \le x \le 2n$ and $2m \le y \le 2n$
(c) $2m \le x < 2n$ and $2m \le y \le n + m$
(d) $2m \le x < 2n$ and $2m \le y \le 2n$

18. Consider the following C program segment where CellNode represents a node in a binary tree: **[2007, 2 marks]**

```
struct Cell Node {
    struct Cell Node *leftChild;
      int element;
    struct CellNode *rightChild;
}
int GetValue (struct CellNode *ptr) {
int value = 0
if (ptr ! = NULL)
      if ((ptr - > leftChild = = NULL) &&
        (ptr -> rightChild == NULL){
      value = 1;
else
      value = value + GetValue (ptr -> leftChild)
                  + GetValue (ptr -> rightChild);
    }
return (value);
}
```

The value returned by GetValue when a pointer to the root of a binary tree is passed as its argument is

(a) the number of nodes in the tree
(b) the number of internal nodes in the tree
(c) the number of leaf nodes in the tree
(d) the height of the tree

19. The inorder and preorder traversal of a binary tree are d b e a f c g and a b d e c f g, respectively
The postorder traversal of the binary tree is **[2007, 2 marks]**

(a) d e b f g c a (b) e d b g f c a
(c) e d b f g c a (d) d e f g b c a

20. Consider the following C function: **[2007, 2 marks]**

```
    int f (int n)
    {static int r = 0;
    if (n < = 0) return 1;
      if (n > 3)
        {r = n;
        return f (n – 2) + 2;
        }
    return f(n – 1) + r;
}
```

What is the value of f(5)?

(a) 5 (b) 7
(c) 9 (d) 18

21. The following postfix expression with single digit operands is evaluated using a stack: **[2007, 2 marks]**

8 2 3 ^ / 2 3 * + 5 1 * -

Note that ^ is the exponentiation operator. The top two elements of the stack after the first * is evaluated are

(a) 6, 1 (b) 5, 7
(c) 3, 2 (d) 1, 5

22. Consider the following segment of C-code **[2007, 1 mark]**

```
int j, n;
    j = 1;
    while (j < = n)
                    j = j*2;
```

The number of comparisons made in the execution of the loop for any n > 0 is

(a) $[\log_2 n] + 1$ (b) n
(a) $[\log_2 n]$ (b) $[\log_2 n] + 1$

23. The following C function takes a single-linked list of integers as a parameter and rearranges the elements of the list. The function is called with the list containing the integers 1, 2, 3, 4, 5, 6, 7 in the given order. What will be the contents of the list after the function completes execution? **[2008, 2 marks]**

```
struct node {
        int value;
        struct node *next;
};
void rearrange (struct node *list){
        struct node *p, *q;
        int temp;
        if (!list || !list -> next) return;
        p = list, q = list –> next;
        while (q){
        temp = p - > value; p -> value = q - > value;
        q - > value = temp; p = q - > next;
        q = p?p -> next: 0;
        }
        }
```

(a) 1, 2, 3, 4, 5, 6, 7 (b) 2, 1, 4, 3, 6, 5, 7
(c) 1, 3, 2, 5, 4, 7, 6 (d) 2, 3, 4, 5, 6, 7, 1

24. Choose the correct option to fill ? 1 and ?2 so that the program below prints an input string in reverse order. Assume that the input string is terminated by a newline character. **[2008, 2 marks]**

```
void recerse (void){
        int c;
        if (?1) reverse ();
        ?2
}
main ( ) {
        printf("Enter Text"); printf("/n");
        reverse ( ); printf("/n")
```

(a) ?1 is (getchar ( )! = '\n')
?2 is getchar (c);
(b) ?1 is (c = getchar ( ))! = '\n')
?2 is getchar (c);
(c) ?1 is (c! = '\n')
?2 is putchar (c);
(d) ?1 is (c = getchar ( ))! = '\n')
?2 is putchar (c);

25. What is printed by the following C program? **[2008, 2 marks]**

```
int f (int x, int * py, int ** ppz)
{
        int y, z;
        **ppz + = 1; z = *ppz;
        *py + = 2; y = *py;
        x + = 3;
        return x + y + z;
}
void main ()
{
        int c, *b, **a,
        c = 4; b & c; a = & b
        printf("%d", f(c, b, a));
}
```

(a) 18 (b) 19
(c) 21 (d) 22

26. Which combination of the integer variables x, y and z makes the variable a get the value 4 in the following expression?

a = (x > y)? ((x > z)? x : z) : ((y > z)? y : z) **[2008, 1 mark]**

(a) x = 3, y = 4, z = 2
(b) x = 6, y = 5, z = 3
(c) x = 6, y = 3, z = 5
(d) x = 5, y = 4, z = 5

Statements for Linked Answer Questions 27, 28 and 29:

Consider a binary max-heap implemented using an array:

27. Which one of the following array represents a binary max-heap? **[2009, 1 mark]**

(a) {25, 12, 16, 13, 10, 8, 14}
(b) {25, 14, 13, 16, 10, 8, 12}
(c) {25, 14, 16, 13, 10, 8, 12}
(d) {25, 14, 12, 13, 10, 8, 16}

28. What is the content of the array after two delete operations on the correct answer to the previous question? **[2009, 1 mark]**

(a) {14, 13, 12, 10, 8} (b) {14, 12, 13, 8, 10}
(c) {14, 13, 8, 12, 10} (d) {14, 13, 12, 8, 10}

29. The keys 12, 18, 13, 2, 3, 23, 5 and 15 are inserted into an initially empty hash table of length 10 using open addressing with hash function h(k) = k mod 10 and linear probing. What is the resultant hash table? **[2009, 1 mark]**

(a)

0	
1	
2	2
3	23
4	
5	15
6	
7	
8	18
9	

(b)

0	
1	
2	12
3	13
4	
5	5
6	
7	
8	18
9	

(c)

0	
1	
2	12
3	13
4	2
5	3
6	23
7	5
8	18
9	15

(d)

0	
1	
2	12, 2
3	13, 3, 23
4	
5	5, 15
6	
7	
8	18
9	

30. What is the maximum height of any AVL-tree with 7 nodes? Assume that the height of a tree with a single node is 0. **[2009, 1 mark]**

(a) 2 (b) 3
(c) 4 (d) 5

31. Consider the program below: **[2009, 1 mark]**

```
#include < stdio.h >
int fun (int n, int * f_p){
        int t, f;
if (n < = 1) {
        *f_p = 1
        return 1;
        }
        t = fun (n – 1, *f_p);
        f = t + *f_p;
        *f_p = t;
        return f;
}
        int main () {
          int x = 15;
          printf (“% d\n”, fun (5, & x));
          return 0;
}
```

The value printed is

(a) 6 (b) 8
(c) 14 (d) 15

Statements for Linked Answer Questions 32 and 33 :

A hash table of length 10 uses open addressing with hash function h(k) = k mod 10, and linear probing. After inserting 6 values into an empty has table, the table is as shown below.

0	
1	
2	42
3	23
4	34
5	52
6	46
7	33
8	
9	

32. Which one of the following choices gives a possible order in which the key values could have been inserted in the table? **[2010, 2 marks]**

(a) 46, 42, 34, 52, 23, 33
(b) 34, 42, 23, 52, 33, 46
(c) 46, 34, 42, 23, 52, 33
(d) 42, 46, 33, 23, 34, 52

33. How many different insertion sequences of the key values using the same hash function and linear probing will result in the hash table shown above? **[2010, 2 marks]**

(a) 10 (b) 20
(c) 30 (d) 40

34. The following program is to be tested for statement coverage. **[2010, 2 marks]**

```
begin
if (a = = b) {S1; exit;}
else, if (c = = d) {S2;}
else {S3; exit;}
S4;
end
```

The test cases T_1, T_2, T_3 and T_4 given below are expressed in terms of the properties satisfied by the values of variables a, b, c and d. The exact values are not given.

T_1: a, b, c and d are all equal

T_2: a, b, c and d are all distinct

T_3: a = b and c! = d

T_4: a! = b and c = d

Which of the test suites given below ensures coverage of statements S_1, S_2, S_3 and S_4?

(a) T_1, T_2, T_3 (b) T_2, T_4
(c) T_3, T_4 (d) T_1, T_2, T_4

35. The program below uses six temporary variables a, b, c, d, e, f. **[2010, 2 marks]**

```
a = 1
b = 10
c = 20
d = a + b
e = c + d
f = c + e
b = c + e
e = b + f
d = 5 + e
return d + f
```

Assuming that all operations take their operands from registers, what is the minimum number of registers needed to execute this program without spilling?

(a) 2 (b) 3
(c) 4 (d) 6

36. The following C function takes a simply-linked list as input argument. It modifies the list as input argument. It modifies the list by moving the last element to the front of the list and returns the modified list. Some part of the code is left blank. **[2010, 2 marks]**

```
type def struct node {
        int value;
        struct node *next;
}       Node*;
Node *move_to_front (Node *head) {
        Node *p, *q;
if (head = = NULL | | (head -> next = = NULL)) return head;
        q = NULL; p = head;
        while (p - > next ! = NULL)     {
        q = p;
```

```
        p = p - > next;
}
return head;
}
```

Choose the correct alternative to replace the blank line.

(a) q = NULL; p - > next = head; head = p;
(b) q - > next = NULL; head = p; p - > next = head;
(c) head = p; p -> next = q; q -> next = NULL;
(d) q-> next = NULL; p-> next = head; head = p;

37. What is the value printed by the following C program? **[2010, 2 marks]**

```
#include < stdio.h>
int f (int *a, int n)
{
        if (n < = 0) return 0;
        else if (*a % 2 = = 0) return *a + f(a + 1, n – 1);
        else return *a – f(a + 1, n – 1);
}
int main ()
{
        int a [ ] = {12, 7, 13, 4, 11, 6};
        print f (“%d, f(a, 6));
        return 0,
}
```

(a) – 9 (b) 5
(c) 15 (d) 19

38. Which languages necessarily need heap allocation in the runtime environment? **[2011, 2 marks]**

(a) Those that support recursion
(b) Those that use dynamic scoping
(c) Those that allow dynamic data structures
(d) Those that use global variables

39. What does the following program print? **[2011, 2 marks]**

```
#include < stdio.h >
void f (int *p, int *q)
        p = q;
        *p = 2;
}
int i = 0, j = 1;
int main () {
        f(&i, &j);
        printf(“%d%d/n”, i, j);
}
```

(a) 2 2 (b) 2 1
(c) 0 1 (d) 0 2

40. In a binary tree with n nodes, every node has an odd number of descendant. Every node is considered to be its own descendant. What is the number of nodes in the tree that have exactly one child? **[2010, 1 mark]**

(a) 0 (b) 1
(c) (n – 1)/2 (d) n – 1

Common Data for Questions 41 and 42

Consider the following recursive C function that takes two arguments unsigned into foo (unsigned int, n, unsigned int r) {

```
if n > 0 return n% foo (n/r, r);
else return 0,
}
```

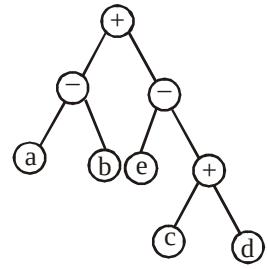
41. What is the return value of the function foo, when it is called as foo (513, 2)? **[2011, 2 marks]**

(a) 9 (b) 8
(c) 5 (d) 2

42. What is the return value of the function foo, when it is called as foo (345, 10)? **[2011, 2 marks]**

(a) 345 (b) 12
(c) 5 (d) 3

43. Consider evaluating the following expression tree on a machine with load store architecture in which memory can be accessed only through load and store instructions. The variables a, b, c, d and e are initially stored in memory. The binary operators used in this expression tree can be evaluated by the machine only when the operands are in registers. The instructions produce result only in a register. If no intermediate results can be stored in memory, what is the minimum number of registers needed to evaluate this expression? **[2011, 2 marks]**



(a) 2 (b) 9
(c) 5 (d) 3

44. We are given a set of n distinct elements and an unlabelled binary tree with n nodes. In how ways can we populate the tree with the given set so that it becomes a binary search tree? **[2011, 2 marks]**

(a) 0 (b) 1
(c) n! (d) $\frac{1}{n+1}\,{}^{2n}C_n$

45. Which of the given options provides the increasing order of asymptotic complexity of functions f_1, f_2, f_3 and f_4?
$f_1(n) = 2^n$, $f_2(n) = n^{3/2}$, $f_3(n) = n\log_2 n$, $f_4(n) = n^{\log 2n}$ **[2011, 2 marks]**

(a) f_3, f_2, f_4, f_1 (b) f_3, f_2, f_1, f_4
(c) f_2, f_3, f_1, f_4 (d) f_2, f_3, f_4, f_1

46. Four matrices M_1, M_2, M_3 and M_4 are of dimensions p.q, q.r, r.s and s.t respectively can be multiplied in several ways with different number of total scalar multiplications. For example, when multiplied as $(M_1 \times M_2) \times (M_3 \times M_4)$ the total number of scalar multiplications, is pqr + rst + prt. When multiplied as $(((M_1 \times M_2) \times M_3) \times M_4)$ the total number of scalar multiplications is pqr + prs + pst. If p = 10, q = 100, r = 20, s = 5 and t = 80, then the minimum number of scalar multiplications needed is **[2011, 2 marks]**

(a) 248000 (b) 44000
(c) 19000 (d) 25000

47. What does the following fragment of C-program print? **[2011, 1 mark]**

```
char c[ ] = “GATE 2011”
char * p = c;
printf (“%s”, p + p[3] – p[1];
```

(a) GATE 2011 (b) E 2011
(c) 2011 (d) 011

48. An algorithm to find the length of the longest monotonically increasing sequence of numbers in an array A[0 – n] is given below.

Let L_i denotes the length of the longest monotonically increasing sequence starting at index i in the array.

Initialize $L_{n-1} = 1$

For all i such that $0 \le i \le n-2$

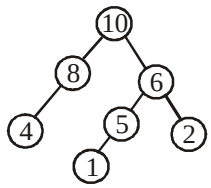
$$L_i = \begin{cases} 1+L_{i+1}, \text{if } A[i] < A[i+1] \\ 1, \quad \text{otherwise} \end{cases}$$

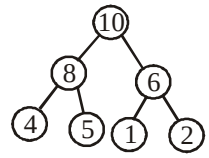
Finally the length of the longest monotonically increasing sequence is Max L, L, ... L.

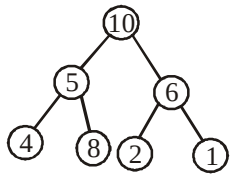
Which of the following statements is true? **[2011, 1 mark]**

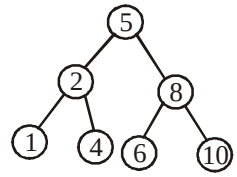
(a) The algorithm uses dynamic programming paradigm
(b) The algorithm has a linear complexity and uses branch and bound paradigm
(c) The algorithm has a non-linear polynomial complexity and uses branch and bound paradigm
(d) The algorithm uses divide and conquer paradigm.

49. A max-heap is a heap where the value of each parent is greater than or equal to the value of its children. Which of the following is a max-heap? **[2011, 1 mark]**

(a)


(b)


(c)


(d)


50. Let G be a weighted graph with edge weights greater than one and G be the graph constructed by squaring the weights of edges in G. Let T and T' be the minimum spanning trees of G and G' respectively, with total weights t and t'. Which of the following statements is true? **[2012, 2 marks]**

(a) T' = T with total weight t' = t^2
(b) T' = T with total weight t' = t^2
(c) T' ¹ T but total weight t' = t^2
(d) None of the above

51. Suppose a circular queue of capacity (n – 1) elements is implemented with an array of n elements. Assume that the insertion and deletion operations are carried out using REAR and FRONT as array index variables, respectively. Initially REAR = FRONT = 0. The conditions to detect queue full and queue empty are **[2012, 2 marks]**

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

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

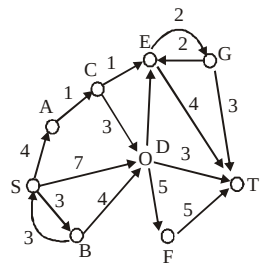
```
Program main;
  Var .....
  Procedure A1;
       Var .....
       Call A2;
  End A1
  Procedure A2;
       Var ...
       Procedure A21;
          Var....
          Call A1;
       End A21
       Call A21;
  End A2;
  Call A1;
End main.
```

Consider the calling chain:

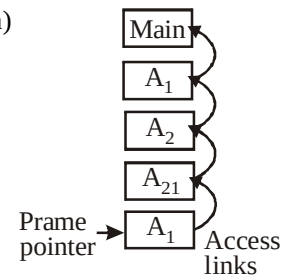
Main → A1 → A2 → A21 → A1

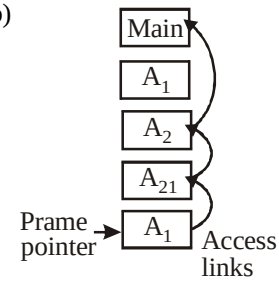
The correct set of activation records along with their access links is given by **[2012, 2 marks]**

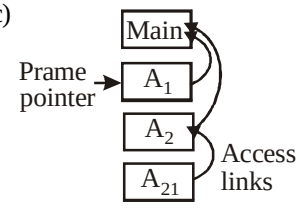
53. Consider the directed graph shown in the figure below. There are multiple shortest paths between vertices S and T. Which one will be reported by Dijkstra's shortest path algorithm'? Assume that in any iteration, the shortest path to a vertex v is updated only when a strictly shorter path to v is discovered. **[2012, 2 marks]**

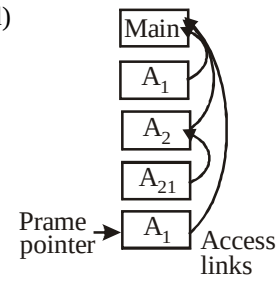


(a) SDT
(b) SBDT
(c) SACDT
(d) SACET

(a)


(b)


(c)


(d)


54. The height of a tree is defined as the number of edges on the longest path in the tree. The function shown in the below in invoked as height (root) to compute the height of a binary tree rooted at the tree pointer root. **[2012, 2 marks]**

```
{   if (n = = NULL) return – 1;
    if (n → left = = NULL)
      if (n → right = = NULL) return 0;
      else return [B1] ;        // Box 1
else {h1 = height (n → left);
    if (n → right = NULL) return (1 + h1);
    else {h2 = height (n → right);
        return [B2] ;        // Box 1
        }
    }
}
```

The appropriate expressions for the two boxes B1 and B2 are

(a) B1 : (1 + height (n → right)
B2 : (1 + max (h1, h2)

(b) B1 : (height (n → right)
B2 : (1 + max (h1, h2)

(c) B1 : (height (n → right)
B2 : max (h1, h2)

(d) B1 : (1 + height (n → right)
B2 : max (h1, h2)

55. The number of elements that can be sorted in Θ(log n) time using heap sort is **[2013, 2 Marks]**

(a) Θ(1) (b) $\Theta\left(\sqrt{\log n}\right)$

(c) $\Theta\left(\frac{\log n}{\log \log n}\right)$ (d) Θ(log n)

56. Consider the following function:

```
int unknown (int n) {
    int i, j, k=0;
    for (i=n/2; i<=n; i++)
    for (j=2; j<=n; j=j*2)
         k = k + n/2;
    return (k);
}
```

The return value of the function is **[2013, 2 Marks]**

(a) $\Theta(n^2)$ (b) $\Theta(n^2 \log n)$

(c) $\Theta(n^3)$ (d) $\Theta(n^3 \log n)$

57. The pre-order traversal sequence of a binary search tree is 30, 20, 10, 15, 25, 23, 39, 35, 42. Which one of the following is the post-order traversal sequence of the same tree? **[2013, 2 Marks]**

(a) 10, 20, 15, 23, 25, 35, 42, 39, 30
(b) 15, 10, 25, 23, 20, 42, 35, 39, 30
(c) 15, 20, 10, 23, 25, 42, 35, 39, 30
(d) 15, 10, 23, 25, 20, 35, 42, 39, 30

58. What is the return value of `f(p,p)`, if the value of `p` is initialized to 5 before the call? Note that the first parameter is passed by reference, whereas the second parameter is passed by value. **[2013, 2 Marks]**

```
int f (int &x, int c) {
    c = c - 1;
    if (c==0) return 1;
    x = x + 1;
    return f(x,c) * x;
}
```

(a) 3024 (b) 6561
(c) 55440 (d) 161051

Common Data for Questions 59 and 60:

The procedure given below is required to find and replace certain characters inside an input character string supplied in array `A`. The characters to be replaced are supplied in array `oldc`, while their respective replacement characters are supplied in array `newc`. Array `A` has a fixed length of five characters, while arrays `oldc` and `newc` contain three characters each. However, the procedure is flawed.

```
void find_and_replace (char *A, char *oldc,
    char *newc)
{
    for (int i=0; i<5; i++)
       for (int j=0; j<3; j++)
                    if (A[i]==oldc[j])
    A[i] = newc[j];
}
```

The procedure is tested with the following four test cases.

1. `oldc = "abc", newc = "dab"`
2. `oldc = "cde", newc = "bcd"`
3. `oldc = "bca", newc = "cda"`
4. `oldc = "abc", newc = "bac"`

59. The tester now tests the program on all input strings of length five consisting of characters `'a', 'b', 'c', 'd'` and `'e'` with duplicates allowed. If the tester carries out this testing with the four test cases given above, how many test cases will be able to capture the flaw? **[2013, 2 Marks]**

(a) Only 1 (b) Only 2
(c) Only 3 (d) All four

60. If array `A` is made to hold the string `"abcde"`, which of the above four test cases will be successful in exposing the flaw in this procedure? **[2013, 2 Marks]**

(a) 2 only (b) 4 only
(c) 3 and 4 only (d) None

61. Consider the following program in C language :

```
#include <stdio.h>
main ()
{
int i;
int *pi = &i;
scanf ("%d", pi);
```

```
printf ("%d\n", i + 5);
}
```

Which one of the following statements is **TRUE** ?

[2014, Set-1, 1 Mark]

(a) Compilation fails
(b) Execution results in a run-time error.
(c) On execution, the value printed is 5 more than the address of variable *i*.
(d) On execution, the value printed is 5 more than the integer value entered.

62. There are 5 bags labeled 1 to 5. All the coins in a given bag have the same weight. Some bags have coins of weight 10 gm, others have coins of weight 11 gm. I pick 1, 2, 4, 8, 16 coins respectively from bags 1 to 5. Their total weight comes out to 323 gm. Then the product of the labels of the bags having 11 gm coins is _____ . **[2014, Set-1, 2 Marks]**

63. Consider the following pseudo code. What is the total number of multiplications to be performed?

[2014, Set-1, 2 Marks]

```
D = 2
for i = 1 to n do
    for j = i to n do
        for k = j + 1 to n do
            D = D*3
```

(a) Half of the product of the 3 consecutive integers.
(b) One-third of the product of the 3 consecutive integers.
(c) One-sixth of the product of the 3 consecutive integers.
(d) None of the above.

64. Consider the following *C* function in which **size** is the number of elements in the array **E**:

```
int MyX(int*E, unsigned int size)
{
    int Y = 0;
    int Z;
    int i, j, k;
    for (i = 0; i < size; i++)
        Y = Y + E[i];
    for (i = 0; i < size; i++)
        for (j = i; j < size; j++)
        {
            Z = 0;
            for (k = i; k <= j; k++)
                Z = Z + E[k];
            if (Z > Y)
                Y = Z;
    }
return Y;
}
```

The value returned by the function **MyX** is the

[2014, Set-1, 2 Marks]

(a) maximum possible sum of elements in any sub-array of array **E**.
(b) maximum element in any sub-array of array **E**.
(c) sum of the maximum elements in all possible sub-arrays of array **E**.
(d) the sum of all the elements in the array **E**.

65. Consider the function func shown below:

```
int func(int num) {
int count = 0;
    while (num) {
        count++;
        num>>= 1;
    }
return (count);
}
```

The value returned by func (435) is _____.

[2014, Set-2, 1 Mark]

66. Suppose *n* and *p* are unsigned int variables in a C program. We wish to set p to ${}^{n}C_{3}$. If *n* is large, which one of the following statements is most likely to set p correctly?

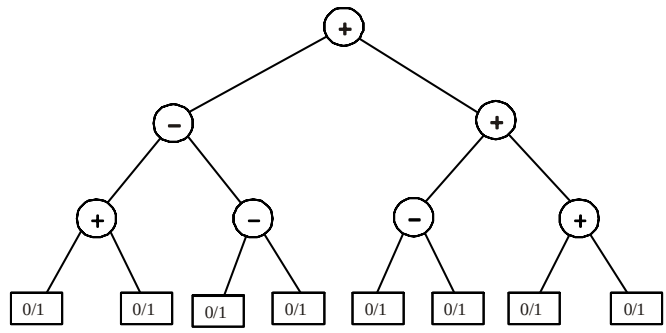
[2014, Set-2, 1 Mark]

(a) $p = n*(n–1)*(n–2) / 6;$
(b) $p = n*(n–1) / 2*(n–2) / 3;$
(c) $p = n*(n–1) / 3*(n–2) / 2;$
(d) $p = n * (n-1) * (n–2) / 6.0;$

67. A priority queue is implemented as a Max-Heap. Initially, it has 5 elements. The level-order traversal of the heap is: 10, 8, 5, 3, 2. Two new elements 1 and 7 are inserted into the heap in that order. The level-order traversal of the heap after the insertion of the elements is: **[2014, Set-2, 1 Mark]**

(a) 10, 8, 7, 3, 2, 1, 5 (b) 10, 8, 7, 2, 3, 1, 5
(c) 10, 8, 7, 1, 2, 3, 5 (d) 10, 8, 7, 5, 3, 2, 1

68. Consider the expression tree shown. Each leaf represents a numerical value, which can either be 0 or 1. Over all possible choices of the values at the leaves, the maximum possible value of the expression represented by the tree is ___.



[2014, Set-2, 2 Marks]

69. Consider the following function

```
double f(double x){
    if( abs(x*x – 3) < 0.01) return x;
    else return f(x/2 + 1.5/x);
}
```

Give a value q (to 2 decimals) such that f(q) will return q:_____. **[2014, Set-2, 2 Marks]**

70. Suppose a stack implementation supports an instruction REVERSE, which reverses the order of elements on the stack, in addition to the PUSH and POP instructions. Which one of the following statements is TRUE with respect to this modified stack? **[2014, Set-2, 2 Marks]**

(a) A queue cannot be implemented using this stack.
(b) A queue can be implemented where ENQUEUE takes a single instruction and DEQUEUE takes a sequence of two instructions.
(c) A queue can be implemented where ENQUEUE takes a sequence of three instructions and DEQUEUE takes a single instruction.
(d) A queue can be implemented where both ENQUEUE and DEQUEUE take a single instruction each.

71. Consider the C function given below.

```
int f(int j)
{
    static int i = 50;
    int k;
    if (i == j)
        {
            printf("something");
            k = f(i);
            return 0;
        }
    else return 0;
}
```

Which one of the following is **TRUE**?

[2014, Set-2, 2 Marks]

(a) The function returns 0 for all values of j.
(b) The function prints the string something for all values of j.
(c) The function returns 0 when j = 50.
(d) The function will exhaust the runtime stack or run into an infinite loop when j = 50.

72. Let A be a square matrix of size n × n. Consider the following pseudocode. What is the expected output ?

[2014, Set-3, 1 Mark]

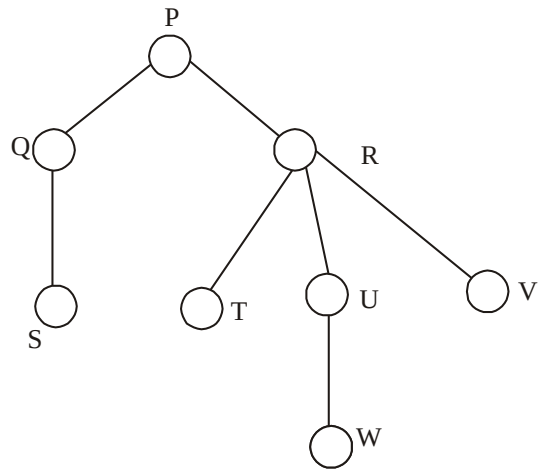
```
c = 100;
for i = 1 to n do
  for j = 1 to n do
{
 Temp = A [i] [j] + C;
 A [i] [j] = A [j] [i];
 A [j] [i] = Temp – C;
}
for i = 1 to n do
    for j = 1 to n do
      output (A [i] [j]);
```

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

73. The minimum number of arithmetic operations required to evaluate the polynomial $P(X) = X^5 + 4X^3 + 6X + 5$ for a given value of X, using only one temporary variable is _____.

[2014, Set-3, 1 Mark]

74. Consider the following rooted tree with the vertex labeled P as the root :



The order in which the nodes are visited during an in-order traversal of the tree is **[2014, Set-3, 1 Mark]**

(a) SQPTRWUV
(b) SQPTUWRV
(c) SQPTWUVR
(d) SQPTRUWV

Practice Exercise

1. Which of the following comments are not true?
1. C provides no input-output features
2. C provides no file access features
3. C borrowed most of its ideas from BCPL
4. C provides no features to manipulate composite objects

(a) 1 only (b) 1, 2 and 3
(c) 1, 2, 3 and 4 (d) None of these

2. Any C program
(a) must contain at least one function
(b) need not contain any function
(c) needs input data
(d) None of the above

3. Preprocessing is typically done
(a) either before or at the beginning of the compilation process
(b) after compilation but before execution
(c) after loading
(d) None of these

4. The purpose of the following program fragment.

```
b = s + b;
s = b – s;
b = b – s;
```

where s, be are two integers is to
(a) transfer the contents of s to b
(b) transfer the contents of b to s
(c) exchange (swap) the contents of s and b
(d) negate the contents of s and b

5. Which of the following about conditional compilation is not true?
1. It is taken care of by the compiler
2. It is setting the compiler option conditionally.
3. It is compiling a program based on a condition.
4. It is taken care of by the pre-processor.

(a) 2 only (b) 3 and 4 only
(c) 1 and 2 (d) All of the above

6. Lengths of the string "correct" and "correct string" are
(a) 7, 14
(b) 8, 14
(c) 6, 13
(d) implementation dependant

7. For C preprocessor, which of the following is/are true?
1. Takes care of conditional compilation
2. Takes care of macros
3. Takes care of includes files
4. Acts before compilation

(a) 1 and 2 (b) 3 only
(c) 1, 2 and 3 (d) All of the above

8. int i = 5; is a statement in a C program. Which of the following are true?
(a) During execution, value of i may change but not its address
(b) During execution both the address and value may change
(c) Repeated execution may result in different addresses for i
(d) i may not have an associated address

9. The declaration

```
enum cities {bethlethem, jericho, nazareth = 1, jerusalem}
```

assign the value 1 to
(a) bethlehem (b) nazareth
(c) bethlehem and nazareth (d) jericho and nazareth

10. The value of an automatic variable that is declared but not initialized will be
(a) 0 (b) – 1
(c) unpredictable (d) None of these

11. The following program fragment

```
int a = 4, b = 6;
printf ("%d", a = = b);
```

(a) outputs an error message (b) prints 0
(c) prints 1 (d) none of the above

12. x – = y + 1; means
(a) x = x – y + 1 (b) x = – x – y – 1
(c) x = – x + y + 1 (d) x = x – y – 1

13. The following program fragment

```
int x[ 5 ] [ 5 ], i, j;
for (i = 0; i < 5; ++i)
for (j = 0; j < 5, j++)
x[i][j] = x[j][i];
```

(a) transposes the given matrix x
(b) makes the given matrix x, symmetric
(c) doesn't alter the matrix x
(d) None of these

14. Consider the statement

```
int val [2][4] = {1, 2, 3, 4, 5, 6, 7, 8};
```

4 will be the value of when array declaration in the row-major order.
(a) val [1] [4] (b) val [0] [4]
(c) val [1] [1] (d) None of the above

15. The maximum number of dimension an array can have in C is
(a) 3 (b) 4
(c) 5 (d) None of the above

16. Consider the array definition
int num [10] = {3, 3, 3};
Pick the correct answers.
(a) num [9] is the last element of the array num
(b) the value of num [8] is 3
(c) the value of num [3] is 3
(d) None of the above

17. Consider the following type definition.
typedef char x [10];
x myArray [5];
What will size of (myArray) be? (Assume one character occupies 1 byte)
(a) 15 bytes (b) 10 bytes
(c) 50 bytes (d) 30 bytes

18. If n has the value 3, then the statement a[++n] = n++;
(a) assigns 3 to a [5]
(b) assigns 4 to a [5]
(c) assigns 4 to a [4]
(d) what is assigned is compiler-dependent

19. If a global variable is of storage class static, then
(a) the static declaration is unnecessary if the entire source code is in a single file
(b) the variable is recognized only in the file in which it defined
(c) it results in a syntax error
(d) both (a) and (b)

20. The default parameter passing mechanism is
(a) call by value (b) call by reference
(c) call by value result (d) none of these

21. The storage class static can be used to
(a) restrict the scope of an external identifier
(b) preserve the exit value of variables
(c) provide privacy to a set of functions
(d) All of the above

22. The following program
```
main ()
    {printf ("tim");
    main ();}
```
(a) is illegal (b) keeps on printing tim
(c) prints tim once (d) None of the above

23. Consider the following program
```
main ()
    {putchar ('M');
     first ();
     putchar ('m');}
first ()
{ _____ }
second ()
{putchar ('d');}
```
If Madam is the required output, then the body of first () must be
(a) empty
(b) second (); putchar ('a');
(c) putchar ('a'); second (); printf("%C", 'a');
(d) none of the above

24. Max is a function that returns the larger of the two integers, given as arguments. Which of the following statements does not finds the largest of three given numbers?
(a) max (max (a, b), max (a, c))
(b) max (a, max (a, c))
(c) max (max (a, b), max (b, c))
(d) max (b, max (a, c))

25. void can be used
(a) as data-type of a function that returns nothing to its calling environment
(b) inside the brackets of a function that does not need any argument
(c) in an expression and in a printf statement
(d) both (a) and (b)

26. The following program,
```
main ()
{
    int i = 2;
    { int i = 4, j = 5;
     printf ("%d%d", i, j);
    }
    printf ("%d%d", i, j);
}
```
(a) will not compile successfully
(b) prints 4525
(c) prints 25 25
(d) None of the above

27. The following program,
```
main ()
{
    inc (); inc (); inc ();
    }
    inc()
    {
    static int x;
    printf ("%d", ++x);
}
```
(a) prints 012
(b) prints 123
(c) prints 3 consecutive, but unpredictable numbers
(d) prints 111

28.
```
main ()
{int a = 5, b = 2;
 print f ("%d", a +++b);
}
```
(a) results in syntax error
(b) prints 7
(c) prints 8
(d) None of these

29. The program fragment
```
    int a = 5, b = 2;
    printf ("%d", a +++++b);
```
(a) prints 7 (b) prints 8
(c) prints 9 (d) none of these

30. If arr is a two dimensional array of 10 rows and 12 columns, then arr [5] logically points to the
(a) sixth row (b) fifth row
(c) fifth column (d) sixth column

31. The statements
int**a;
(a) is illegal
(b) is legal but meaningless
(c) is syntactically and semantically correct
(d) none of these

32. Consider the following declaration,
int a, * b = &a, **c = & b;
the following program fragment
a = 4;
**c = 5 ;
(a) does not change the value of a
(b) assigns address of c to a
(c) assigns the value of b to a
(d) assigns 5 to a

33. Let x be an array. Which of the following operations are illegal?
1. ++x 2. x+1
3. x++ 4. x*2
(a) 1 only (b) 3 and 4
(c) 2, 3 and 4 (d) 1, 3 and 4

34. Pick the correct answers.
If x is an one dimensional array, then
(a) &x[i] is same as x + i – 1
(b) * (x + i) is same as * (&x [i])
(c) *(x + i) is same as x[i]
(d) both (b) and (c)

35. Choose the correct statements
(a) An entire array can be passed as argument to a function.
(b) A part of an array can be passed as argument to a function.
(c) Any change done to an array that is passed as an argument to a function will be local to the function.
(d) Both (a) and (b)

36. A pointer variable can not be
(a) passed to a function as argument
(b) changed within a function
(c) returned by a function
(d) assigned an integer value

37. If func is a function needing three arguments a1, a2, a3, then func can be invoked by
(a) func (a1, a2, a3); (b) (*func) (a1, a2, a3);
(c) *func (a1, a2, a3); (d) Both (a) and (b)

38. The following program
```
main ()
{
    float a = .5, b = .7;
    if (b < = 7)
     if (a < .5)
       printf ("IES");
    else
      printf ("PSU");
    else
      printf ("GATE");
}
```
output is
(a) PSU (b) IES
(c) GATE (d) None of these

39. a → b is syntactically correct if
(a) a and b are structures
(b) a is a structure and b is a pointer to a structure
(c) a is a pointer to a structure and b is a structure
(d) a is a pointer to a structure is which b is a field

40. If a file is opened in r+ mode then
1. reading is possible
2. writing is possible
3. it will be created if it does not exist
(a) 1 is true (b) 1 and 3 are true
(c) 1 and 2 are true (d) 1, 2 and 3 are true

41. The contents of a file will be lost if it is opened in
(a) a mode (b) w mode and w + mode
(c) r + mode (d) a + mode

42. Bit field
(a) is a field having many sub-fields
(b) is a structure declaring the sizes of the members in terms of bits
(c) is a member of a structure whose size is specified in terms of bits
(d) none of these

43. The statement printf ("%d", size of (“”)); prints
(a) an error message (b) 0
(c) garbage (d) 1

44. The statement printf ("%d", (a++)); prints
(a) the current value of a (b) the value of a + 1
(c) an error message (d) garbage

45. Consider the function
find (int x, int y)
{return ((x < y) ? 0 : (x – y)); }
Let a, b be two non-negative integers. The call find (a, find (a, b)) can be used to find the
(a) maximum of a, b
(b) positive difference of a, b
(c) sum of a, b
(d) minimum of a, b

46. If abc is the input, then the following program fragment
char x, y, z;
printf ("%d", scanf ("%c%c%c, &x, &y, &z)); results in
(a) a syntax error (b) a fatal error
(c) segmentation violation (d) printing of 3

47. Consider the statements
putchar (getchar ());
putchar (getchar ());
If
a
b
is the input, the output will be
(a) an error message (b) a/nb
(c) ab (d) a b

48. If y is of integer type then the expressions
3 * (y – 8)/9 and (y – 8) / 9*3
yield the same value if
(a) y must yield the same value
(b) y must yield different values
(c) y may or may not yield the same value
(d) None of these

49. The program fragment
```
int i = 263;
putchar (i);
```
(a) prints 263
(b) prints the ASCII equivalent of 263
(c) rings the bell
(d) prints garbage

50. The following statement
printf ("%f", 9/5);
prints
(a) 1.8 (b) 1.0
(c) 2.0 (d) none of these

51. The following loop
for (putchar ('c'); putchar ('a'); putchar ('r')) putchar ('t');
outputs
(a) a syntax error (b) cartrt
(c) catrat (d) catratratratrat...

52. The following program
```
main ()
{
  int i = 5;
 if (i = = 5) return;
 else printf "i is not five");
 printf ("over");
}
```
results in
(a) a syntax error
(b) an execution error
(c) printing of over
(d) execution termination, without printing anything

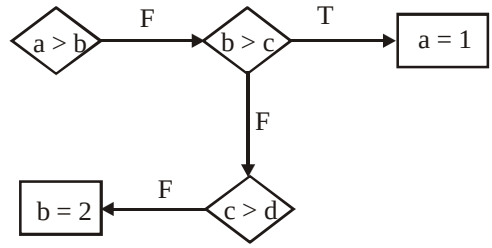
53. The following program fragment
int i = 5;
do {putchar (i + 100); printf ("%d", i – –):}
while (i);
results in the printing of
(a) i5h4g3f 2e1 (b) i4h3g2 fle0
(c) an error message (d) none of these

54. The following statements
for (i = 3; i < 15; i + = 3)
{ printf ("%d", i);
++i;
}
will result in the printing of
(a) 3 6 9 12 (b) 3 6 9 12 15
(c) 3 7 11 (d) 3 7 11 15

55. If a = 9, b = 5 and c = 3, then the expression (a – a/b * b % c) > a % b % c evaluates to
(a) true (b) false
(c) invalid (d) 0

56. Consider the following program fragment
```
if (a > b)
 printf ("a > b")
else
 printf ("else part");
 printf ("a < = b");
```
then a < = b
will be printed if
(a) a > b (b) a < b
(c) a = = b (d) All of the above

57. Consider the following flow chart.



Which of the following does not correctly implements the above flow chart?

(a)
```
if (a > b)
if (b > c)
a = 1;
else if (c > d)
b = 2;
```
(b)
```
if (a < = b)
if (b > c)
a = 1;
else if (c < = d)
b = 2;
```
(c)
```
if (a > b)
;
else if (b > c)
a = 1;
else if (c < = d)
b = 2;
```
(d)
```
if (a < = b)
;
else if (b > c)
a = 1;
else if (c > d)
;
else b = 2;
```

58. The following statement
```
if (a > b)
if (c > b)
print f ("one");
else
if (c = = a) printf ("two");
else print f ("three");
else print f ("four");
```
(a) results in a syntax error
(b) prints four if c < = b, can here print four
(c) prints two if c < = b, can never print two
(d) prints four if a < = b, can never print four

59. The following program fragment
```
int x = 4, y = x, i;
for (i = 1; i < 4; ++ i)
x + = x;
```
outputs an integer that is same as
(a) 8 * y (b) y * (1 + 2 + 3 + 4)
(c) y * 4 (d) y * y

60. Consider the declaration
static char hello { } = "hello" ;
The output of print f ("% s \ n", hello);
will not be the same as that of
(a) puts ("hello");
(b) puts (hello);
(c) printf ("%s \ n", "hello");
(d) puts ("hello \ n");

61. The following program
```
main ()
{
static int a[ ] = {7, 8, 9};
print f ("%d", 2 [a] + a[2]);
}
```
(a) results in bus error
(b) results in segmentation violation error
(c) will not compile successfully
(d) None of these

62. The following program
```
main ()
{
    static char a[3] [4] = {"abcd", "mnop", "fghi"},
    putchar (**a);
}
```
(a) will not compile successfully
(b) results in run-time error
(c) prints garbage
(d) None of these

63. The output of the following program is
```
main ()
{   static int x[ ] = {1, 2, 3, 4, 5, 6, 7, 8},
    int i;
    for (i = 2; i < 6; ++i)
     x [x[i]] = x[i];
    for (i = 0; i < 8; ++i)
    print f ("%d", x[i]);
}
```
(a) 1 2 3 3 5 5 7 8 (b) 1 2 3 4 5 6 7 8
(c) 8 7 6 5 4 3 2 1 (d) 1 2 3 5 4 6 7 8

64. Consider the following program
```
main ()
{   int x = 2, y = 2;
    if (x < y) return (x = x + y);
    else printf ("z1");
    printf ("z2");
}
```
Choose the correct statements
(a) The output is z2
(b) The output is z1z2
(c) This will result in compilation error
(d) None of the above

65. A possible output of the following program fragment
static char wer [ ] [5] = {"harmot", "merli", "axari");
printf ({"%d %d", wer, wer [0], &wer [0] [0]); is
(a) 262164 262164 262164 (b) 262164 262165 262166
(c) 262164 262165 262165 (d) 262164 262164 262165

66. The following program
```
main ()
{   int abc ();
    abc ();
    (*abc) ();
    }
    int abc ()
    {printf ("come");}
```
(a) results in a compilation error
(b) prints come come
(c) results in a run time error
(d) prints come

67. The possible output of printf ("%d %d, wer [1], wer [1] + 1); is
(a) 162 163 (b) 162 166
(c) 162 166 (b) 162 165

68. The possible output of printf ("%d %d, wer, wer + 1); is
(a) 262 262 (b) 262 266
(c) 262 263 (b) 262 265

69. While sorting a set of names, representing the names as an array of pointers is preferable to representing the name as a two dimensional array of characters, because
(a) storage needed will be proportional to the size of the data
(b) execution will be faster
(c) swapping process becomes easier and faster
(d) All of the above

70. Consider the two declarations
```
void * voidPtr;
char *charPtr;
```
Which of the following assignments are syntactically correct?
(a) voidPtr = charPtr (b) charPtr = voidPtr
(c) * voidPtr = * charPtr (d) * charPtr = voidPtr

71. Consider the declaration
```
char x[ ] = "SUCCESS";
char *y = "SUCCESS";
```
Pick the correct answers.
(a) The output of puts (x) and puts (y) will be the same.
(b) The output of puts (x) and puts (y) will be different.
(c) The output of puts (y) is implementation dependent.
(d) None of the above comments are true.

72. Consider three pegs A, B, C and four disks of different sizes. Initially, the four disks are stacked on peg A, in orderof decreasing size. The task is to move all the disks from peg A to peg C with the help of peg B. themoves are to be made under the following constraints:
[i] In each step, exactly one disk is moved from one peg to another.
[ii] A disk cannot be placed on another disk of smaller size.
If we denote the movement of a disk from onepeg to another by $y \rightarrow y$, where y, y are A, B or C, then represent the sequence of the minimum numberof moves to accomplish this as a binary tree with node labels of the form $(y \rightarrow y)$ such that the in-ordertraversal of the tree gives the correct sequence of the moves.If there are n disks, what is the total number of moves required in terms of n. Answer can be an expression in terms of n.
(a) $2^n - 1$ (b) 2^{n+1}
(c) 2^{n-1} (d) $2^n - 3$

73. The following sequence of operations is performed on a stack :
PUSH (10), PUSH (20), POP, PUSH (10), PUSH (20), POP, POP, POP, PUSH (20), POP.
(a) 10 20 10 20 20 (b) 20 10 10 20 20
(c) 20 20 10 10 20 (d) None of these

74. Choose the false statements.
(a) The scope of a macro definition need not be the entire program.
(b) The scope of a macro definition extends from the point of definition to the end of the file.
(c) A macro definition may go beyond a line
(d) None of the above

75. Calloc (m, n); is equivalent to
(a) malloc (m*n, 0);
(b) memset (0, m*n);
(c) ptr = malloc (m*n); memset (p, 0, m*n)
(d) ptr = malloc (m*n); strcpy (p, 0)

76. The for loop

```
for (i = 0, i < 10; ++i)
printf ("%d", i &1);
```

prints

(a) 0101010101 (b) 0111111111
(c) 0000000000 (d) 1111111111

77. The output of the following program

```
main ()
{   int a = 1, b = 2, c = 3;
    printf ("%d", a + = (a + = 3, 5, a));
}
```

will be

(a) 8 (b) 12
(c) 9 (d) 6

78. Consider the following program segment.

```
char *a, *b, c [10], d[10];
a = b;
b = c;
c = d;
d = a;
```

Choose the statements having errors.

(a) No error (b) a = b; and b = c;
(c) c = d; and d = a; (d) a = b; and d = a;

79. Consider the following statements

```
# define hypotenuse (a, b) sqrt (a * a + b * b);
The macro-call hypotenuse (a + 2, b + 3);
```

(a) Finds the hypotenuse of a triangle with sides a + 2 and b + 3
(b) Finds the square root of $(a + 2)^2 + (b + 3)^2$
(c) is invalid
(d) Finds the square root of 3*a + 4*b + 5

80. Which of the following comments about arrays and pointers is/are not true?

1. Both are exactly same
2. Array is a constant pointer
3. Pointer is an one-dimensional array
4. Pointer is a dynamic array

(a) 1, 3 and 4 (b) 1, 2, and 3
(c) 2, 3 and 4 (d) 1, 2, 3 and 4

81. Use of macro instead of function is recommended

(a) when one wants to reduce the execution time
(b) when there is a loop with a function call inside
(c) when a function is called in many places in a program
(d) In (a) and (b) above

82. Which of the following statements is (are) correct?

(a) Enum variables can be assigned new values
(b) Enum variables can be compared
(c) Enumeration feature does not increase the power of C
(d) All of these

83. For 'C' programming language,

(a) constant expressions are evaluated at compile time
(b) string constants can be concatenated at compile time
(c) size of array should be known at compile time
(d) All of these

84. For loop in a C program, if the condition is missing

(a) it is assumed to be present and taken to be false
(b) it is assumed to be present and taken to the true
(c) it result in a syntax error
(d) execution will be terminated abruptly

85. Which of the following statements about for loop is/are correct?

(a) Index value is retained outside the loop
(b) Index value can be changed from within the loop
(c) Goto can be used to jump, out of the loop
(d) All of these

86. If c is a variable initialised to 1, how many times will the following loop be executed?

```
while ((c > 0). && (c < 60)) {
  loop body
  c + + ;}
```

(a) 60 (b) 59
(c) 61 (d) 1

87. The program fragment

```
int i = 263;
putchar (i);
```

prints

(a) 263 (b) ASCII equivalent of 263
(c) rings the bell (d) garbage

88. If statement

```
b = (int *) **c;
```

is appended to the above program fragment, then

(a) value of b is unaffected
(b) value of b will be the address of c
(c) value of b becomes 5
(d) None of these

89. Consider the declarations:

```
char first (int (*) (char, float));
int second (char, float);
```

Which of the following function invocation is valid?

(a) first (*second) (b) first (& second);
(c) first (second); (d) None of these

90. Consider the declaration:

```
static struct {unsigned a : 5;
  unsigned b : 5;
  unsigned c : 5;
  unsigned d : }v = (1, 2, 3, 4);
```

v occupies

(a) 4 words (b) 2 words
(c) 1 word (d) None of these

91. The value of ab if ab & 0 × 3f equals 0 × 27 is

(a) 047 (b) 0 × 0f
(c) 0 × f 3 (d) 0 × 27

92. A function can make

(a) one throw
(b) one throw of each scale type
(c) one throw of each programmer defined type
(d) as many throws of as many types as necessary.

93. In the statement template << class T>>,

(a) T is a class (b) T is a scalar variable
(c) either (a) or (b) (d) None of these

94. In C programming language, which of the following type of operators have the highest precedence
(a) relational operators
(b) equality operators
(c) logical operators
(d) arithmetic operators

95. In C programming language, which of the following operators has the highest precedence?
(a) unary + (b) *
(c) ≥ (d) = =

96. What will be the value of x and y after execution of the following statement (C language) n = = 5; x = n++; y = – x?
(a) 5, 4 (b) 6, 5
(c) 6, 6 (d) 5, 5

97. C programming language provides operations which deal directly with objects such as
(a) strings and sets
(b) lists and arrays
(c) characters, integers, and floating point numbers
(d) All of these

98. C programming language by itself provides
(a) input facility
(b) output facility
(c) both input and output facilities
(d) no input and output facilities

99. In what kind of storage structure for strings, one can easily insert, delete, concatenate and rearrange substrings?
(a) Fixed length storage structure
(b) Variable length storage with fixed maximum
(c) Linked list storage
(d) Array type storage

100. The time required to search an element in a linked list of length n is
(a) $O(\log_2 n)$ (b) O(n)
(c) O(1) (d) $O(n^2)$

101. Consider a linked list of n element which is pointed by an external pointer. What is the time taken to delete the element which is successor of the element pointed to by a given pointer?
(a) O(1) (b) $O(\log_2 n)$
(c) O(n) (d) $O(n \log_2 n)$

102. Which of the following is a tabular listing of contents of certain registers and memory locations at different times during the execution of a program?
(a) Loop program (b) Program trace
(c) Subroutine program (d) Byte sorting program

103. For a linear search in an array of n elements the time complexity for best, worst and average case are....., ... and .... respectively
(a) O(n), O(1) and O(n/2) (b) O(1), O(n) and O(n/2)
(c) O/1, O(n) and O(n) (d) O(1), O(n) and $\left(\frac{n-1}{2}\right)$

104. Using the standard algorithm, what is the time required to determine that a number n is prime?
(a) Linear time (b) Logarithmic time
(c) Constant time (d) Quadratic time

105. Following is a recursive function for computing the sum of integers from 0 to N.
function sum (N : integer): integer
begin
if N = 0 then Sum = 0
else
end;
The missing line in the else part is
(a) Sum : = N + Sum (N)
(b) Sum : = N + Sum (N – 1)
(c) Sum : = (N – 1) + Sum (N)
(d) Sum : = (N – 1) + Sum (N – 1)

106. What is the value of F(4) using the following procedure?
function F(k : integer) : integer;
begin
if (k < 3)
then F: = k
else F : = F(k – 1) * F(k – 2) + F(k – 3)
end;
(a) 5 (b) 6
(c) 7 (d) 8

107. Which of the following types of expressions does not require precedence rule for evaluated?
(a) Full parenthesized infix expression
(b) Prefix expression
(c) Partially parenthesized infix expression
(d) More than one of these

108. If space occupied by null terminated string "S_1" and "S_2" in "c" are respectively "m" and "n", the space occupied by the string obtained by concatenating "S_1" and "S_2" is always
(a) less than m + n (b) equal to m + n
(c) greater than m + n (d) None of these

109. Output of the following 'C' program is

```
main ()
{
  printf ("\n%x", – 1 >>4);
}
```

(a) ffff (b) 0fff
(c) 0000 (d) fff0

110. If abc is the input, then following program fragment

```
char x, y, z;
printf ("%d", scanf ("%c%c%c", &x, &y, &z));
```

results in
(a) a syntax error (b) a fatal error
(c) segmentation violation (d) printing of 3

111. The rule for implicit type conversion in 'C' is
(a) int < unsigned < float < double
(b) unsigned < int < float < double
(c) int < unsigned < double < float
(d) unsigned < int < double < float

112. Result of the execution of the following 'C' statements is

```
int i = 5;
do {putchar (i + 100); printf ("%d", i– –);}
while (i);
```

(a) i5hug3f 2e1 (b) 14h3g2f1e0
(c) an error message (d) None of these

113. A "switch" statement is used to
(a) switch between functions in a program
(b) switch from one variable to another variable
(c) to choose from multiple possibilities which may arise due to different values of a single variable
(d) to use switching variable

114. When a function is recursively called, all automatic variables
(a) are initialized during each execution of the function
(b) are retained from the last execution
(c) are maintained in a stack
(d) None of these

115. For x and y are variables as declared below
double x = 0.005, y = – 0.01;
what is the value of ceil (x + y), where ceil is a function to compute ceiling of a number?
(a) 1 (b) 0
(c) 0.005 (d) 0.5

116. The declarations
typedef float height [100];
height men, women;
(a) define men and women as 100 element floating point arrays
(b) define men and women as floating point variables
(c) define height, men and women as floating point variables
(d) are illegal

117. A short integer occupies 2 bytes, an ordinary integer 4 bytes and a long integer occupies 8 bytes of memory.
If a structure is defined as
struct TAB {
short a;
int b;
long c;
}TABLE [10];
then total memory requirement for TABLE is
(a) 14 (b) 140
(c) 40 (d) 24

118. If i, j, k are integer variable with values 1, 2, 3 respectively, then what is the value of the expression
((j + k) > (i + 5))
(a) 6 (b) 5
(c) 1 (d) 0

119. The expression a << 6 shifts all bits of a six places to the left. If a 0x6db7, then what is the value of a << 6?
(a) 0xa72b (b) 0xa2b
(c) 0x6dc0 (d) 0x1111

120. The declaration
union id {
char colour [12];
int size; } shirt, pant;
denotes shirt and pant are variable of type id and
(a) each can have a value of colour and size
(b) each can represent either a 12-character colour or a integer size at a time
(c) shirt and pant are same as struct variables
(d) variable shirt and pant cannot be used simultaneously in a statement.

121. The expression 5 – 2 – 3 * 5 – 2 will evaluate to 18, if – is left associative and
(a) * has precedence over *
(b) * has precedence over –
(c) – has precedence over *
(d) – has precedence over –

122. What is the output of this program?

```
1.  #include <iostream>
2.  #include <functional>
3.  #include <numeric>
4.  using namespace std;
5.  int myop (int x, int y)
6.  {
7.      return x + y;
8.  }
9.  int main ()
10.  {
11.     int val[] = {1, 2, 3, 5};
12.     int result[7];
13.     adjacent_difference (val, val + 7, result);
14.     for (int i = 0; i < 4; i++)
15.         cout << result[i] <<' ';
16.     return 0;
17.  }
```

(a) 1112 (b) 2111
(c) 1212 (d) None of these

123. The C declaration
int b [100];
reserves _____ successive memory locations, each large enough to contain single integer.
(a) 200 (b) 10,000
(c) 100 (d) 10

124. Match List I with List II and select the correct answer from the codes given below the lists:

	List I		**List I**
A.	m = malloc (5); m = NULL	1.	Using doing long pointers
B.	Free (n); n → value = 5;	2.	Using uninitialized pointers
C.	char* P, *P = 'a';	3.	Lost memory

Codes:

	A	**B**	**C**
(a)	1	2	3
(b)	3	1	2
(c)	3	2	1
(d)	2	3	1

125. The average search time of hashing, with linear probing will be less if the load factor
(a) is for less than one (b) equals one
(c) is for greater than one (d) None of these

126. argv is a/an
(a) array of character pointers
(b) pointer to an array of character pointers
(c) array of strings
(d) None of these

127. In the following declarations:

```
typedef struct {
char name [20];
char middlename [5];
char surname [20];
    } NAME
NAME class [20];
```

class is

(a) an array of 20 characters only
(b) an array of 20 names where each name consists of a name, middlename and surname
(c) a new type
(d) none of these

128. What would be the values of i, x and c if

```
scanf ("%3d, %5f, %c", &i, &x, &c)
```

is executed with input data 10b 256, 875bT?

(a) i = 10, b = 56.875, C = T
(b) i = 100, b = 256.87, C = T
(c) i = 010, b = 256.87, C = '5'
(d) i = 10, b = 256.8, C = '7'

129. The five items : A, B, C, D and E are pushed in a stack, one after the other starting from A. The stack is popped four times and each element is inserted in a queue. Then two elements are deleted from the queue and pushed back on the stack. Now one item is popped from the stack. The popped item is

(a) A (b) B
(c) C (d) D

130. Which of the following is an illegal array definition?

(a) type COLOGNE: (LIME, PINE, MUSK, MENTHOL);
var a : array [COLOGNE] of REAL;
(b) var a : array [REAL] of REAL;
(c) var a : array ['A'.. 'Z'] of REAL;
(d) var a : array [BOOLEAN] of REAL;

131. Which of the following assertions is most strongly satisfied at the point marked [1]?

(a) list [j] < list [j + 1] for all j such that item ≤ j < n
(b) list [j] < list [j + 1] for all j such that item < j ≤ n
(c) list [j] ≤ list [j + 1] for all j such that item ≤ j < n
(d) list [j] < list [j – 1] for all j such that 1 < j ≤ item

132. Using Pop (SI, Item), Push (SI, Item), Read (Item), Print (Item), the variables SI (stack) and Item, and given the input file: A, B, C, D, E, F, < EOF >
Which stacks are possible?

(a)
5 A
4 B
3 C
2 D
1 E

(b)
5
4
3 D
2 A
1 F

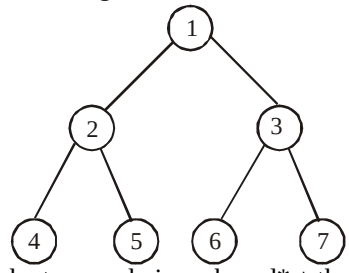
(c)
5
4
3 F
2 D
1 B

(d)
5
4
3 C
2 E
1 B

133. Using Pop (SI, Item), Push (SI, Item), Getlist (Item), P_{op} (S2, Item), Push (S2, Item), and the variables S1, S2 (stacks with Top 1 and Top 2) and Item and given the input file:
A, B, C, D, E, F, < EOF >
Which stacks are possible:

(a) All possible stacks with A, B, C, D, E and F
(b) No possible stacks with A, B, C, D, E and F
(c) Exactly and only those stacks which can be produced with SI alone
(d) Twice as many stacks can be produced with SI alone

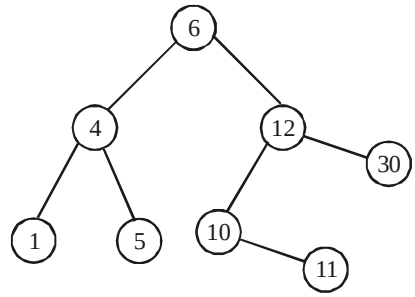
134. Consider the following tree



If the post order traversal gives ab – cd* + then the label of the nodes 1, 2, 3, ... will be

(a) +, –, *, a, b, c, d (b) a, –, b, +, *, d
(c) a, b, c, d, d, –, *, + (d) –, a, b, +, *, c, d

135. Consider the following tree



If this tree is used for sorting, then a new number 8 should be places as the

(a) left child of the node labeled 30
(b) right child of the node labeled 5
(c) right child of the node labeled 30
(d) left child of the node labeled 10

136. The number of possible binary search trees with 3 nodes is

(a) 12 (b) 13
(c) 5 (d) 15

137. You want to check whether a given set of items is sorted. Which of the following sorting methods will be the most efficient if it is already is sorted order?

(a) Bubble sort (b) Selection sort
(c) Insertion sort (d) Merge sort

138. As part of the maintenance work, you are entrusted with the work of rearranging the library books in a shelf in proper order, at the end of each day. The ideal choice will be

(a) bubble sort (b) insertion sort
(c) selection sort (d) heap sort

139. Which of the following algorithms solves the all-pair shortest path problem?

(a) Dijkstra's algorithm (b) Floyd's algorithm
(c) Prim's algorithm (d) Warshall's algorithm

140. Which of the following expressions accesses the $(i, j)^{th}$ entry of a (m × n) matrix stored in column major form?

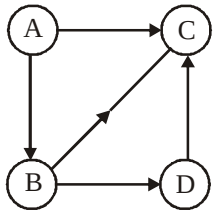
(a) n × (i – 1) + j (b) m × (j – 1) + i
(c) m × (n – j) + j (d) n × (m – i) + j

141. Sparse matrices have

(a) many zero entries (b) many non-zero entries

(c) higher dimension (d) none of the above

142. Consider the graph in figure



Which of the following is a valid topological sorting?

(a) A B C D (b) B A C D

(c) B A D C (d) A B D C

143. For merging two sorted lists of sizes m and n into a sorted list of size m + n. Find out the time complexity of this merging process.

(a) O(m) (b) O(n)

(c) O(m + n) (d) O(log(m) + log (n))

144. A binary tree has n leaf nodes. The maximum numbers of nodes of degree 2 in this tree is

(a) $\log_2 n$ (b) n – 1

(c) n (d) 2^n

145. The postfix expression for the infix expression

A + B* (C + D)/F + D*E is:

(a) AB + CD + *F/D + E* (b) ABCD + *F / + DE* +

(c) A*B + CD? F*DE ++ (d) F*DE++

146. Stack is useful for implementing

(a) recursion (b) breadth first search

(c) depth first search (d) both (a) and (c)

147. A machine took 200 sec to sort 200 names, using bubble sort. In 800 sec, it can approximately sort

(a) 400 names (b) 700 names

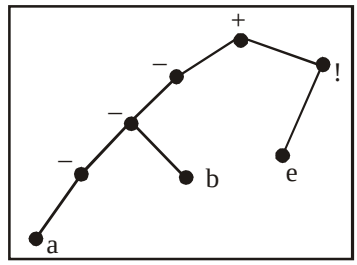
(c) 750 names (d) 800 names

148. Which of the following is useful in implementing heap sort?

(a) Stack (b) Set

(c) List (d) Queue

149. The expression tree given in Fig. evaluates to 1, if



1. a = – b and e = 0
2. a = – b and e = 1
3. a = b and e = 0
4. a = b and e = 1

(a) 2 only (b) 3 and 4

(c) 1 and 2 (d) 1 and 4

150. Unrestricted use of goto is harmful, because it

(a) makes debugging difficult

(b) increases the running time of programs

(c) increases memory requirement of programs

(d) results in the compiler generating longer machine code

151. The maximum degree of any vertex in a simple graph with n vertices is

(a) n (b) n – 1

(c) n + 1 (d) 2n – 1

152. The recurrence relation that arises in relation with the complexity of binary search is

(a) T(n) = T(n/2) + k, where k is a constant

(b) T(n) = 2T(n/2) + k, where k is a constant

(c) T(n) = T(n/2) + log(n)

(d) T(n) = T(n/2) + n

NUMERICAL TYPE QUESTIONS

153. The expression 4 + 6 / 3 * 2 – 2 + 7% 3 evaluates to

154. Consider the following program segment.

```
i = 6720; j =4 ;
while (i%j) = = 0)
{i = i/j;
j = j + 1;
}
```

On termination j will have the value

155. If 7 bits are used to store a character, the percentage reduction of needed storage will be

156. Consider the following program fragment.

```
d = 0;
for (i = 1; i < 31; ++i)
```

```
for (j = 1; j < 31; ++k)
 for (k = 1; k < 31; ++k)
  if (((i + j + k) % 3) = = 0)
      d = d + 1;
printf (“%d”, d);
```

the output will be

157. Suppose one character at a time comes as an input from a string of letters. There is an option either to (i) print the incoming letter or to (ii) put the incoming letter on to a stack. Also a letter from top of the stack can be popped out at any time and printed. The total number of total distinct words that can be formed out of a string of three letters in this fashion, is. Enter a value.

158.
```
#include<stdio.h>
#define EOF -1
void push(int); /* Push the argument on the stack */
int pop(void); /* pop the top of the stack */
void flagError();
int main( )
{ int c, m, n, r;
while ((c = getchar( )) != EOF)
{ if (isdigit(c))
push  (c)
else if (c = = ‘+’) || (c = = ‘*’))
{ m = pop( );
n = pop( );
are = (c = = ‘+’) ? n + m : n*m;
push(r);
}
else if (c != ‘ ‘)
flagError( );
}
printf(“%c”, pop( ));
}
```

What is the output of the program for the following input?

5 2 * 3 3 2 + * +

HINTS & SOLUTIONS

PAST GATE QUESTIONS EXERCISE

1. **(a)** a + b × c – d ^ e ^ f
a + b × c – d ^ e f ^
a + b × c – d e f ^ ^
a + b c × – d e f ^ ^
a b c × – d e f ^ ^
a b c × + d e f ^ ^
a b c × + d e f ^ ^–
the result is obtained.

2. **(a)**

3. **(c)** As given p is a single variable that is used to access the queue and with the single variable it is not possible to perform both the functions of enqueue and dequeue with a constant time since, insertion and deletion are operations that needs to be done from the opposite end of the queue rear and front respectively.

4. **(b)** The program undergoes normal execution upto line number 7 since, no error is present til there but as soon as the execution goes to line 8 an error occurs as in this the pointer s is assigned a character variable, i.e., p and this assignment is not permissible in C language thus, the program produces a garbage value as an output or no output is returned.

5. **(c)** The iterations that the given code will undergo are:
From the given conditions we does not take into account when n = 1
Iteration 1
N = 1 + 1 = 2 therefore, i = 2
Iteration 2
N = 2 + 2 = 4 therefore, i = 3
Iteration 3
N = 4 + 3 = 2 = 4 therefore, i = 4
Hence, the value returned after three iterations is 7
When, i = 4 and it also fulfill the condition of n>=5

6. **(a)** Inorder traversal of a BST always gives elements in increasing order. Among all four options, (a) is the only increasing order sequence.

7. **(c)** Whenever the a function's data type is not declared in the program, the default declaration is taken into consideration. By default, the function foo in this is assumed to be of int type, then also it returns a double type of value. This situation will generate compiler warning due to this mismatch leading to unintended results.

8. **(d)** sum has no use in foo(), it is there just to confuse. Function foo() just prints all digits of a number. In main, there is one more printf statement after foo(), so one more 0 is printed after all digits of n.
From the given code it is found that foo is a recursive function and when the function foo (a, sum) is called where a = 2048 and sum = 0 as per given conditions.
Then, the execution of the function takes in the following manner.
i) k= n % 10 => modulus operator will return the remainder. for example, if n=2048, then 2048 %10 will store the value 8 in k.
ii) j is declared as integer datatype in foo function. Therefore after division if the result contains decimal part, it will be truncated and only the integer part will be stored in j. For example if j=2048/10, (actual result = 204.8) then 204 will be stored in j.
iii) The sum variable declared in the main function will not be used by the foo function.

9. **(a)** We have to store frequencies of scores above 50. That is number of students having score 51, number of students having score 52 and so on. For that an array of size 50 is the best option.

10. **(b)** Object Oriented Programming (OPP) is a programming paradigm. The language is object oriented as it use objects. Objects are the data structures that contain data fields and methods together with their interactions.
The main features of the Programming techniques are
1. data abstraction
2. encapsulation
3. modularity
4. polymorphism
5. inheritance

Therefore, the essential features are given by statements (i) and (iv).

11. **(c)** Languages needs declaration of any statement that we write before its use thus, the common property of both the languages is that both are declarative.

12. **(c)** An **abstract data type** is a mathematical model for a certain class of data structures that have similar behaviour. An abstract data type is indirectly defined by the operations and mathematical constraints thus, is a user defined data type, not a system defined, that can perform operations defined by it on that data.

13. **(c)** Syntax to declare pointer to a function =>datatype (*pointer_variable)(list of arguments)
To make a pointer to a function =>pointer_variable = function_name
Note: don't use parenthesis of the function.
To call (invoke) the function =>pointer_variable(list of arguments)

14. **(b)** For the given code only the statements S2 and S3 are valid and thus, are true. Since, the code may generate a segmentation fault at runtime depending on the arguments passed as the arguments are passed by reference and also the swap procedure is correctly implemented.

15. **(d)** Both functions work1 &work2 performs the same task, therefore S1 is true.

In S2 it is asking about improvement in performance i.e. reduction in CPU time. When compared work2 will reduce the CPU time, because in work1 a[i+2] is computed twice but in work2 a[i+2] is computed once and stored in t2, and then t2 is used. When we consider the performance in terms of reduction in CPU time, S2 is correct.

16. **(c)** The condition (i) is true if the last inserted element in c[] is from a[] and condition (ii) is true if the last inserted element is from b[].

17. **(a)** The order in which insert and delete operations are performed matters here.

The best case: Insert and delete operations are performed alternatively. In every delete operation, 2 pop and 1 push operations are performed. So, total m+ n push (n push for insert() and m push for delete()) operations and 2m pop operations are performed.

The worst case: First n elements are inserted and then m elements are deleted. In first delete operation, n + 1 pop operations and n push operation are performed. Other than first, in all delete operations, 1 pop operation is performed. So, total m + n pop operations and 2n push operations are performed (n push for insert() and m push for delete())

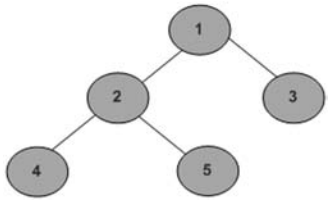
18. **(c)** A node is a leaf node if both left and right child nodes of it are NULL.

1) If node is NULL then return 0.

2) Else If left and right child nodes are NULL return 1.

3) Else recursively calculate leaf count of the tree using below formula.

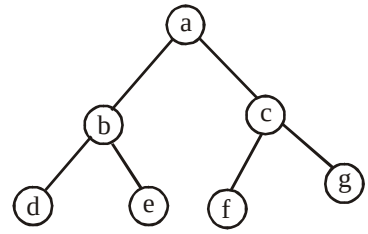
Leaf count of a tree = Leaf count of left subtree +
Leaf count of right subtree



Example Tree

Leaf count for the above tree is 3.

19. **(a)** The inorder traversal sequence is dbeafcg and the preorder traversal sequence is abdecfg so, the tree is



In the postorder traversal, the sequence is debfgca.

20. **(d)** We follow, the following steps to obtain the value of f(5)

f(5)
r = 5
f(3) + 2 = 18
↑
f(2) + 5 = 16
↑
f(1) + 5 = 11
↑
f(0) + 5 = 6
↑
1

21. **(a)** The algorithm for evaluating any postfix expression is fairly straightforward:
1. While there are input tokens left
o Read the next token from input.
o If the token is a value
+ Push it onto the stack.
o Otherwise, the token is an operator
(operator here includes both operators, and functions).
* It is known a priori that the operator takes n arguments.
* If there are fewer than n values on the stack
(Error) The user has not input sufficient values in the expression.
* Else, Pop the top n values from the stack.
* Evaluate the operator, with the values as arguments.
* Push the returned results, if any, back onto the stack.
2. If there is only one value in the stack
o That value is the result of the calculation.
3. If there are more values in the stack
o (Error) The user input has too many values.
Let us run the above algorithm for the given expression.
First three tokens are values, so they are simply pushed.
After pushing 8, 2 and 3, the stack is as follows
8, 2, 3
When ^ is read, top two are popped and power(2^3) is calculated
8, 8
When / is read, top two are popped and division(8/8) is performed
1
Next two tokens are values, so they are simply pushed.
After pushing 2 and 3, the stack is as follows
1, 2, 3
When * comes, top two are popped and multiplication is performed.
1, 6

22. (d) From the statement j = j*2 in the code we get to know that j increases in power of 2's. Let us say that this statement execute x times then, according to the question for while loop

$2^x <= n$

Therefore, $x <= \log_2 n$

And also for termination of while loop there will be an extra comparison required. Thus, total number of comparisons = x + 1

$= [\log_2 n] + 2$

23. (b) The function rearrange() exchanges data of every node with its next node. It starts exchanging data from the first node itself.

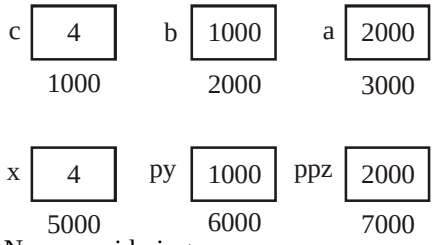
24. (d) The option chosen is

?1 is ((c = getchar ( ))! = '\n')

?2 is putchar (c);

Because the operator '1=' has higher priority than '=' operator so according to this C = getchar () should be contained in brackets and when the string is reversed then the function putchar (c) is used so that the characters can be printed.

25. (b) The program gets executed in the following manner Graphical Representation



Now, considering

int y, z;

**ppy + = 1; z = *ppz = 6

*py + = 2; y = *py = 6

x = 4 + 3 = 7

return x + y + z;

and

c = 4; b & c; a = &b;

printf ("%d", f(c, b, a)),

From the code,

The output is printed as 6 + 6 + 7 = 19.

26. (a) The operator "?:" in C is the ternary operator which means that, if the expression is exp 1? exp2 : exp3, so it means, if exp1 is true then exp2 is returned as the answer else exp3 is the required answer.

So, in the given expression let us consider x = 3, y = 4, z = 2,

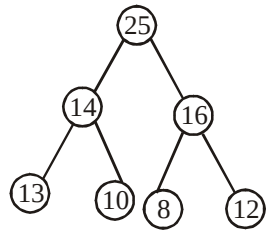
Then, expression becomes a

= (3 > 4)? ((3 > 2)? 3:2): ((4>2?4:2)

From this, we get that 3>4 is false so we go for the else part of the statement which is 4>2 and is true thus, the answer is 4, the true part of the statement.

27. (c) Suppose that we have a node x, then for the condition $1 <= x <= n/2$ and $A[x] >= A[2x + 1]$ where 2x and 2x + 1 are left child and right child of the node x respectively of a binary max-heap.

Thus, under given cases the tree obtained is.



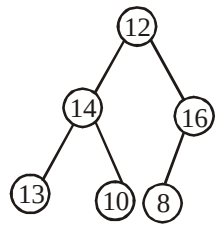
28. (d) Always a greater element is deleted from the binary heap first. The answer of previous question gave the array as [25, 14, 16, 13, 10, 12, 8]

So, when the greatest element, i.e., 25 is deleted the array becomes [14, 16, 13, 10, 12, 8]

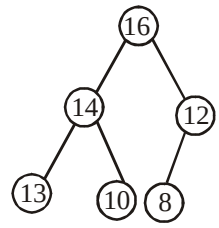
And next after second deletion the array becomes [14, 13, 10, 12, 8]

Thus, the procedure for the obtaining the final tree is as follows.

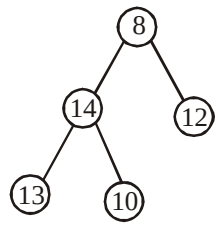
Replacing 25 with



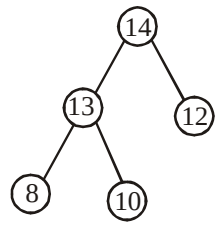
After heapifying



Replacing 16 by 8



After heapifying



29. (c) 12 mod 10 = 2

18 mod 10 = 8

13 mod 10 = 3

2 mod 10 = 2 collision

(2 + 1) mod 10 = 3 again collision

(using linear probing)

(3 + 1) mod 10 = 4

3 mod 10 = 3 collision

(3 + 1) mod 10 = 4 again collision

(using linear probing)

(4 + 1) mod 10 = 5

23 mod 10 = 3 collision

(3 + 1) mod 10 = 4 collision

(4 + 1) mod 10 = 5 again collision

(using linear probing)

(5 + 1) mod 10 = 6

5 mod 10 = 5 collision

(5 + 1) mod 10 = 6 again collision

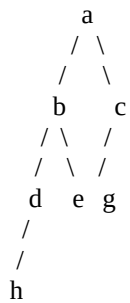
(6 + 1) mod 10 = 7

15 mod 10 = 5 collision

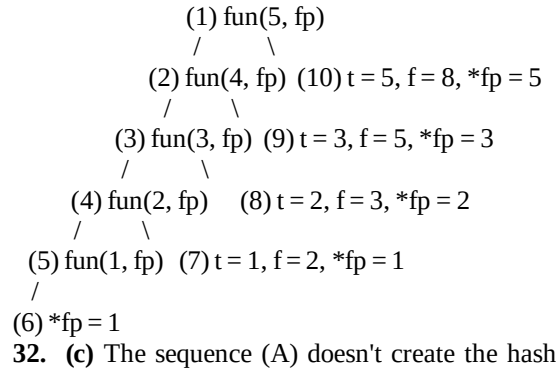
(5 + 1) mod 10 = 6 collision
(6 + 1) mod 10 = 7 collision
(7 + 1) mod 10 = 8 collision
(8 + 1) mod 10 = 9 collision
So, resulting hash table

0	
1	
2	12
3	13
4	2
5	3
6	23
7	5
8	18
9	15

30. (b) AVL trees are binary trees with the following restrictions.
1) the height difference of the children is at most 1.
2) both children are AVL trees



31. (b) The program calculates nth Fibonacci Number. The statement t = fun (n-1, fp) gives the (n-1)th Fibonacci number and *fp is used to store the (n-2)th Fibonacci Number. Initial value of *fp (which is 15 in the above prgram) doesn't matter. Following recursion tree shows all steps from 1 to 10, for exceution of fun(5, &x).



32. (c) The sequence (A) doesn't create the hash table as the element 52 appears before 23 in this sequence.
The sequence (B) doesn't create the hash table as the element 33 appears before 46 in this sequence.
The sequence (C) creates the hash table as 42, 23 and 34 appear before 52 and 33, and 46 appears before 33.
The sequence (D) doesn't create the hash table as the element 33 appears before 23 in this sequence.

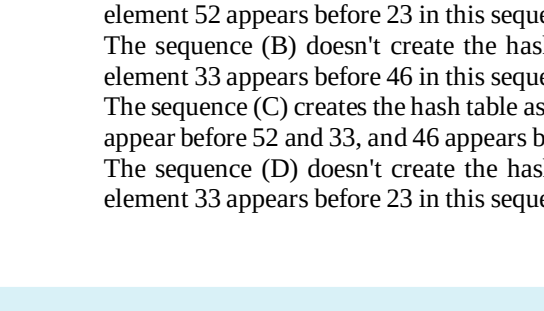
33. (c) In a valid insertion sequence, the elements 42, 23 and 34 must appear before 52 and 33, and 46 must appear before 33.
Total number of different sequences = 3! x 5 = 30
In the above expression, 3! is for elements 42, 23 and 34 as they can appear in any order, and 5 is for element 46 as it can appear at 5 different places.

34. (d) In a given program we take the test cases and apply.
First take T_1, if all value equal means
a = b = c = d
So, due to T_1, a = b condition satisfied and S_1 and S_4 executed.
So, from T_2 when all a, b, c, d distinct.
S_1, S_2 not execute, S_3 execute.
from T_3 when a = b then, S_1 execute but c = d so, S_2 not execute but S_3 and S_4 execute but we have no need of T_3 because we get all result from above two.
By using T_4. If a! = b and c = d
So, S_1 not execute and S_2 and S_4 execute so all of S_1, S_2, S_3, S_4 execute and covered by T_1, T_2 and T_4.

35. (b) Let AX, BX, CX be three registers used to store results of temporary variables a, b, c, d, e, f.
a(AX) = 1
b(BX) = 10
c(CX) = 20
d(AX) = a(AX) + b(BX)
e(BX) = c(CX) + d(AX)
f(AX) = c(CX) + e(BX)
b(BX) = c(CX) + e(BX)
e(BX) = b(BX) + f(AX)
d(CX) = 5 + e(BX)
return d(CX) + f(AX)
Thus, only 3 registers can be used without any overwriting of data.

36. (d) When the while loop ends, q contains address of second last node and p contains address of last node. So we need to do following things after while loop.
(i) Set next of q as NULL (q->next = NULL).
(ii) Set next of p as head (p->next = head).
(iii) Make head as p (head = p)
Step (ii) must be performed before step (iii). If we change head first, then we lose track of head node in the original linked list.

37. (c) f() is a recursive function which adds f(a+1, n-1) to *a if *a is even. If *a is odd then f() subtracts f(a+1, n-1) from *a. See below recursion tree for execution of f(a, 6).



|
4 + f(add(11), 2)
|
|
11 - f(add(6), 1)
|
|
6 + 0

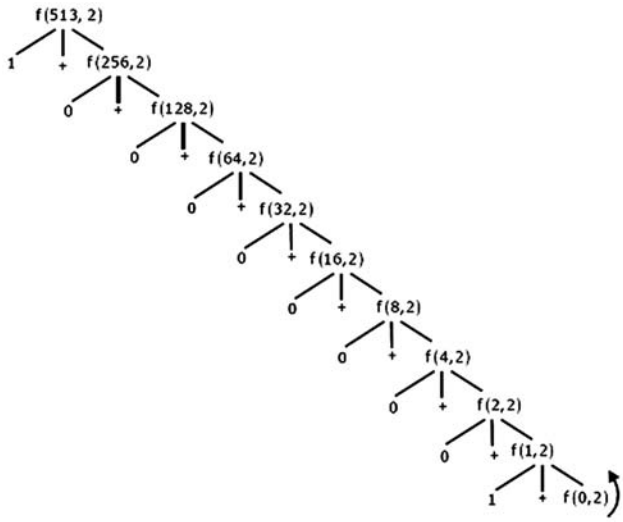
So, the final returned value is 12 + (7 - (13 - (4 + (11 - (6 + 0))))) = 15

38. (c) linked **data structures allow** more flexibility in **and the** implementation of **these** linked **data structure** is through **dynamic data structures**

39. (d)
```
* p points to i and q points to j */
void f(int *p, int *q)
{
p = q;    /* p also points to j now */
*p = 2;   /* Value of j is changed to 2 now */
}
```

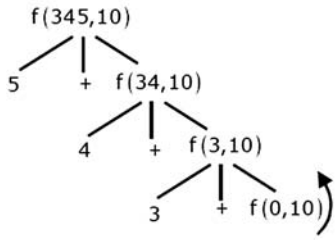
40. (a) Such a binary tree is full binary tree (a binary tree where every node has 0 or 2 children).

41. (d)



1 + 1 = 2

42. (b)



5 + 4 + 3 = 12

43. (d) R1 ← c, R2 ← d, R2 ← R1 + R2, R1 ← e, R2 ← R1-R2
Now to calculate the rest of the expression we must load a and b into the registers but we need the content of R2 later.
So we must use another Register.
R1 ← a, R3 ← b, R1 ← R1-R3, R1 ← R1+R2

44. (b) It is stated that there is a binary tree and we have populate the tree with n elements. Sorting the n elements in the increasing order,and placing them in the inorder traversal nodes of the binarytree makes it only BST possible.

45. (c) For n = 8
$f_1(n) = 2^n = 2^8 = 256$
$f_2(n) = n^{3/2} = 8^{3/2} = \sqrt{8\times8\times8} = 8 \times 2\sqrt{2} = 16\sqrt{2}$
$f_3(n) = n \log_2 n (= 8 \log_2 8 = 8 \times 3 = 24$
$f_4(n) = {}_n{}^{\log_2 n} = {}_8{}^{\log_2 8} = 8^3 = 64 \times 8 = 512$
$\Rightarrow f_2, f_3, f_1, f_4$

46. (c) M1 × M2 × M3
For $(M_1 \times M_2) \times M_3 = (p \times q \times r) + (p \times r \times s)$
$= (10 \times 100 \times 20) + (10 \times 20 \times 5)$
$= 20000 + 1000 = 21000$
$M_1 \times (M_2 \times M_3) = (p \times q \times s) + (q \times r \times s)$
$= (10 \times 100 \times 5) + (100 \times 20 \times 5)$
$M_1 \times (M_2 \times M_3) < (M_1 \times M_2) \times M_3$
$= (10 \times 100 \times 5) + (100 \times 20 \times 5)$
$M_1 \times (M_2 \times M_3) < (M_1 \times M_2) \times M_3$
This, $(M_1 \times (M_2 \times M_3))\ M_4 = 15000 + p \times s \times t$
$= 15000 + 10 \times 5 \times 80$
$= 15000 + 4000 = 19000$

47. (c) 2011 charc[] = "GATE2011";
p now has the base address string "GATE2011"
char*p =c;
p[3] is 'E' and p[1] is 'A'.
p[3] - p[1] = ASCII value of 'E' - ASCII value of 'A' = 4
So the expression p + p[3] - p[1] becomes p + 4 which is base address of string "2011

48. (a) The algorithm has dynamic programming paradigm as i has been chosen randomly.

49. (b) A binary tree is max-heap if it is a complete binary tree (A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible) and it follows the max-heap property (value of each parent is greater than or equal to the values of its children).

(a) is not a max-heap because it is not a complete binary tree
(b) is a max-heap because it is complete binary tree and follows max-heap property.
(c) is not a max-heap because 8 is a child of 5 in this tree, so violates the max-heap property.
(d) is not a max-heap because 8 is a child of 5 in this tree, so violates the max-heap property. There are many other nodes in this tree which violate max-heap property in this tree.

50. (d) Squaring the weights of the edges in a weighted graph will not change the minimum spanning tree. Assume the opposite to obtain a contradiction. If the minimum spanning tree changes then at least one edge from the old graph G in the old minimum spanning tree T must be replaced by a new edge in tree T' from the graph G' with squared edge weights. The new edge from G' must have a lower weight than the edge from G. This implies that there exists some weights C1 and C2 such that C1 < C2 and C12 >= C22. This is a contradiction.
Sums of squares of two or more numbers is always smaller than square of sum.
Example: $2^2 + 2^2 < 4^2$

51. **(a)** FRONT points to the first element that we can remove and then we increment FRONT REAR points to the first empty space in queue so we insert an element and increment REAR so now initially queue is empty F=R=0 insert A (AT 0th index) --> increment REAR (REAR =1 FRONT=0) now if we want to delete A we have FRONT pointing at that location so we delete A and increment FRONT (REAR=1 FRONT=1 queue empty)

52. **(d)** Link to activation record of closest lexically enclosing block in program text. It depends on the static program text

53. **(d)** Dijkstra's Algorithm picks nodes as follows:
B(3) from S
A(4) from S
C(5) from A
E(6) from C
D(7) from S or B
G(8) from E
T(10) from D or E
Thus the shortest path to T is SBDT, SDT or SACET.
But because of "the shortest path to a vertex v is updated only when a strictly shorter path to v is discovered", when E is visited, T's prior node for the shortest path will be set as E and not changed again.
Thus the answer is SACET.

54. **(a)** The box B1 gets executed when left subtree of n is NULL and right sbtree is not NULL. In this case, height of n will be height of right subtree plus one.
The box B2 gets executed when both left and right subtrees of n are not NULL. In this case, height of n will be max of heights of left and right sbtrees of n plus 1.

55. **(c)** The number of elements that can be sorted in Θ (log n) time using heap sort is $\Theta\left(\frac{\log n}{\log\log n}\right)$

Consider the number of elements is "k", which can be sorted in θ (k log k) time.
Analyzing the options in decreasing order of complexity since we need a tight bound i.e., θ.

i.e., θ (log n), $\Theta\left(\frac{\log n}{\log\log n}\right), \Theta(\sqrt{\log n}), \Theta(1)$

So if $k \in \Theta$ (log n) time required for heap sort is O (k log k) i.e.,
θ (log n × log log n), But this is not in Θ (log n)

If $k \in \Theta\left(\frac{\log n}{\log\log n}\right)$ time required for Heap Sort

$$\Theta\left(\frac{\log n}{\log\log n}\times\log\left(\frac{\log n}{\log\log n}\right)\right)$$

i.e., $$\Theta\left(\log n\times\left[\frac{\log\left(\frac{\log n}{\log\log n}\right)}{\underbrace{\log\log n}_{\le 1}}\right]\right)$$

So, this is in Θ (log n)

Hence, answer is (c) $\Theta\left(\frac{\log n}{\log\log n}\right)$

56. **(b)** The return value of the function is θ (n^2 log n)

The outer for loop goes $\frac{n}{2}$ + 1 iterations.

The inner for loop runs independent of the outer loop.

And for each inner iteration, $\frac{n}{2}$ gets added to k.

$\therefore$ $\frac{n}{2}$ × # outer loops × # inner loops per outer loop.

Inner loops = θ (log n) [$\because 2^{\theta(\log n)} = \theta(n)$]

$$\therefore \frac{n}{2}\times\left[\frac{n}{2}+1\right].\theta(\log n) = \theta(n^2\log n)$$

57. **(d)**

58. **(b)** Return value f (p, p) if the value of p is intialized to 5 before the call.
Since, reference of p is passed as 'x'
Thus, any change in value of x in f would be reflected globally.
The recursion can be broken down as

$f^{5\ 5}(x,c)$

$x * f^{6\ 4}(x,c)$

$x * f^{7\ 3}(x,c)$

$x * f^{8\ 2}(x,c)$

$x * f^{9\ 1}(x,c)$

59. **(c)** Analyzing the flow in the function. Fix an 'i' in the outer loop.
Suppose A[i] = old c[0], then A[i] is set to new c[0] [j = 0].
Now suppose that old c[1]= new c[0]
$\Rightarrow$ A[i] = old c[1] (after the j = 0th iteration)
which means in the next iteration with j = 1; A[i] again gets changed form new c[0] (or old c[1]) to new c[1] which is incorrect.
The current loop should "break" after a match is found and A[i] is replaced.
The inner loop should look like

```
for   (int j = 0; j < 3, j + +)
if    (A [i] = = old c[j])
      A[j] = new c [j];
break;
```

The above flow is exposed by the test cases 3 & 4 only. For test 1 and test 2, the expected output is

produced by the code.
Test 3 Expected A after replacements
= "abcde" → "acdde"
result from code = "abcde" → "addde"
Test 4 Expected "abcde" → "bacde"
result form code "abcde" → "abcde".
∴ Answer is 3 and 4 only (c)

60. (b) The FLAWS are identified by a test case of the following kind.
old c = O_1 O_2 O_3
new c = $n_1 n_2 n_3$

with $\begin{pmatrix} n_1 & = & O_2 \\ n_1 & \neq & n_2 \end{pmatrix}$ or $\begin{pmatrix} n_2 & = & O_3 \\ n_2 & \neq & n_3 \end{pmatrix}$

Hence, the answer is (b) only two.
1 (∵ c = 0 in this call)
∴ Answer is = x^4 * x * x * x.
The final value of x = 9
∴ 9^4 = 6561 i.e., Option (b)

61. (d) We concentrate on following code segment:
→ int *Pi = & i ;
Nothing wrong as 'Pi' is declared as integer pointer and is assigned the address of 'i' which is an "int".
→ scanf ("%d", Pi);
We know that scanf ( ) has two arguments:
First is control string ("%d" in this case), telling it what to read from the keyboard.
Second is the address where to store the read item.
So, 'Pi' refers to the address of "i", so the value scanned by scanf ( ) will be stored at the address of 'i'.
Above statement is equivalent to:
scanf ("%d", & i);
Finally the best statement:
printf ("%d\n", i + 5);
It prints the value 5 more than the value stored in i.
So, the program executes successfully and prints the value 5 more than the integer value entered by the user.

62. 12
No. of Bag = 5
All coins in a bag have same weight.

Labels:	1	2	3	4	5
No. of coins	1	2	4	8	16

As total weight is 323, the Bag with label 1 must contain coins of 11 gm to make the total odd.
If we remove it the remaining weight = 323 – 11 = 312 gm

Label:	2	3	4	5
No. of coins:	2	4	8	16
Weights: (I):	10	11	11	10

Total weight = (2 × 10) + (4 × 11) + (8 × 11) + (16 × 10)
= 20 + 44 + 88 + 160
= 312
So, label 1, 3, 4 contains coins of 11 gm.
Multiplying these label numbers we get
1 × 3 × 4 = 12.

63. (c) One-sixth of the product of the 3 consecutive integers.

64. (a)
Clearly the code segment: for (i = 0; i < size; i + +) Y = Y + E [i]
Computes the sum of all element in the array E.
Note : As the array E may contain negative elements, the sum of elements of subarray of E may be greater than the sum of all elements in the array E.
Now, comes the nested i, j, k loops:
'i' changes the starting of the subway.
'j' changes the end position of the subarray.
'k' loop is used to compute sum of elements of the particular subarray.
The sum of subarray is stored in z and is compared to the earlier max sum Y. Finally the larger sum gets stored in Y.
i = 0, j = 14 ⟷
i = 0, j = 2 ⟷
i = 0, j = 1 ⟷
i = 0, j = 0 ⟷

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

size = 15
i = 2, j = 2 ⟷
i = 2, j = 3 ⟷
i = 2, j = 4 ⟷
⋮
i = 2, j = 14 ⟷
This can be easily seen how sum of subarray is computed with different values of i and j.

65. 9
435 ⇒ 110110011
num > > 1
Thus a num is shifted one bit right every time while loop is executed.
While loop is executed 9 times successfully, and 10^{th} time num is zero.
∴ Count is incremented 9 times.

66. (b) P = $^nC_3$

$= \frac{n(n-1)(n-2)}{6}$

If we multiply, n, (n – 1) and (n – 2) together, it may go beyond the range of unsigned integer. (∴ option (A) and (D) are wrong)
For all values of n, n(n – 1)/2 will always be an integer value
But for n(n – 1)/3, it is not certain.
Take options (b)

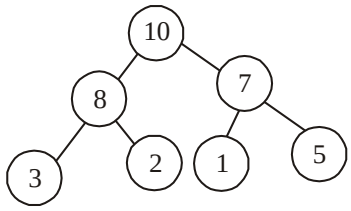
$P = \frac{n(n-1)/2}{P_1} \times \frac{(n-2)/3}{P_2}$

P_1 will be having no error, thus P will be more accurate
Option (c)

$$P = \frac{n(n-1)/3}{P_1} \times \frac{(n-2)/2}{P_2}$$

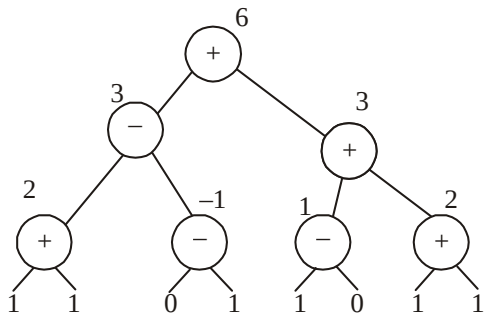
There is possibility of truncation in P_1, the accuracy of P is less

67. (a)



After insertion of elements,
level - order tansvasal is :
10, 8, 7, 3, 2, 1, 5

68. 6



69. 1.72 to 1.74

$$f(g) + q \Rightarrow \frac{x}{2} + \frac{1.5}{x} = x$$

$$\Rightarrow \frac{x^2+3}{2x} = x \Rightarrow x^2 + 3 = 2x^2$$

$$\Rightarrow x^2 = 3 \Rightarrow x = 1.73$$

70. (c) The queue can be implemated where ENQUEUE takes a sequence of three instructions (reverse, push, reverse) and DEQUEUE takes a single instruction (pop).

71. (d) When j = 50, j (50), the function goes to an infinite loop by calling f(i) recursively for i = j = 50

72. (a) For the given pseudo code, each element of the upper triangle is interchanged by the corresponding element in lower triangle and later the lower triangular element is interchanged by the upper triangular once. Thus the final output matrix will be same as that of input matrix A.

73. 7

$P(x) = x^5 + 4x^3 + 6x + 5$
$= x^3 (x^2 + 4) + 6x + 5$
Now using only one temporary variable 't'
(i) $t = x * x$ (Evaluate x^2 and store in memory)
(ii) $t = t + 4$ (Evaluate $(x^2 + 4)$ and store in memory)
(iii) $t = x^2$ (Retrieve x^2 from memory)
(iv) $t = t * x$ (Evaluate x^3 and store in memory)
(v) $t = t * (x^2 + 4)$ (Evaluate $x^3 (x^2 + 4)$ and store in memory)
(vi) $t = 6 * x$ (Evaluate 6x and store in memory)
(vii) $t = t + 5$ (Evaluate $(6x + 5)$ and store in memory)
(viii) $t = t + x^3 (x^2 + 4)$ (Retrieve $x^2 (x^2 + 4)$ from memory and evaluate $\{x^3 (x^2 + 4) + 6x + 5\}$
For the above steps, total number of arithmetic operation is 7
i.e., 4 multiplications and 3 additions.

74. (a) The in order transversal is as :
left, root, middle, right
$\therefore$ Nodes are visited in SQPTRWUV order.

PRACTICE EXERCISE

1. (d) BCPL: Basic Combined Programming Language primary used for system software and called simply B.

2. (a) Functions can return any type supported by C (except for arrays and functions), including the pointers, structures and unions

3. (a) a **preprocessor** is a program that processes its input data to produce output that is used as input to another program. The output is said to be a **preprocessed** form of the input data, which is often used by some subsequent programs like compilers

4. (c) **5. (b)**

6. (a) Blank space between the "." (double quotes) increments the length by '1', 'correct' has length seven and 'correct string' has length 14.
Hence option (a) is correct.

7. (d) The C Preprocessor is a part of the C compilation process that recognizes special statements, analyses them (before compilation) and acts on them as required. The preprocessor can be used to make programs more efficient, more readable, and more portable to multiple systems.

8. (c)

9. (d) enum keyword assigns the integer values in ascending order if not specified, here bethlehem will be assigned zero, jericho will assigned '1' nazareth is already '1' and then increasing '1' we, get Jerusalem as '2' hence (d) is correct option.

10. (c) The value of automatic variable that is declared but not initialized can give garbage value although some implementations or compilers give it default value as zero, hence value is unpredictable.

11. (b) The given program fragment checks the condition a = = b (a = 4, b = 6) hence cond^n becomes false and returns zero.

12. (d) **13. (b)**

14. (d) int val [2] [4] = {1, 2, 3, 4, 5, 6, 7, 8} is a 2-dimensional array

1	2	3	4
5	6	7	8

treating the array declaration in the row-major '4' comes

under the first row and fourth column which has the index value val [0] [3] hence (d) is correct option.

15. (d) The standard recommends the implementations to accept at least 256 (ISO 14882, B.2), but they may support less or more:

16. (a) Of the option (a), (b), (c), (d) the array index value start from zero and ends with 'g', since array has only three elements, the remaining values are treated as zero or garbage value only option (a) is satisfying the condition.

17. (c) Here by using typedef we are making a new data type of x having size equal to 10 bytes (size of one character is 1 byte). In x my Array (s) we are declaring my Array of size '5' of the data type x hence size of my Array becomes 5 × 10 = 50 bytes.

18. (d)

19. (d) If global variables are by default static then it means that we would be able to access them in a single file.But we can use global variables in different files as well.

20. (a) The procedure cannot change the variable or any of its members.

21. (d)

22. (b) In the body of 'main' () the printf statement prints 'tim" followed by calling of the main () itself which in turn again "tim" and call main () this process never ends and keeps on printing 'tim'

23. (c) Since 'Madam' is the required o/P, the function first (), should print 'a' call the function second (), that prints the 'd' and print 'a' again.
Hence (c) is the correct option.

24. (b) For the option (b) only 'a' and 'c' are compared but not 'b' hence largest of there nos. is on consideration of only a & c.

25. (d) We have already seen examples of its use when we have defined functions that return no value, i.e. functions which only print a message and have no value to return. Such a function is used for its side effect and not for its value. In the function declaration and definition, we have indicated that the function does not return a value by using the data type void to show an empty type, i.e. no value. Similarly, when a function has no formal parameters, the keyword void is used in the function prototype and header to signify that there is no information passed to the function.

26. (a) This given program will not compile successfully. The scope of the variable 'j' is the single 'printf' statement that follows it. So, the last statement that involves 'j' will complain about the undeclared identifier 'j'.

27. (b) By default x will be initialized to 0. Since its storage class is static, it preserves its exit value (and forbids reinitialization on re-entry). So, 123 will be printed.

28. (b) The compiler will tokenize a +++b as a, ++, +, b. So, a +++ b is equivalent to a +++b, which evaluates to 7.

29. (d) a +++++ b will be tokenized to a, ++, ++, ++, +, b. The compiler (parser), while grouping the tokens to expression, finds the second ++, applied to a++, an integer. Since ++ and ++b operator needs address (i.e., L-value), it will display the error message – Invalid I value in increment. So, to add a++ and ++b, use parenthesis or blanks to tokenize the way you intended.

30. (a) Consider arr [10][12] here arr [5] is logically pointing to the 6th row of the array this comes from the definition of the two-dimensional array.

31. (c) The statement int*b corresponds to a integer pointer which will contain address of some other variable. Similarly int **a will contain the address of some pointer variable which is linked any other variable holding any value (like int *b holding address of some variable) hence, it is semantically and syntactically correct.

32. (d) **C = 5, essentially means a = 5, as can be seen with the following pictorial representation of the given declarations.

Address	Value	Name
100	4	a
120	100	b
135	120	c

33. (d) If x is an array then ++X, X++, X*2 are trying to modify the address (X represent the base address of array) which is illegal. While X + 1 is equivalent to * (X [0] + 1) hence it is legal.

34. (d) num[i] is same as *(num+i) and num[i] is same as *(&num[i]) , hence both b and c are correct

35. (d) Choose the correct statements
We have seen how individual values (variables and pointers) can be passed to functions. Now let us see how we can pass an entire array to a function.
Suppose an array is defined as:
#define MAXSIZE 100
int myarr[MAXSIZE];
In order to pass the array myarr to a function foonction one may define the function as:
int foonction (int A[MAXSIZE] , int size)
{
...
}
This function takes two arguments, the first is an array of size MAXSIZE, and the second an integer argument named size. Here this second argument is meant for passing the actual size of the array. Your array can hold 100 integers
C++ does not allow to pass an entire array as an argument to a function. However, You can pass a pointer to an array by specifying the array's name without an index.
If you want to pass a single-dimension array as an argument in a function, you would have to declare function formal parameter in one of following three ways and all three declaration methods produce similar results because each tells the compiler that an integer

pointer is going to be received.
Way-1
Formal parameters as a pointer as follows:
```
void myFunction(int*param)
{
.
.
.
}
```
Way-2
Formal parameters as a sized array as follows:
```
void myFunction(int param[10])
{
.
.
.
}
```
Way-3
Formal parameters as an unsized array as follows:
```
void myFunction(int param[])
{
.
.
.
}
```

36. (d) We must associate a pointer to a particular type: You can't assign the address of a short int to a long int.pointer is a variable and thus its values need to be stored somewhere.
A pointer is definitely NOT an integer.

37. (d)

38. (a) a = .5, b = .7 then the first 'if' becomes false it goes to 'else' portion hence the result is "PSU".

39. (d) Pointers can be accessed along with structures. A pointer variable of structure can be created as below:
```
struct name {
member1;
member2;
.
.
};
-------- Inside function -------
struct name *ptr;
```
Here, the pointer variable of type struct name is created.
Structure's member through pointer can be used in two ways:
1. Referencing pointer to another address to access memory
2. Using dynamic memory allocation
Consider an example to access structure's member through pointer.
```
#include <stdio.h>
struct name{
int a;
float b;
};
int main(){
struct name *ptr,p;
ptr=&p;          /* Referencing pointer to memory address of p */
printf("Enter integer: ");
scanf("%d",&(*ptr).a);
printf("Enter number: ");
scanf("%f",&(*ptr).b);
printf("Displaying: ");
printf("%d%f",(*ptr).a,(*ptr).b);
return 0;
}
```
In this example, the pointer variable of type struct name is referenced to the address of p. Then, only the structure member through pointer can can accessed.
Structure pointer member can also be accessed using -> operator.
(*ptr).a is same as ptr->a
(*ptr).b is same as ptr->b

40. (c) r + = reading mode (only reading, writing possible).

41. (b) **42. (c)**

43. (d) printf ("%d", size of (" "));
*size of (" ") is pointing to the integer equivalent of " " which is equal to '1'.
Hence correct option is (d).

44. (a) printf (11%d", (a ++)); this given statement post-increments 'a' which means 'a' is printed first then it is incremented.

45. (d) The function definition
```
find (int x, int y)
{return ((x < y)? 0 : (x – y));}
```
determines the greater between the two x and y and gives 0 if x is minimum else gives (x – y) as result.
Now, the function call
(a, find (a, b))
is a rested call. first it calculate find (a, b). Then it gives the minimum of two.
Hence, it is used to find the minimum of a, b.

46. (d) The scanf function returns the number of successful matches. i.e., 3 in this case. a, b and c.

47. (b) The input is actually a/nb. Since we are reading only two characters, only a and \ n will be read and printed.

48. (b) If y = 11, the expression 3* (y – 8)/9 becomes 3*3/9. Which evaluates to 1. But the expression (y – 8) /9*3 becomes 3/9*3, which evaluates to 0 (since 3/9 is 0).

49. (c) 263 in binary form is 100000111. If one tries to print an integer as a character, only the last 8 bits will be considered – the rest chopped off. So, in this case the ASCII value of 00000111 (i.e., decimal 7) will be printed. Look in the ASCII table. It is ringing a bell!

50. (d) 9/5 yields integer 1. Printing 1 as a floating point number prints garbage.

51. (d) Here the loop initializes with the printing of 'c' followed by printing of 'a' then its prints 't'. after the first iteration is over it prints 'r'. then 'a' and then 't'.
This process continues.
hence o/p is catratratratrat....

52. (d) Initially i = 5 then 'if' condition is true and it returns to main () without executing any other statement. Hence nothing is printed.

53. (a) putchar (105) will print the ASCII equivalent of 105 i.e., 'i'. The printf statement prints the current value of i, i.e. 5 and then decrements it. So, h4 will be printed in the next pass. This continues until 'i' becomes 0, at which point the loop gets terminated.

54. (c) i + = 3 is equivalent to i = i + 3.
In the body of loop 'p' is incremented by '1'. Hence total of 4 is incremented overall in the current value of 'i'.
hence 3 7 11 is printed.

55. (a)

56. (d) The else clause has no brackets i.e., {and }. This means the else clause is made up of only one statement. So printf ("a < = b"); will be executed anyway, i.e., if a > b or a < = b. Hence. answer.

57. (a) **58. (d)**

59. (a) For a given program fragment, gives the O/P as a integer i.e. same as 8 * y.

60. (d) The given 'printf' statement prints' hello followed by / n (new line). 'puts' function can not print hello/n' at the same time, using the single 'puts' function.

61. (d) a[2] will be converted to * (a + 2).
* (a + 2) can as well be written as * (2 + 1).
* (2 + a) is nothing but 2 [a]. So, a [2] is essentially same as 2 [a], which is same as
* (2 + a). So, it prints 9 + 9 = 18. Some of the modern compilers don't accept 2[a].

62. (d) * a points to the string "abcd". ** a is the first character of "abcd", which is the character 'a'.

63. (a)

64. (b) 2 < 2 it conduction false so if go to else part and output Z1 Z2.

65. (a)

66. (b) The function abc can be invoked as abc () or (*abc) (). Both are two different ways of doing the same thing.

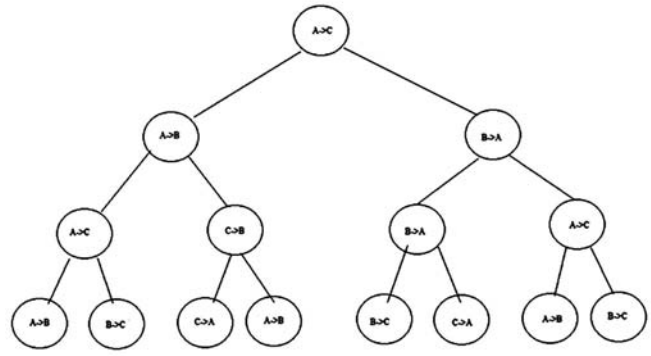
67. (a) It is not pointer so wer [1] = 162 and wer [1] + 1 = 162 + 1 = 163.

68. (b) **69. (d)**

70. (a) character pointer can be assigned to a void pointer but reverse is not true.

71. (a) The declaration char x [ ] = "success"; are equivalent hence O/P of 'puts (x)' and 'puts (y)' will be the same.

72. (a)



For one disk the steps involved is A->C ie 1 step
For two disks the steps involved are A->B,A->C,B->C ie 3steps
For three disks the steps involved are A->C,A->B,C->B,A->C,B->A,B->C,A->C ie 7steps
For 4 disks the steps involved are A->B,A->C,B->C,A->B,C->A,C->B,A->B,A->C,B->C,B->A,C->A,B->A,A->B,A->C,B->C i.e. 15 steps
So, for 1 disk $2^1 - 1 = 1$
2 disks $2^2 - 1 = 3$
3 disks $2^3 - 1 = 7$
4 disks $2^4 - 1 = 15$
n disks $2^n - 1$ (By induction)

73. (c) 20 20 10 10 20

String	Status of stack	Status of array(output)
Push(10)	10	
Push(20)	10 20	
Pop	10	20
Push(10)	10 10	20
Push(20)	10 10 20	20
Pop	10 10	20 20
Pop	10	20 20 10
Pop		20 20 10 10
Push(20)	20	20 20 10 10
Pop		20 20 10 10 20

74. (d) **75. (c)**

76. (a) The binary representation of odd numbers will have a 1 as the least significant digit. So, an odd number ANDed with 1, produces a 1. Even number end with 0. So, an even number Anded with 1, produces a 0. This for loop generates even and odd numbers alternatively. So, it prints alternate 0's and 1's.

77. (a)

78. (c) In the given program fragment, the initialisation c = d and d = a are invalid. Because it is trying to assign pointer to the pointer.

79. (d) The macro call will be expanded as
sqrt (a + 2 * a + 2 + b + 3 * b + 3)
i.e. sqrt (3 * a + 4 * b + 5). Hence the answer.

80. (d) Array and pointer are same in implementation. By using pointer we can access any of the index array. Array is a constant pointer and pointer is a one dimensional dynamic array.

81. (d)

82. (d) **83. (d)** **84. (b)** **85. (d)** **86. (b)**

87. (c) 263 in binary form is 100000111. If one tries to print an integer as a character, only the last 8 bits will be considered – the rest chopped off. So, in this case the ASCII value of 00000111 (i.e., decimal 7) will be printed. Look in the ASCII table. It is ringing a bell!

88. (c) **89. (c)** **90. (d)**

91. (d) Let ab be O × MN. N & f should yield 7 i.e., N & 1111 should produce 0111. So, N should be 0111, i.e., 7. Similarly, M can be found to be 2. So, ab is 0 × 27.

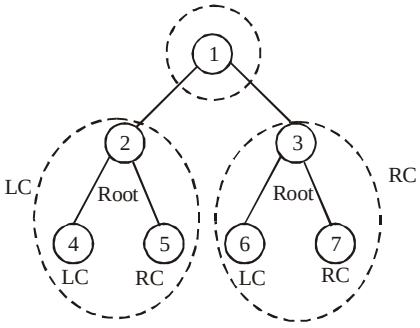
92. (d) **93. (a)** **94. (d)** **95. (a)** **96. (a)**
97. (c) **98. (d)** **99. (c)** **100. (b)** **101. (a)**
102. (b) **103. (c)** **104. (a)** **105. (b)** **106. (a)**
107. (d) **108. (a)** **109. (a)** **110. (d)** **111. (a)**
112. (a) **113. (c)** **114. (a)** **115. (a)** **116. (a)**
117. (b) **118. (c)** **119. (c)** **120. (b)**

121. (c) 5 – 2 – 3 * 5 – 2 will yield 18, if it is treated as (5 – (2 – 3)) * (5 – 2). i.e., if – has precedence over* and if it associates from the right.

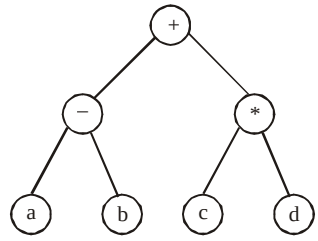
122. (a) 1112
In this program, We are calculating the adjacent difference of the given range by using function adjacent_difference.Output:$ g++ gnl.cpp$ a.out1 1 1 2

123. (c) **124. (d)** **125. (a)** **126. (a)**
127. (b) **128. (d)** **129. (d)** **130. (b)** **131. (c)**
132. (c) **133. (a)**

134. (a) Post order traversal = ab – cd* + as we know that post order traversal goes in the order of
LEFT CHILD → RIGHT CHILD → PARENT (ROOT)



Hence, Label of 4, 5, 2 will be a, b, – respectively
Label of 6, 7, 3 will be c, d, * respectively
Label of 1 will be +



hence (a) is the correct option.

135. (d) Since the tree is used for sorting hence taking INORDER TRAVERSAL (Left-Root-Right) we have 8 placed at the left child of the node labeled 10.
Hence (d) is the correct option.

136. (c) No. of possible binary search trees with n nodes = $\frac{2n C_n}{n+1}$

$\therefore$ for 3 nodes = $\frac{2 \times 3 C_3}{3+1} = \frac{6 C_3}{4} = 5$

137. (c) Insertion sort in best case = O(n)
and bubble sort = $O(n^2)$; selection sort = $O(n^2)$; merge sort = O (nlogn)

138. (b)

139. (b) Dijkstra's algorithm solves single source shortest path problem.
Warshall's algorithm finds transitive closure of a given graph.
Prim's algorithm constructs a minimum cost spanning tree for a given weighted graph.

140. (b) **141. (a)**

142. (d) In topological sorting we have to list out all the nodes in such a way that whatever there is an edge connecting i and j, i should precede j in the listing. For some graphs, this is not at all possible, for some this can be done in more than one way. Option (d) is the only correct answer for this question.

143. (c) Each comparison will append one item to the existing merge list. In the worst case one needs m + n – 1 comparisons which is of order m + n.

144. (b) It can be proved by induction that a strictly binary tree with 'n' leaf nodes will have a total of 2n – 1 nodes. So, number of non-leaf nodes is (2n – 1) – n = n – 1.

145. (b) **146. (d)**

147. (a) For sorting 200 names bubble sort makes 200 × 199/2 = 19900 comparisons. The time needed for 1 comparison is 200 sec (approximately). In 800 sec it can make 80,000 comparisons. We have to find n, such that n(n – 1)/2 = 80,000. Solving, n is approximately 400.

148. (a)

149. (c) The corresponding expression is – (– a – b) + e!. This is 1 if a = – b and e is either 1 or 0, since 1! = 0! = 1.

150. (a) **151. (b)** **152. (a)**

153. 4 + 6 / 3 * 2 – 2 + (7 % 3)
4 + (6/3)*2 – 2 + 1
4 + (2 * 2) – 2 + 1
(4 + 4) – 2 + 1
8 – 2 + 1
= 7

154. **Step-1**
i = 6720; j = 4; i % j = 0

$$i = \frac{6720}{4} = 1680$$

j = 5
Step-2
i = 1680; j = 5 i % j = 0
i = 336; j = 6
Step-3
i = 336; j = 6 i % j = 0
i = 56; j = 7
Step-4
i = 56; j = y i % j = 0
i = 8; j = 8
Step-5
i = 8; j = 8 i % j = 0
i = 1; j = 9
Step-6
i = 1; j = 9 i % j = 0
hence on termination j = 9

155. For each 8 bits one can save 1 bit. So percentage reduction will be 1/8* 100 i.e., 12.5%

156. a + b + c % 3 will be 0 if a + b + c is a multiple of 3. This will happen in one of the following ways. All three – a, b and c are multiples of 3. This can only happen if a, b and c take one of the 10 values, – 3, 6, 9, ... 30, independent of the one another. So, there are 10 × 10 × 10 = 1000 ways this can happen. Another possibility is that a, b and c all leave a remainder 1 so that a + b + c is evenly divisible by 3. Considering all the different possibilities and adding, we get 9000. That will be the integer that gets printed.

157. (b) Total no of distict word can be 5.
because total no words starting with a would be 2. (abc ,acb)
Total no of words starting with 2 (bca,bac)
Total no of words starting with c will be 1 (cba) because to print c as first letter we have to push a and b in stack
So total no of words formed = 2+2+1

158. (a) 25
push(5)
2. push(2)
3. m = 2;n=5 ;r=5*2
4. push(r) /*push(10)*/
5. push(3)
6. push(3)
7. push(2)
8. m=2; n= 3;r=3+2
9. push(r) /* push(5)*/
10. r=5*3
11. push(r) /*push(15)*/
12. m=15;n = 10;r=15+10
13. push(r)
14. printf("%d",pop());

Which will print 25