Programming and Data Structures

UNIT 3

PART A

Chapter 1

Programming in C

LEARNING OBJECTIVES

- *Basic concepts*
- *Character set*
- *Identifier*
- *Declaring a variable*
- *Visualization of declaration*
- *Constants*
- *Single character constants*
- *String constants*
- *Using const keyword*
- *Precedence decreases as we move from top to bottom*
- *Type conversion*
- *Documentation section*
- *Preprocessing*
- *Global declaration*
- *Control statements*
- *Selection/Decision making statement*
- *Looping statements*
- *Unconditional jump statements*

Basic Concepts

Character Set

A character refers to an alphabet, digit or a special symbol.
Alphabets: *A – Z*, *a – z*

Digits: 0 -9
Special symbols:
~ ! # % ∧ and * ( ) – + { } [ ] – < > , . | ? \ | : ; ” ’ White space

Identifier

Identifier is a user-defined name used for naming a variable or a function.
Rules for naming an identifier

- Consists only letters, digits and underscore
- Starts only with an alphabet or underscore
- Keywords cannot be used.
- Can be as long as you like, first 31 characters are significant.

Example: Valid identifiers: RollNo, Roll_No, _Roll_No
rollno, Name2;
Invalid: 2name, Roll No.

Variable

The name itself represents value, is not constant. Variable is a data name whose value varies/changes during program execution. Variable name is a name given to memory cell (may be one or multiple bytes).

Data Types

Represents type of data and set of operations to perform on data .

Data Type			
Primitive/Basic	**Derived**	**User defined**	**Valueless**
– Char	– Array	– Structure	
– float	– pointer	– union	– void
– double		Enumeration	
– integer			

Type	Keyword	Number of Bytes
Integer	int	2
Floating	float	4
Double	double	8
Character	char	1

Declaring a Variable

- Before using a variable, you must give some information to compiler about the variable. i.e., you must declare it.
- Declaration statement includes the type and variable name.

Syntax:
Datatype Var_name;
Example:

```
int roll_no;
char ch;
float age;
```

- When we declare a variable
  - memory space is allocated to hold a value of specified type.
  - space is associated with variable name
  - space is associated with a unique Address.

Table 1 *Visualization of declaration*

	roll no
int roll no;	garbage
	2002
	marks
int marks = 10;	10
	3008
	diameter
float diameter = 5.9	5.9
	4252
	ch → variable name
char ch : 'A'	A → value
	2820 → address

Note: The default value is garbage, i.e., an unknown value is assigned randomly.

Renaming data types with typedef Typedef is a keyword, which can form complex types from the basic type, and will assign some simpler names for such combinations. This is more helpful when some declaration is very tough, confusing or varies from one implementation to another.

For example, the data type unsigned long int is redefined as LONG as follows:

```
typedef unsigned long int LONG;
```

Uses of enumerated data types Enumerated data types are most useful when one is working over small, discrete set of values, in which each is having a meaning and it is not a number.

A best example can be given on months jan, feb, mar, …, dec, which are 12 in number, with assigning consecutive numbers for it.

The main advantages are storage efficiency, the *c*-code can become readable

Constants

A constant value is one which does not change during the execution of a program.
C supports several types of constants:

1. Integer constants
2. Real constants
3. Single character constants
4. Strings constants

Integer constants

An integer constant is a sequence of digits. It consists of a set of digits 0 to 9 preceded by an optional + or − sign spaces, commas, and non-digit characters are not permitted between digits.
Examples for valid decimal integer constants are
123
–31
0
562321
+78
Examples for invalid integer constants are
20,000
₹1000

Real constants

Real constants consist of a fractional part in their representation. Integer constants are inadequate to represent quantities that vary continuously.
Examples of real constants are
0.0026
–0.97
435.29
+487.0

Single character constants

A single character constant represents a single character which is enclosed in a pair of quotation symbols.
Examples for character constants are
'5'
'x'
';'

String constants

A string constant is a set of characters enclosed in double quotation marks. The characters in a string constant sequence may be alphabet, number, special character and blank space.
Examples of string constants are
"VISHAL"
"1234"
"C language"
"!….?"

Naming constants

A name given to a constant value. Value of name does not change during program execution.

Using const keyword

When we use 'const' with data type, memory will be allocated to variable and the initialized value does not change.
const int $x = 10$;
const float pi = 3.141;

Using # define

define x 10
define pi 3.141
Where '# define' is instruction to preprocessor so memory is allocated. The preprocessor replace each occurrence of name with value in program before execution.

OPERATOR

An operator is a symbol which performs operations on given data elements.

Table 2 *Precedence and Associativity*

( ) Parenthesis [ ] Index → Member of • Member of	$L-R$
Pre ++, – – (unary) – , &(address of) * (Indirection)	$R-L$
Arithmetic * , /, %	$L-R$
Arithmetic: +, –	$L-R$
Bitwise shift : ≪, ≫	$L-R$
Relational: <, >, =. >, > = = =, ! =	$L-R$
Bitwise ex –OR : ∧	$L-R$
Logical AND : &&	$L-R$
Logical OR : ||	$L-R$
Conditional: ? :	$R-L$
Assignment & compound Assignment =, + =, – =, * =, / = ; % =	$R-L$
Separation operator: , (comma)	$L-R$

Note: For Assignment operator, only a variable is allowed on its left.

Precedence Decreases as We Move from Top to Bottom

Examples:

1. int a,b,c;
   $a = b = c = 0$;
   Assigns '0' to a,b,c;

2. int a, $b = 55$, $c = 10$;
   initializes 'b' with 55 and 'c' with '10'.
   $b + c = a$; // Invalid
   Only variable is allowed on left side of assignment.

3. int $a = 15$, $b = 20$, $c = 2$, $d = 5$, $e = 10$, f, g, h, i;
   $f = a << c$;
   'a' is left shifted for 'c' times and result stored in 'f'
   i.e.,
   $a = 15 = (1\ 1\ 1\ 1)_2$
   ↓ ↓ ↓ ↓
   1 1 1 1 0 (After first shift)
   ↓ ↓ ↓ ↓
   1 1 1 1 0 0 (After second shift)
   One left shift multiplies 15 by 2 = 30
   Again the 2nd left shift multiplies 30 by 2 = 60
   Thus $15 \times 2^2 = 60$, where the power of 2 is the number of times shift is made. Value of 'f' becomes 60.

Note: Left shift multiplies the value by 2. Right shift divides the value by 2.

$g = a$ and b;
a – 0 1 1 1 1
b – 1 0 1 0 0
0 0 1 0 0 = 4
'&' performs bitwise AND. So 'g' value is '4'.

$h = a|b$; a – 0 1 1 1 1
b – 1 0 0 0 0
1 1 1 1 1 = 31
"I" performs bitwise 'OR'. R value is '31'.

$i = a \hat{} d$: a – 1 1 1 1
b – 0 1 0 1
1 0 1 0 = 10
'^' performs bit-wise ex – OR. i value is '10'.

4. int $a = 100$, $b = 200$, $c = 300$, x;
   `x = (a>b)?((a>c)?a:c):((b>c)? b:c);`
   `false` `c`
   `x = c`
   `so, x = 300`

5. int $i = 10$, $j = 10$, x, y;
   `x = i+++++i+i+++++i+++i`
   `executes as`
   `++ i`
   `++ i` } `pre-increments`
   `++ i`
   `X = i + i + i + i + i`
   `i ++ ;` } `post-increments`
   `i ++ ;`
   `so x = 65, i = 15.`
   `y = j - - + - - j + j - - + - - j +`
   `- - j`

```
executes as
- - j;
- - j;  pre decrements
- - j;
y = j + j + j + j + j ;
j - -;
j - -;  post decrements.
y = 35; j = 5
```

6. int $i = 10$;

```
printf("%d%d%d%d%d", i++, ++i, ++i,
i++, ++i);
evaluates the values in printf from
right to left.
So
i++, ++i, ++i, i++, ++i
<-------------------
Prints 14 14 13 11 11
Printf ("%d", i)
Prints 15:
```

Type Conversion

'*C*' allows mixed mode operations, i.e., variables of different type may appear in same expression. To perform the operation, the data need to convert into compatible type.

The conversion takes place in two ways:

Implicit

C automatically converts any intermediate values to proper type so that the expression can be evaluated without losing any significance.

For mixed mode operations, generally the 'lower' type is automatically converted to 'higher' type before the operation proceeds.

Explicit

'*C*' allows programmer to use type conversion operator to convert a data value to the required type.

Syntax:
$V1$ = (type) $V2$;
Type in parenthesis represents the destination type.

Example: int $a = 3$, $b = 2$, float x, y;

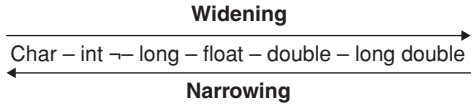
Case I: $x = a/b$;
results $x = 1.000000$

Case II: y = (float) a/b;
results $y = 1.500000$.

Because, in case 1, the integer division is performed and so returns an integer by division operator. While assigning the integer value implicitly converted to 1.000000, then assigns to float variable x where as in case 2, (float)a converts value of 'a' to float, the second variable 'b' is integer. The compiler implicitly converts integer to float. Then it performs float division. So 1.500000 is stored into floating variables.

Notes: '*C*' allows both implicit and explicit type conversion. Type conversion is of two types:

1. Narrowing**:** Conversion of 'higher' type to 'lower' type.
2. Widening**:** Conversion of 'lower' type to 'higher' type.



Note: Narrowing causes loss of data.

Input/output Functions

Function	Purpose
printf	prints formatted string
scanf	reads formatted string
getchar	reads character
putchar	displays a character
gets	reads a string
puts	displays a string

Format Specifier	Purpose
%c	single character
%d	decimal integer
%e	floating point
%f	floating point
%h	short int
%o	octal integer
%x	hexa decimal
%s	string
%u	unsigned decimal integer

Note: scanf("%s", string_var); does not read string which contains white space. Hence to read multi word string use gets(string_var);

Example 1: Which of following comment regarding the reading of a string using scanf( ) and gets ( ) is true?
(A) Both can be used interchangeably
(B) scanf is delimited by end of line, gets is delimited by blank space
(C) scanf is delimited by blank, gets is delimited by end of line
(D) None of these
Ans: (C)

Program Structure

```
/* Documentation section */
Preprocessor commands;
Global declaration;
main ()
```

```
{
Body of main;
}
User defined function area;
```

Documentation section/comments Ignored by compiler, provides additional information to user to improve readability.

Preprocessing Tells the compiler to do pre-processing before doing compilation. For example

#include < stdio.h > tells to include stdio header file.

Global declaration It contains variable declarations, these are accessible in more than one function.

Function Functions are main building blocks of 'C' program. Every 'C' program contains one or more functions. A mandatory function called 'main( )' instructs the compiler to start execution from here.

User defined area Here user can define his own functions.

Control Statements

The statement that controls the execution sequence of a program is called "control statement".

The control statements are classified as:

1. Selection statement: if, switch
2. Iterative/looping statement: While, do-while, for
3. Unconditional jump statements: break, continue, return, goto

Selection/Decision-making Statement

Makes a decision to select and execute statement(*s*) based on the condition. '*C*' supports if and switch selection statements.

The if statement "if" is called two-way selection statement.

Syntax:

```
if (expression) // simple-if
       statement(s);
if (expression) // if-else
{
       statement1(s);
}
else
{
       statements(s);
}
if (expression) // ladder else-if.
{
       Statement1(s);
}
else if (expression2)
{
Statement2(s);
}
else
{
Statement3(s);
}
```

Nested if:

```
       if (expression1)
{
       Statement(s)1;
       if (expression(s)2)
 else
       Statement(s)3;
}
else
Statement(s)4;
```

Note: If the expression evaluates to true then the statements of if block gets executed otherwise else block statements will execute.

Example 2: Consider the following program segment:

```
if (a > b) printf ("a > b");
else
printf ("else part");
printf ("a < = b");
a < = b will be printed if
```

(A) $a > b$ (B) $a < b$
(C) $a = b$ (D) all of these

Ans: (D)

Because the statement, printf("a < = b"); is not the part of either if block or else block.

The switch statement Switch is a multi-way (n-way) selection statement.

Syntax:

```
switch (var_name/exp)
{
case const1: stmts1;
             break;
case const2: stmts2;
             break;
       .
       .
       .
case constn: stmts n;
             break;
default: statements;
}
```

Notes:

- For switch only the integral (integer/char) type variables or expression evaluates to integral allowed.
- Absence of break after case statements leads continuation execution of all case statements followed by matching case block.

Example 3:

```
main ()
{
int i = 10;
switch(i)
{
case 10 : printf ("case 10");
case 15 : printf ("case 15");
case 20 : printf ("case 20");
default : printf ("default case");
}
}
```

Output: Case 10 case 15 case 20 default case
Reason: Missing break after each case, leads to execution of all the cases from matching case.

Example 4:

```
main ( )
{
int i = 10;
switch (i)
{
case 10 : printf("case 10");
break ;
case 8 + 2 : printf("case 8+2");
             break;
default : printf(" No matching case");
}
}
```

Program raises an error called 'Duplicate case' while compiling because the expression '8 + 2' evaluates to '10'.

Looping Statements

Sometimes, there is a situation to execute statement(s) repeatedly for a number of times or until the condition satisfies. *C*' supports following looping statements: while, do-while, for.

While Statement

Syntax: while (condition)

```
{
Statement(s);
}
```

If the condition is true the block of statements will execute and control returns to condition, i.e., the statement(*s*) executes till the condition becomes false.

Notes:
- 'While' executes the block either '0' or more times.
- 'While' is called entry control loop.

Do-while Statement.

Syntax:

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

do-while is same as 'while; except that the statement(s) will execute for at least once.

Notes:
- The condition will not be evaluate to execute the block for first time.
- 'do-while' is called exit-control loop.

Example 5:

```
main ( )
{
int i = 0;
while (i! = 0)
{
        printf("%d", i);
        i++;
}
}
```

No output, because the condition is false for the first time.

```
main ( )
{
int  i = 0;
do
{
printf("%d", i);
i++;
} while (i! = 0);
}
```

Output: Displays 0 to 32767 and–32768 to–1

The for loop 'for' provides more concise loop control structure.

Syntax:

```
for(exp1; exp2; exp3)
{
Statement(s);
}
```

Expression 1: Initialization expression may contain multiple initializations. It executes only once before executing the loop for first time.

Expression 2: Condition expression. Only one condition expression is allowed. That may be single or compound condition, evaluates before every execution.

Expression 3: Modification statement may contain multiple statements. It executes on completion of loop body for every iteration.

Note: All the expressions in parenthesis are optional. Two semi-colons (;) are compulsory even though there are no expressions.

Odd loops In the for loop, while loop, the condition specifies the number of times a loop can be executed. Sometimes a user may not know, about the number of times a loop is to be executed. If we want to execute a loop for unknown number of times, then the concept of odd loops should be implemented, these can be done using the for, while (or) do-while loops. Let us illustrate odd–loop with a program

```
# include <stdio.h>
main()
{
int num, x;
num = 1;
while (num = = 1)
{
printf ("enter a number");
scanf ("%d", & x);
if((x % 2) = = 0)
printf("number is even");
else
printf("number is odd");
printf("do u want to test any num.");
printf("for yes-enter '1', No-enter '0'");
Scanf("%d",& num);
}
}
```

Unconditional Jump Statements

- "*C*" language permits to jump from one statement to another.
- '*C*' supports break, continue, return and goto jump statements.

Break statement Breaks the execution sequence. That is when the break statement executes in a block (loop) it'll come out from block (loop).

Syntax:

```
break;
```

Continue statement Used to skip a part of the loop under certain conditions.

Syntax:

```
continue;
```

Return statement Terminates the execution of a function and returns the control to the calling function.

Syntax:

```
return [exp/value];
```

Goto statement Jumps from one point to another with in a function.

Syntax:

```
label1:                 goto label2:
Statement(s);           Statement(s);
goto label1;            label2;
reverse jump            forward jump
```

Reverse jump, executes the statements repeatedly where as in forward jump, the statements are skipped from execution.

Example 6:

```
main( )
{
int i ;
for (i=1; i<=10; i++)
{
        if (i = = 5)
                break;
        printf("%d" , i);
}
}
```

output: 1 2 3 4

if $i = 5$, then the loop will break.

Example 7:

```
main( )
{
        int i ;
        for (i  = 1; i<=10; i++)
        {
                if (i = = 5)
                continue;
                printf("%d" , i);
        }
}
```

o/p: 1 2 3 4 6 7 8 9 10

if $i = 5$, the loop statements skipped for that iteration. So it does not print '5'.

Example 8:

Output for the following program segment

```
for (i = 1, j = 10 ; i < 6; ++i, --j)
printf("\n %d %d", i, j);
```

Output:

1	10
2	9
3	8
4	7
5	6

Note: Since for statement allows multiple initialization and multiple update statements, expression 1 and expression 3, does not raise any error.

EXERCISES

Practice Problems I

Directions for questions 1 to 15: Select the correct alternative from the given choices.

1. What will be the output of the following program?

```
void main()
{
int i;
char a[ ] =" \0 ";
if (printf("%s\n", a))
printf ("ok \n");
else
printf("program error \n");
}
```

(A) ok (B) progam error
(C) no output (D) compilation error

2. Output of the following will be

```
# define FALSE-1
# define TRUE 1
# define NULL 0
main( )
{
    if(NULL)
    puts("NULL");
    else if(FALSE)
    puts("TRUE");
    else
    puts("FALSE");
    }
```

(A) NULL (B) TRUE
(C) FALSE (D) 1

3. main( )

```
{
    printf("%x",-1 << 4) ;
}
```

For the above program output will be
(A) FFF0 (B) FF00
(C) 00FF (D) 0FFF

4. For the following program

define sqr (*a*) *a***a*

```
main( )
    {
    int i;
    i = 64 / sqr(4);
    printf( "%d", i);
    }
```

output will be
(A) 4 (B) 16
(C) 64 (D) compilation error

5. #define clrscr ( ) 1000

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

Output of the above program will be?
(A) error (B) No output
(C) 1000 (D) 1

6. Output of the following program is

```
main( )
{
    int i = -2;
    +i;
    printf("i = %d, +i = %d\n", i, +i);
```

(A) error (B) –2, +2
(C) –2, –2 (D) –2, 2

7. main( )

```
{
  int n;
  printf("%d", scanf ("%d", & n));
}
```

For the above program if input is given as 20. What will be the output?
(A) 20 (B) 1
(C) 2 (D) 0

8. How many times will the following code be executed?

```
{
    x = 10;
    while (x = 1)
           x ++;
}
```

(A) Never
(B) Once
(C) 15 times
(D) Infinite number of times

9. The following statement

```
printf("%d", 9%5); prints
```

(A) 1.8 (B) 1.0
(C) 4 (D) 2

10. int *a*;

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

What is the output of the above code fragment?
(A) 0 (B) 2
(C) Garbage value (D) 3

11. printf("%d", printf("time"));
(A) syntax error
(B) outputs time 4
(C) outputs garbage
(D) prints time and terminates abruptly

12. The following program

```
main( )
{
    int i = 2;
    {
    int i = 4, j = 5;
```

```
    printf ("%d%d",i,j);
    }
    printf ("%d%d",i,j);
}
```

(A) Compiler error: unrecognised symbol *j*;
(B) Prints 2545
(C) Print 4525
(D) None of the above

13. What is the output of the following program fragment?

```
for (i = 3; i < 15; i + = 3);
printf ("%d", i);
```

(A) a syntax error (B) an execution error
(C) prints 12 (D) prints 15

14. What is the output of the following program segment?

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

(A) Outputs an error message
(B) Prints 0
(C) Prints 1
(D) None of these

15. The statements:

```
a = 7;
printf("%d", (a++));
prints
```

(A) Value of 8 (B) Value of 7
(C) Value of 0 (D) None of the above

Practice Problems 2

Directions for questions 1 to 12: Select the correct alternative from the given choices.

1. If the condition is missing in a FOR loop of a C program then
(A) It is assumed to be present and taken to be false
(B) It is assumed to be present and taken to be true
(C) It results in syntax error
(D) Execution will be terminated abruptly

2. Which of the following operators in '*C*' does not associate from the right?
(A) = (B) + =
(C) postfix++ (D) >

3. In a C programming language $x - = y + 1$ means
(A) $x = -x - y - 1$ (B) $x = x - y + 1$
(C) $x = x - y - 1$ (D) $x = -x + y + 1$

4. Minimum number of temporary variables needed to swap two variables is
(A) 1 (B) 2
(C) 3 (D) 0

5. A preprocessor command
(A) need not start on a new line
(B) need not start on the first column
(C) has # as the first character
(D) comes after the first executable statement

6. printf ("%d", printf ("%d", printf("time4kids")));
(A) Outputs time (B) Syntax error
(C) Outputs 9 (D) None of the above

7. for ($i = 1$; $i < 5$; i++)
if (i!=3)
printf("%d", i);
Outputs:
(A) 12345 (B) Error
(C) 1245 (D) 0000

8. Which operand in '*C*' takes only integer operands?
(A) * (B) /
(C) % (D) +

9. An unrestricted use of 'goto' statement is harmful because
(A) it results in increasing the executing time of the program
(B) it increases the memory of the program
(C) it decreases the readability and testing of program
(D) None of the above

10. What will be the output?

```
main()
{
int i = 0, j = 0;
if(i && j ++)
printf("%d..%d", i++, j);
printf("%d..%d", i, j);
}
```

(A) 1..1 (B) 2..2
(C) 0..0 (D) 1..1, 1..1

11. What is the output?

```
main ()
{
int a = 0;
int b = 20;
char x = 1;
char y = 10;
if(a, b, x, y);
printf("hello");
}
```

(A) logical error (B) Garbage value
(C) hello (D) 20

12. What will be the value of count after executing the below program:

```
main ( ) {
int count = 10, digit = 0;
while (digit < = 9) {
printf ("%d\n", ++count);
++digit;
}
}
```

(A) 10 (B) 11
(C) 20 (D) 21

Previous Years' Questions

1. Which one of the following are essential features of an object-oriented programming language?
(i) Abstraction and encapsulation
(ii) Strictly-typedness
(iii) Type-safe property coupled with sub-type rule
(iv) Polymorphism in the presence of inheritance

[2005]

(A) (i) and (ii) only
(B) (i) and (iv) only
(C) (i), (ii) and (iv) only
(D) (i), (iii) and (iv) only

2. Which of the following are true?

(i) A programming language which does not permit global variables of any kind and has no nesting of procedures/functions, but permits recursion can be implemented with static storage allocation

(ii) Multi-level access link (or display) arrangement is needed to arrange activation records only if the programming language being implemented has nesting of procedures/functions

(iii) Recursion in programming languages cannot be implemented with dynamic storage allocation

(iv) Nesting procedures/functions and recursion require a dynamic heap allocation scheme and cannot be implemented with a stack-based allocation scheme for activation records

(v) Programming languages which permit a function to return a function as its result cannot be implemented with a stack-based storage allocation scheme for activation records

[2008]

(A) (ii) and (v) only (B) (i), (iii) and (iv) only
(C) (i), (ii) and (v) only (D) (ii), (iii) and (v) only

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

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

[2012]

(A) No choice
(B) Choice A
(C) Choice A
Choice B No choice
(D) Program gives no output as it is erroneous

4. 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]

(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) $p = n * (n - 1) * (n - 2)/6.0;$

5. The secant method is used to find the root of an equation $f(x) = 0$. It is started from two distinct estimates x_a and x_b for the root. It is an iterative procedure involving linear interpolation to a root. The iteration stops if $f(x_b)$ is very small and then x_b is the solution. The procedure is given below. Observe that there is an expression which is missing and is marked by ?. Which is the suitable expression that is to put in place of ? so that it follows all steps of the secant method?

[2015]

Secant

Initialize: x_a, x_b, ε, N // ε = convergence indicator // N = maximum no. of iterations
$f_b = f(x_b)$
$i = 0$
while ($i < N$ and $|f_b| > \varepsilon$) do
$i = i + 1$ // update counter
x_t = ? // missing expression for // intermediate value
$x_a = x_b$ // reset x_a
$x_b = x_t$ // reset x_b
$f_b = f(x_b)$ // function value at new x_b
end while
if $|f_b| > \varepsilon$ then // loop is terminated with $i = N$
write "Non-convergence"
else
write "return x_b"
end if

(A) $x_b - (f_b - f(x_a))\, f_b/(x_b - x_a)$
(B) $x_a - (f_a - f(x_a))\, f_a/(x_b - x_a)$
(C) $x_b - (x_b - x_a) f_b/(f_b - f(x_a))$
(D) $x_a - (x_b - x_a) f_a/(f_b - f(x_a))$

6. Consider the following C program:

```
#include<stdio.h>
int main( )
{
int i, j, k = 0;
j = 2 * 3 / 4 + 2.0 / 5 + 8 / 5;
k -= --j;
for (i = 0; i < 5; i ++)
{
switch(i + k)
```

```
{
case 1:
case 2: printf("\n%d", i + k);
case 3: printf("\n%d", i + k);
default: printf(("\n%d", i + k);
}
}
return 0;
}
```

The number of times printf statement is executed is _____. **[2015]**

7. Consider the C program fragment below which is meant to divide x by y using repeated subtractions. The variables x, y, q and r are all unsigned int.

```
while (r >= y) {
    r = r – y;
    q = q + 1;
}
```

Which of the following conditions on the variables x, y, q and r before the execution of the fragment will ensure that the loop terminates in a state satisfying the condition x == (y*q + r)? **[2017]**

(A) (q == r) && (r == 0)
(B) (x > 0) && (r == x) &&(y > 0)
(C) (q == 0) && (r == x) && (y > 0)
(D) (q == 0) && (y > 0)

8. Consider the following C Program.

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

The output of the program is _____. **[2017]**

Answer Keys

Exercises

Practice Problems 1

1. A	**2.** B	**3.** A	**4.** C	**5.** C	**6.** C	**7.** B	**8.** D	**9.** C	**10.** C
11. B	**12.** A	**13.** D	**14.** D	**15.** B					

Practice Problems 2

1. B	**2.** D	**3.** C	**4.** D	**5.** C	**6.** D	**7.** C	**8.** C	**9.** C	**10.** C
11. C	**12.** C								

Previous Years' Questions

1. B	**2.** D	**3.** C	**4.** B	**5.** C	**6.** 10	**7.** C	**8.** 0

Chapter 2

Functions

LEARNING OBJECTIVES

- *Functions*
- *Library functions*
- *User defined functions*
- *Defining user defined functions*
- *Recursion*
- *Parameter passing*
- *Pass by value*
- *Pass by address*
- *Scope*
- *Life time*
- *Binding*

Functions

A function is a block of code that performs a specific task. It has a name and is reusable, i.e., it can be executed from as many different parts in a program as required.

Functions make possible top down modular programming. In this style of programming, the high-level logic of overall problem is solved first, whereas the detail of each lower-level function is addressed later. This approach reduces the complexity in writing program.

1. Every C program can be thought of collection of functions.
2. main( ) is also a function.

Types of Functions

Library functions

These are the in-built functions of '*C*' library. These are already defined in header files.

Example 1: printf( ); is a function which is used to print at output. It is defined in '`stdio.h`' file.

User-defined functions

Programmers can create their own function in '*C*' to perform specific tasks.

Example 2:
```
# include <stdio.h>
main( )
{
message( );
}
message( )
{
printf("Hello");
}
```

- A function receives zero (or) more parameters, performs a specific task, and returns zero or one value.
- A function is invoked by its name and parameters.
- No two functions have the same name in a single C program.
- The communication between the function and invoker is through the parameter and the return value.
- A function is independent.
- It is "completely" self-contained.
- It can be called at any place of your code and can be ported to another program.
- Functions make programs reusable and readable.

Example 3: Return the largest of two integers.
```
int maximum (int a, int b)
{
if (a > b)
return a;
else
return b;
}
```

Note: Function calls execute with the help of execution stack. Execution of 'C program' starts with main( ) function. Main( ) is a user-defined function.

Defining User-defined Functions

In order to work with user-defined functions, it requires the following concepts about functions:

- Declaration of a function
- Definition of a function
- Function call

Declaration specifies what

- is the name of the function
- are the parameters to pass (type, order and number of parameters).
- it returns on completion of execution

Example 4: int maximum (int, int); int maximum (int *a*, int *b*);

Syntax:

```
Return_type Function_Name(Parameter_list);
```

- Names of parameters are optional in declaration.
- Default return type for 'C' functions is 'int'.
- A function, whose return type is void returns nothing.
- Empty parenthesis after a function name in declaration says, function does not accept any parameters.

Definition specifies *how*

- to perform the specified task
- to accept passed parameters
- to process parameters or execute instruction to producer equired results (return value).

Function definition is a self-contained block of instructions, will be executed on call:

Syntax:

```
Return _type Function -Name(paralist)
{
Local declaration(s);
Executable statements(s);
}
int maximum (int a, int b)
{
if (a > b)
  return a;
else
  return b;
}
```

Function call specifies

1. where to execute the function
2. when to execute the function

Note: If the function definition provided before use (call), the declaration is optional.

The following example describes control flow during function call:

```
void hello(); // Declaration
void main()
{
printf("\n function");
hello();
printf("\n Main after call to hello")
void hello()//Definition
 {
printf ("\n function Hello");
return;
 }
```

A *return* statement has two important uses:

1. first, it causes an immediate exit from the function.
2. second, it may be used to return a value.

If a function does not return any value, then the return statement is optional.

Recursion

In general, programmers use two approaches to write repetitive algorithms. One approach using *loops*, the other is *recursion*.

Recursive functions typically implement recurrence relations, which are mathematical formula in which the desired expression (function) involving a positive integer, *n*, is described in terms of the function applied to corresponding values for integers less than '*n*'.

1. The function written in terms of itself is a recursive case.
2. The recursive case must call the function with a decreasing '*n*'.
3. Recursion in computer programming is exemplified when a function defined in terms of itself.
4. Recursion is a repetitive process in which a function calls itself.

Note: Recursive function must have an *if* condition to force the function to return without recursive call being executed. If there is no such condition, then the function execution falls into infinite loop.

> Rules for designing recursive function
> 1. Determine base case
> 2. Determine general case
> 3. Combine, base and general case into a function

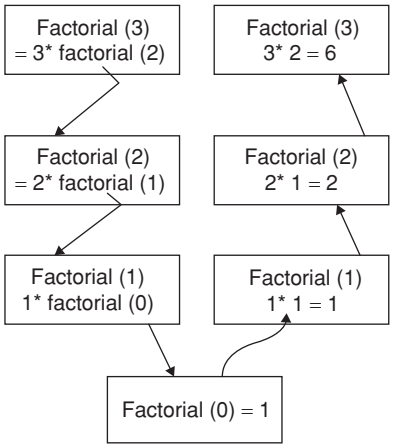
Example 5: Recursive factorial function

```
1. int factorial (int n)
2. {
3. if (n = = 0)
4. return 1;
5. else
6. return (n * factorial (n - 1));
7. }
```

The statement 3 is a base condition, which stops the recursive call of function.

The statement 6 reduces the size of problem by recursively calling the factorial with $(n - 1)$.

Execution sequences for factorial (3):



Disadvantages:

1. Recursive programs increase the execution time of program.
2. Recursive programs typically use a large amount of computer memory and the greater the recursion, the more memory is used.
3. Recursive programs can be confusing to develop and extremely complicated to debug.

Parameter Passing

There are two ways of passing parameters to functions in 'C' language.

1. Pass-by-value: When parameters are passed by value, create copies in called function. This mechanism is used when we do not want to change the value of actual parameters.
2. Pass-by-address: In this case, only the addresses of parameters are passed to the called function. Therefore, manipulation of formal parameters affects actual parameters.

Examples 6:

```
void swap1(int, int); /* function – to swap
two numbers by passing values */
void swap2 (int *, int *); /* function to
swap two numbers by passing Address * /
   void main ()
{
int a = 10, b = 15, c = 5, d = 25;
printf("value of a and b before swapping
:%d, %d" a , b );
swap1(a, b);
printf("values of a and b after swapping :
%d, %d", a, b);
printf ("values of c and d before swapping
:%d%d", c,d );
Swap2(&c, &d);
printf("values of c and d after swapping
%d, %d", c, d);
}
void swap1(int x, int y )
{
int temp;
temp = x;
x = y;
y = temp;
}
void swap2 (int *x, int *y)
{
int temp;
temp = *x;
*x = *y:
*y = temp;
}
```

Output:

Value of *a* and *b* before swapping: 10, 15
Value of *a* and *b* after swapping: 10, 15
Value of *c* and *d* before swapping: 5, 25
Value of *c* and *d* after swapping: 25, 5

Solved Examples

Example 1: Consider the program below:

```
#include<stdio.h>
int fun (int n, int *fp)
```

Table 1 *Comparison of pass-by-value and pass-by-address*

	Pass-by-value		Pass-by-address
1.	Also known as call-by-value	1.	Also known as call-by-address or call by-reference
2.	Pass the values of actual parameters	2.	Pass the address of actual parameters
3.	Formal parameters act as duplicates or as a copy to actual parameters	3.	Formal parameters acts as references to the actual parameters
4.	Operations on formal parameter does not affect actual parameters	4.	Operations on formal parameters affect actual parameters
5.	Passing of parameters is time consuming as the data size increases	5.	The size of parameters does not affect the time for transferring references.
6.	Actual parameters are secured	6.	Helps to return multiple parameters

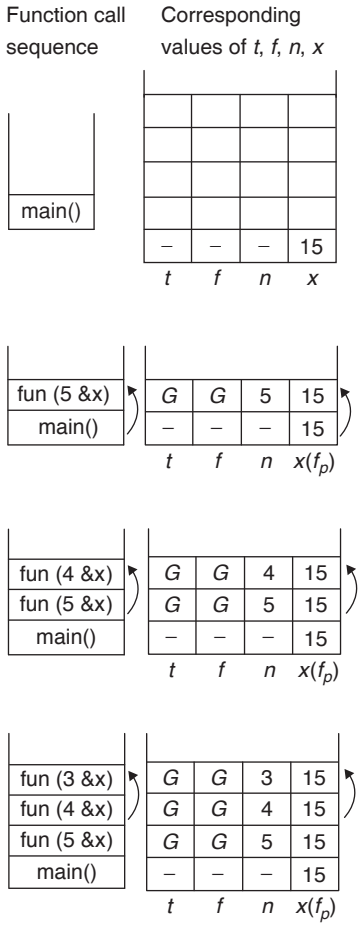
```
{
int t,f;
if (n < =1)
{
*fp=1;
return 1;
 }
t = fun(n-1, fp);
f = t+ *fp;
*fp = t;
return f;
}
int main ()
{
int x = 15;
printf ("%d\n", fun(5,&x));
return 0;
}
```

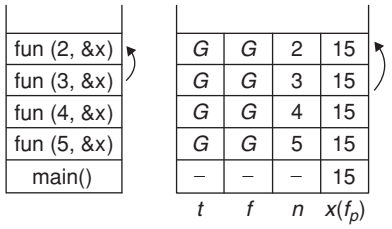
What is the output?

(A) 2 (B) 4
(C) 8 (D) 16

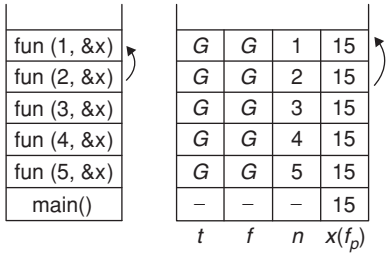
Solution: (C)

Execution stack

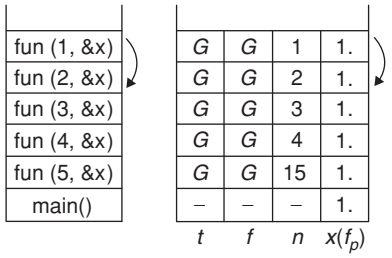


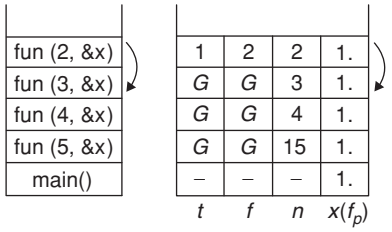


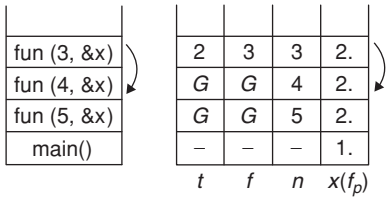
Note: '–' indicates *no memory* allocated to variable. '*G*' indicates *garbage value*.

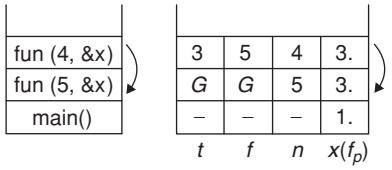


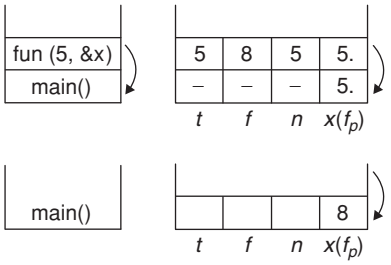
For the function call fun(1, &x) condition (*n*<=1) is true. So Assigns '1' to fp and returns '1'.











Finally, *x* contains '8', so printf prints '8'.

Example 2: What does the following program prints?

```
#include < stdio.h>
void f (int *p, int *q)
{
p=q;
*p=12;
}
int i = 0, j=1;
int main()
{
f(&i, &j);
printf(" %d%d ", i, j);
return 0 ;
}
```

(A) 2 12 (B) 12 1
(C) 0 1 (D) 0 12

Solution: (D)

```
main( )
f (&i, &j)
address of 'i' is stored in to p.
and address of 'j' is stored into 'q'.
i.e., *p and*q refers i and j.
The statement:
p = q; updates pointer 'p', so that both
pointers refer to parameter 'j'.
*p = 12
Changes value of 'j' to '12' But 'i' does
not effected. So, prints 0 12.
```

Example 3: What is the value printed by the following program?

```
# 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 G [ ] = { 12, 7, 13, 4, 11, 6};
printf("%d", f (a,b));
return 0;
}
```

(a) –9 (b) 12
(c) 15 (d) 20

Solution: (C)

	0	1	2	3	4	5
a	12	7	13	4	11	6

$f(a, 6)$ is the first call to function $f( )$.

The array _ name refers to base address of array, i.e., address of first element.

Thus,

$F(a, 6)$

12 % 2 = 0. So,

$$12 + f\underset{\downarrow 7}{(\overline{a+1}),\left(\frac{n-1}{5}\right)}\left[*a \text{ is even}\right]$$

$$12+\left(7-\left(\underset{\downarrow 13}{f(a+1),\left(\frac{n-1}{4}\right)}\right)\right)[*a \text{is odd}]$$

$$12+\left(7-\left(13-f\left(\underset{\downarrow 4}{(a+1)},\left(\frac{n-1}{3}\right)\right)\right)\right)[*a \text{ is odd}]$$

$$12+\left(7-\left(13-\left(4+f\left(\underset{\downarrow 11}{(a+1),\frac{(n-1)}{2}},\right)\right)\right)\right)[*a \text{ is even}]$$

$$12+\left(7-\left(13-\left(4+\left(11-f\left(\underset{\downarrow 6}{(a+1),\left(\frac{n-1}{1}\right)}\right)\right)\right)\right)\right)[*a \text{ is odd}]$$

$$12+\left(7-\left(13-\left(4+\left(11-\left(6+\underset{\downarrow 0}{\frac{f(a+1),(n-1)}{0}}\right)\right)\right)\right)\right)[*a \text{ is even}]$$

$$12 + (7 \quad (13 - (4 + (11 - (6 + 0))))) = 15$$

Scope, Lifetime and Binding

Storage classes specify the scope, lifetime and binding of variables. To fully define a variable, one needs to mention not only its 'type' but also its 'storage class'.

A variable name identifies some physical location within computer memory where a collection of bits are allocated for storing value of variable.

Storage class tells us:

1. Where the variable would be stored (either in memory or CPU registers)?
2. What will be the initial value of a variable, if no value is specifically initialized?
3. What is the scope of a variable (where it can be accessed)?
4. What is the life of a variable?

Scope

The scope defines the visibility of an object. It defines where an object can be referenced/accessed; generally, the scope of variable is local or global.

1. The variables defined within a block have *local scope*. They are *visible only to the block* in which they are defined.
2. The variables defined in global area are visible from their definition until the end of program. It is *visible everywhere* in program.

Lifetime

The lifetime of a variable defines the duration for which the computer allocates memory for it (the duration between allocation and deallocation of memory).

In *C*, variable can have automatic, static or dynamic lifetime.

1. Automatic: Variables with automatic lifetime are created each time their declaration are encountered and are destroyed each time their blocks are exited.
2. Static: A variable is created when the declaration is executed for the first time and destroyed when the execution stops/terminates.
3. Dynamic: The variable's memory is allocated and deallocated through memory management functions.

Binding

Binding finds the corresponding binding occurrence (declaration/definition) for an applied occurrence (usage) of an identifier. For Binding.

1. Scope of variables should be known. What is the block structure? In which block the identifier is variable?
2. What will happen if we use same identifier name again? '*C* forbids use of same identifier name in the same scope'. Same name can be used in different scopes.

Examples:

1.
```
double f,y;
int f(  ) // error
{
.
.
.
}
double y; // error
```

2.
```
double y;
int f( )
{
double f;// legal
int y; //legal
}
```

There are four storage classes in *C*.

Storage class	Storage Area	Default Initial Value	Lifetime	Scope	Keyword
Automatic	Memory	Till the control remains in block	Till the control remains in block	Local	auto
Register	CPU register	An unpredictable value (or) garbage value	Till the control remains in block	Local	register
Static	Memory	Zero	Value of variable persist between function calls	Local	static
External	Memory	Unpredictable or garbage value	Throughout program execution	Global	extern

Note: Default storage class is auto.

Example 4: What will be the output for the program?

```
int i = 33;
main( )
{
   extern int i;
     {
         int i = 22;
          {
               const volatile unsigned i
               = 11;
                    printf (" %d ", i);
              }
                    printf (" %d ", i);
              }
printf ("%d ", i) ;
}
```

(A) error
(B) 11 22 33
(C) 11 22 garbage
(D) 11 11 11

Solution: (B)

'{' introduces new block and thus new scope. In the innermost block, *i* is declared as const volatile unsigned which is a valid declaration. *i* is assumed of type int. So printf prints 11. In the next block, *i* has value 22 and so printf prints 22. In the outermost block, *i* is declared as extern, so no storage space is allocated for it. After compilation is over, the linker resolves it to global variable, *i* since it is the only variable visible there. So it prints its value as 33.

Example 5: Consider the following C program:

```
int f(int n)
{
static int r;
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) 15 (b) 17
(c) 18 (d) 19

Solution: (C)

Call Sequence	*r*	Return Sequence
f(5) ↓	5	18
f(3)+2	5	16+2 ↓
f(2)+*r*	5	11+5 ↓
f(1)+*r*	5	6+5 ↓
f(0)+*r*	5	1+5 ↓

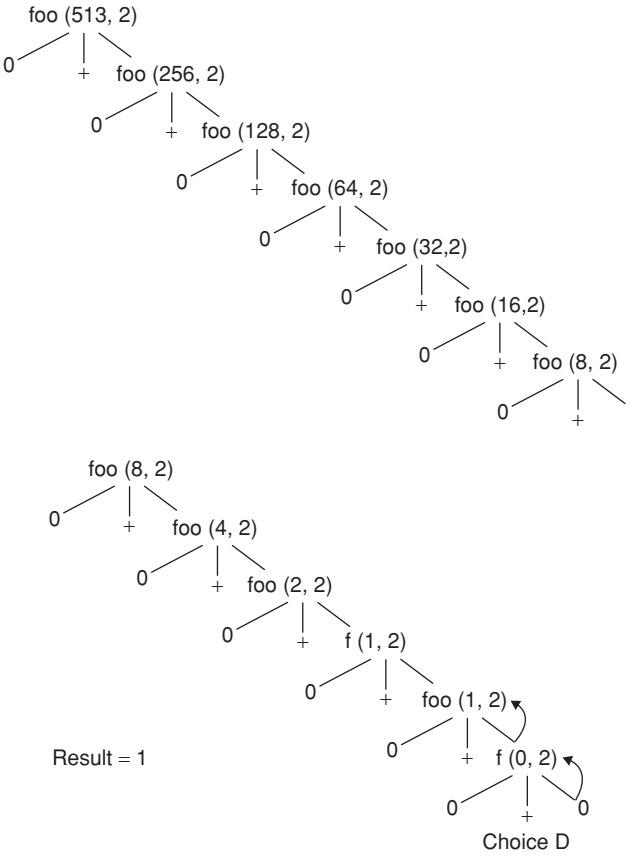
Common data for questions 6 and 7: Consider the following recursive 'C' function that takes two arguments.
unsigned int foo (unsigned int *n*, unsigned int *r*)

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

Example 6: What is the return value of the function foo when it is called as foo (512,2)?

(A) 9 (B) 8
(C) 2 (D) 1

Solution: (D)

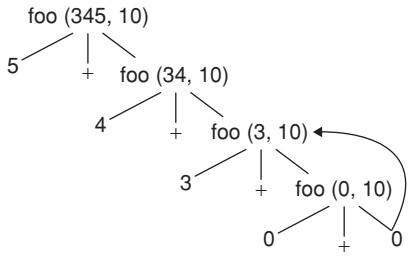


Result = 1

Example 7: What is return value for the function call foo (345, 10)?

(A) 345 (B) 12
(C) 5 (D) 3

Solution: (B)



result 5 + 4 + 3 = 12

EXERCISES

Practice Problems I

Directions for questions 1 to 15: Select the correct alternative from the given choices.

1. What will be the output of the following program?

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

(A) overflow error (B) syntax error
(C) returns 0 (D) returns 1

2. Output of the following program is

```
 main( )
  {
static int var = 6;
printf("%d\t", var--);
if(var)
                    main( );
  }
```

(A) 5 4 3 2 1 0 (B) 6 6 6 6 6 6
(C) 6 5 4 3 2 1 (D) Error

3. Which of the following will be the output of the program?

```
main( )
{
 char str[ ] = "Hello";
 display( str );
}
void display (char *str)
    {
        printf ( "%s", str) ;
    }
```

(A) compilation error (B) hello
(C) print null string (D) no output

4. Consider the following C function

```
int fun (int n)
{
  static int x = 0;
  if (n<=0) return 1;
  if (n>3)
     {
          x = n;
       return fun(n-2)+3;
     }
 return fun(n-1)+ x;
 }
```

What is the value of fun(5)?

(A) 4 (B) 15
(C) 18 (D) 19

5. For the following C function

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

In order to exchange the values of two variables *w* and *z*,

(A) call swap (*w*, *z*)
(B) call swap (and *w*, and *z*)
(C) swap (*w*, *z*) cannot be used as it does not return any value
(D) swap (*w*, *z*) cannot be used as the parameters are passed by value

6. Choose the correct option to fill? *x* and? *y* so that the program below prints an input string in reverse order. Assume that the input string is terminated by a new line character:

```
void Rev(void) {
  int a;
  if (?x) Rev( );
  ?y
      }
main( ) {
printf("Enter the text");
printf(" \n");
Rev( );
printf("\n");
}
```

(A) `? x is (getchar( )! = '\n')`
`? y is getchar (A);`
(B) `? x is((A = getchar( )) ! = '\n')`
`? y is getchar(A) ;`
(C) `? x is (A! = '\n')`
`? y is putchar (A);`
(D) `? x is (A = getchar ( )) ! = '\n')`
`? y is putchar(A) ;`

7.

```
main ( )
  {
     extern int a;
      a = 30;
       printf ("%d", a);
  }
```

What will be the output of the above program?

(A) 30 (B) Compiler error
(C) Runtime error (D) Linker error

8. Which of the following will be the output of the program?

```
void main ( )
 {
    int n = ret(sizeof(float));
    printf("\n value is %d ", ++n);
}
int ret(int ret)
{
```

```
    ret += 2.5;
   return (ret);
}
```

(A) Value is 6 (B) Value is 6.5
(C) Value is 7 (D) Value is 7.5

9. The following program

```
main( )
{
    pt( ); pt( );pt( );
}
pt( )
{
    static int a;
    printf("%d", ++a) ;
}
```

prints
(A) 0 1 2
(B) 1 2 3
(C) 3 consecutive, but unpredictable numbers
(D) 1 1 1

10. What is the output of the following program?

```
main( ) {
    int i = 0;
    while (i < 4) {
                        sum(i);
                        i++;
                    }
          }
void sum(int i) {
    static int k;
printf ("%d", k + i);
k++;
                    }
```

(A) 0 2 4 6 (B) 0 1 2 3
(C) 0 2 0 0 (D) 1 3 5 7

11. What will be the output of following code?

```
# include <stdio.h>
aaa() {
printf("hi");
}
bbb() {
printf("hello");
}
ccc()
{
printf("bye");
}
main ()
{
int *ptr[3]( );
ptr[0] = aaa;
ptr[1] = bbb;
ptr[2] = ccc;
ptr[2]();
}
```

(A) hi (B) hello
(C) bye (D) Garbage value

12. What is the output?

```
void main()
{
static int i = 5;
if(--i)
{
main();
printf("%d", i);
}
}
```

(A) 5 (B) 5 5 5 5
(C) 0 0 0 0 (D) 1 1 1 1

13. If the following function gets compiled, what error would be raised?

```
double fun(int x, double y)
{
int x;
x = 100;
return y;
}
```

(A) Function should be defined as int fun(int *x*, double *y*)
(B) Missing parenthesis in return
(C) Redeclaration of *x*
(D) All of these

14. Consider the following function:

```
fun(int x)
{
if ((x/2)! = 0)
return (fun (x/2)  10 + x%2);
else return 1;
}
```

What will happen if the function 'fun' called with value 16 i.e., as fun(16).
(A) Infinite loop
(B) Random value will be returned
(C) 11111
(D) 10000

15. What is the output of the following program?

```
void main( )
{
static int x = 5;
printf("%d", x - - );
if (x ! = 0)
main( );
}
```

(A) error:main( ) cannot be called from main( )
(B) Infinite loop
(C) 5 4 3 2 1
(D) 0

Practice Problems 2

Directions for questions 1 to 15: Select the correct alternative from the given choices.

1. An external variable
(A) is globally accessible by all functions
(B) has a declaration "extern" associated with it when declared within a function
(C) will be initialized to 0, if not initialized
(D) all of the above

2. The order in which actual arguments are evaluated in a function call
(A) is from the left (B) is from the right
(C) is unpredictable (D) none of the above

3. In *C* language, it is necessary to declare the type of a function in the calling program if the function
(A) returns an integer (B) Returns a float
(C) both (A) and (B) (D) none of the above

4. What is the output?

```
void main()
{
int k = ret(sizeof(int));
printf("%d", ++k);
}
int ret (int ret)
{
ret + = 2.5;
return (ret);
}
```

(A) 3.5 (B) 5
(C) 4 (D) logical error

5. When a recursive function is called, all its automatic variables are
(A) maintained in stack
(B) retained from last execution
(C) initialized during each call of function
(D) none of these

6. Consider the following program segment:

```
int fun(int x, int y)
{
if(x > 0)
return ((x % y) + fun(x/y, y));
else
return 0;
}
```

What will be the output of the program segment if the function is called as fun(525, 25)?
(A) 25 (B) 12
(C) 21 (D) 42

7. Consider the following C program segment:

```
int fun (int x)
{
static int i = 0;
if (x < = 0)
return 1;
else if (x > 5)
{
i = x;
return fun (x - 3) +2;
}
return fun (x - 2) + i;
}
```

What is the value of fun(7)?
(A) 17 (B) 10
(C) 11 (D) 9

8. Consider the following C program:

```
void rearrange( )
{
char ch;
if (X)
rearrange( );
Y;
}
void main ( )
{
printf("\n enter text to print reverse
order :");
rearrange( ) ;
}
```

Choose the correct option to fill *X* and *Y*, so that the program prints the entered text in reverse order. Assume that input string terminates with new line.

(A)
```
X:   (getchar(ch) = = '\n')
Y:   putchar(ch);
```
(B)
```
X:   (getchar(ch)! = '\n')
Y:   ch = putchar( );
```
(C)
```
X:   ((ch = getchar( ) )! = '\n')
Y:   putchar(ch);
```
(D)
```
X:   ((ch = getchar( )) = = '\n')
Y:   putchar (ch);
```

9. Consider the following C function:

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

The value returned by *f*(1) is
(A) 5 (B) 6
(C) 7 (D) 8

10. Consider the following C function:

```
int incr (int i)
{
static int count = 0;
count = count + i;
return (count);
```

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

The *j* value will be
(A) 10
(B) 4
(C) 6
(D) 7

11. The following function

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

(A) finds the maximum of *a*, *b*, *c*
(B) finds the middle value of *a*, *b*, *c* after sorting
(C) finds the minimum of *a*, *b*, *c*
(D) none of the above

12. Consider the following pseudo code

```
f(a, b)
{
while(b! = 0)
{
t = b;
b = a % b;
a = t;
}
return a;
}
```

(A) The above code computes HCF of two numbers *a* and *b*
(B) The above code computes LCM of *a* and *b*
(C) The above code computes GCD of *a* and *b*
(D) None of the above

13.
```
1. main ( )
2. {int a = 10, *j;
3. void *k;
4. j = k = &a;
5. j++;
6. k++;
7. printf("\n %u, %u", j, k);
8. }
```

Which of the following is true in reference to the above code?
(A) The above code will compile successfully
(B) Error on line number 6
(C) Error on line number 3
(D) Error on line number 4

14. Aliasing in the context of programming language refers to
(A) multiple variables having the same memory location
(B) multiple variables having the same value
(C) multiple variables having the same identifier
(D) multiple uses of the same variable

15. Match the following:

X:	*m* = malloc (5); *m* = NULL;	1:	Using dangling pointers
Y:	free (*n*); *n* ↓ value = 5;	2:	Using un initialized pointers
Z:	char **p*; **p* = '*a*';	3:	Lost memory

(A) X – 1 Y – 3 Z – 2
(B) X – 3 Y – 1 Z – 2
(C) X – 3 Y – 2 Z – 1
(D) X – 2 Y – 1 Z – 3

Previous Years' Questions

1. In the following C function, let $n \geq m$.

```
int gcd(n,m)
{
if (n%m ==0) return m;
   n = n%m;
return gcd(m,n);
}
```

How many recursive calls are made by this function? **[2007]**

(A) $\Theta(\log_2 n)$ (B) $\Omega(n)$

(C) $\Theta(\log_2 \log_2 n)$ (D) $\Theta(\sqrt{n})$

2. What is the time *complexity* of the following recursive function?

```
int DoSomething (int n) {
if (n <= 2)
     return 1;
 else
return(DoSomething(floor(sqrt(n)))+ n);}
```

[2007]

(A) $\Theta(n^2)$ (B) $\Theta(n \log_2 n)$

(C) $\Theta(\log_2 n)$ (D) $\Theta(\log_2 \log_2 n)$

3. 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.

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

[2008]

(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);`

4. Consider the program below:

```
# 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 **[2009]**

(A) 6 (B) 8

(C) 14 (D) 15

5. What is the value printed by the following C program?

```
#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};
printf("%d", f(a,6));
return 0;
}
```

[2010]

(A) –9 (B) 5

(C) 15 (D) 19

Common data for questions 6 and 7: Consider the following recursive C function that takes two arguments.

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

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

(A) 9 (B) 8

(C) 5 (D) 2

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

(A) 345 (B) 12
(C) 5 (D) 3

Common data for questions 8 and 9: Consider the following C code segment

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

8. What output will be generated by the given code segment if: **[2012]**
Line 1 is replaced by **auto int** a **= 1;**
Line 2 is replaced by **register int** a **= 2;**

(A)	(B)	(C)	(D)
3 1	4 2	4 2	4 2
4 1	6 1	6 2	4 2
4 2	6 1	2 0	2 0

9. What output will be generated by the given code segment? **[2012]**

(A)	(B)	(C)	(D)
3 1	4 2	4 2	3 1
4 1	6 1	6 2	5 2
4 2	6 1	2 0	5 2

10. 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.

```
int f(int &x, int c) {
   c = c - 1;
   if (c == 0) return 1;
   x = x + 1;
   return f(x, c) * x;
}
```
[2013]

(A) 3024 (B) 6561
(C) 55440 (D) 161051

11. Consider the following pseudo code. What is the total number of multiplications to be performed? **[2014]**

```
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 three consecutive integers.
(B) One-third of the product of the three consecutive integers.
(C) One-sixth of the product of the three consecutive integers.
(D) None of the above.

12. 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]**

13. 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 two decimals) such that $f(q)$ will return q:_____ **[2014]**

14. Consider the following pseudo code, where x and y are positive integers **[2015]**

```
begin
        q := 0
        r := x
        while r ≥ y do
              begin
                    r := r - y
                    q := q + 1
              end
        end
```

The post condition that needs to be satisfied after the program terminates is

(A) $\{r = qx + y \wedge r < y\}$
(B) $\{x = qy + r \wedge r < y\}$
(C) $\{y = qx + r \wedge 0 < r < y\}$
(D) $\{q + 1 < r - y \wedge y > 0\}$

15. Consider the following C function **[2015]**

```
int fun(int n) {
    int x = 1, k;
    if (n == 1) return x;
    for (k = 1; k < n; ++k)
        x = x + fun(k) * fun(n - k);
    return x;
}
```

The return value of fun(5) is _____

16. Consider the following recursive C function

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

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

(A) 15 (B) 25
(C) 35 (D) 45

17. Consider the following C program **[2015]**

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

The output of the program is _____

18. Suppose $c = < c[0],\ldots,c[k-1] >$ is an array of length k, where all the entries are from the set $\{0, 1\}$. For any positive integers a and n, consider the following pseudo code. **[2015]**

DOSOMETHING (c, a, n)
$z \leftarrow 1$
for $i \leftarrow 0$ to $k - 1$
do $z \leftarrow z^2 \bmod n$
if $c[i] = 1$
then $z \leftarrow (z \times a) \bmod n$
return z

If $k = 4$, $c = <1, 0, 1, 1>$, $a = 2$ and $n = 8$, then the output of DOSOMETHING(c, a, n) is _____

19. What will be the output of the following C program? **[2016]**

```
void count (int n) {
static int d = 1;
printf("%d ",n);
printf("%d ",d);
d ++;
if (n > 1) count (n -1);
printf("%d ", d);
}
void main ( ) {
count (3);
}
```

(A) 3 1 2 2 1 3 4 4 4
(B) 3 1 2 1 1 1 2 2 2
(C) 3 1 2 2 1 3 4
(D) 3 1 2 1 1 1 2

20. The following function computes X^Y for positive integers X and Y. **[2016]**

```
int exp (int X, int Y)
{
int res = 1, a = X, b = Y;
while (b! = 0)
{
if (b%2 = = 0) {a = a*a; b = b/2;}
else {res = res *a; b = b -1;}
}
return res;
}
```

Which one of the following conditions is **TRUE** before every iteration of the loop?

(A) $X^Y = a^b$
(B) $(\text{res} * a)^Y = (\text{res} * X)^b$
(C) $X^Y = \text{res} * a^b$
(D) $X^Y = (\text{res} * a)^b$

21. Consider the following two functions.

```
void fun1 (int n) {
    if (n == 0) return;
    printf ("%d", n);
    fun2 (n - 2) ;
    printf ("%d", n);
}
```

```
void fun2 (int n) {
if (n == 0) return;
printf ("%d", n);
fun1(++n)
printf ("%d", n);
}
```

The output printed when fun1 (5) is called is **[2017]**

(A) 53423122233445 (B) 53423120112233
(C) 53423122132435 (D) 53423120213243

22. Consider the C functions foo and bar given below:

```
int foo (int val) {
   int x = 0;
   while (val > 0) {
      x =x + foo (val--);
   }
   return val;
}
int bar (int val) {
   int x = 0;
   while (val > 0) {
      x =x +bar (val - 1);
      }
   return val;
}
```

Invocations of foo (3) and bar (3) will result in: **[2017]**

(A) Return of 6 and 6 respectively.
(B) Infinite loop and abnormal termination respectively.
(C) Abnormal termination and infinite loop respectively.
(D) Both terminating abnormally.

23. The output of executing the following C program is_____.

```
# include <stdio.h>
int total (int v)  {
   static int count = 0;
   while (v)  {
      count + = v&1;
      v >> = 1;
   }
    return count;
}
void main ( )  {
   static int x = 0;
   int i = 5;
   for (; i > 0,i--)  {
      x = x + total (i);
   }
   printf ("%d\n", x);
}
```

[2017]

24. Consider the following C program:

```
#include <stdio.h>
int counter = 0;
int calc (int a, int b) {
   int c;
   counter++;
   if (b==3) return (a*a*a);
   else {
      c = calc (a, b/3);
      return (c*c*c);
   }
}
int main () {
   calc (4, 81);
   printf ("%d", counter);
}
```

The output of this program is _____. **[2018]**

25. Consider the following program written in pseudo-code. Assume that *x* and *y* are integers.

```
Count (x,y) {
if (y ! = 1) {
   if (x ! = 1) {
      print ("*");
      Count (x/2, y);
   }
   else {
      y = y-1;
      Count (1024, y);
      }
   }
}
```

The number of times that the print statement is executed by the call count (1024, 1024) is _____. **[2018]**

Answer Keys

Exercises

Practice Problems 1

1. A	**2.** C	**3.** A	**4.** D	**5.** D	**6.** D	**7.** D	**8.** C	**9.** B	**10.** A
11. C	**12.** C	**13.** C	**14.** D	**15.** C					

Practice Problems 2

1. D	**2.** C	**3.** B	**4.** B	**5.** C	**6.** C	**7.** A	**8.** C	**9.** C	**10.** A
11. B	**12.** C	**13.** B	**14.** A	**15.** B					

Previous Years' Questions

1. C	**2.** -	**3.** –D	**4.** B	**5.** C	**6.** D	**7.** B	**8.** D	**9.** C	**10.** B
11. C	**12.** 9	**13.** 1.72 to 1.74		**14.** B	**15.** 51	**16.** B	**17.** 230	**18.** 0	**19.** A
20. C	**21.** A	**22.** C	**23.** 23	**24.** 4	**25.** 10230				

Chapter 3

Arrays, Pointers and Structures

LEARNING OBJECTIVES

- *Arrays*
- *Array initialization*
- *Passing array elements to function*
- *Two dimensional arrays*
- *Syntax for 3D array declaration*
- *Pointers*
- *Pointer to pointer*
- *Pointer to void (generic pointer)*
- *Array of pointers*
- *Pointer to function*
- *Dynamic memory management*
- *Memory allocation function*
- *Realloc*
- *Structures*
- *Nesting of structures*
- *Array of structures*
- *Structures & functions*
- *Union*
- *Declaration*
- *Bit fields*

Arrays

In *C* we have the following derived data types:

- Arrays
- Pointers
- Structures
- Unions

Imagine a problem that requires to read, process, and print 10 integers. We can declare 10 variables, each with different name. Having 10 different names creates a problem; we need 10 read and 10 write statements each for different variable.

Definition

An array is a collection of elements of same data type. Array is a sequenced collection. So, we can refer to the elements in array as 0th element, 1st element, and so on, until we get the last element. The array elements are individually addressed with their subscripts/indices along with array name. We place subscript value in square brackets ([ ]) followed by array name. This notation is called indexing.

There is a powerful programming construct, loop, that makes array processing easy. It does not matter if there are 1, 10, 100 or 1000 elements.

We can also use a variable name in subscript, as the value of variable changes; it refers different elements at different times.

Syntax of Array Declaration

```
Data_type array_name_[size];
```

Here, data type says the type of elements in collection, array_name is the name given to collection of elements and size says the number of elements in array.

Example: int marks[6];

Here, 'int' specifies the type of variable, marks specifies name of variable. The number 6 tells the dimension/size. The '[ ]' tells the compiler that we are dealing with array.

Accessing array elements: All the array elements are numbered, starting from 0, thus marks [3] is not the third, but the fourth element.

Example: marks[2] – 3rd element
marks[0] – 1st element
We can use the variable as index.
Thus marks[i] – ith element. As the value of i changes, refers different elements in array.

Summary about Arrays

- An array is a collection of similar elements.
- The first element in array is numbered 0, and the last element is one less than the total size of the array.
- An array is also known as subscripted variable.
- Before using an array, its type and dimension must be declared.
- How big an array is, its elements are always stored in contiguous memory locations.
- Individual elements accessed by index indicating relative position in collection.
- Index of an array must be an integer.

Array Initialization

Syntax

```
Data_type array_name[size] = {values};
```

Example:

```
int n[6]= {2,4,8,12,20,25}; // Array initialized with list of values
int num[10] = {2,4,12,20,35};
// remaining 5 elements are initialized with 0
// values
int b[10] = {0}; // Entire array elements initialized with 0.
```

Note:

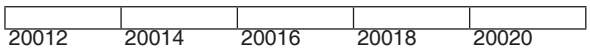
- Till the array elements are not given any specific value, they are supposed to contain garbage values.
- If the number of elements used for initialization is lesser than the size of array, then the remaining elements are initialized with zero.
- Where the array is initialized with all the elements, mentioning the dimension is optional.

Array Elements in Memory

Consider the following declaration – int num[5].

What happens in memory when we make this declaration?

- 10 bytes get received in memory, 2 bytes each for 5 integers.
- Since array is not initialized, all five values in it would be garbage. This happens because the default storage class is auto. If it is declared as static, all the array elements would be initialized with 0.



Note: In *C*, the compiler does not check whether the subscript used for array exceeds the size of array.

Data entered with a subscript exceeding the array size will simply be placed in memory out size the array, and there will be no error/warning message to warn the programmer.

Passing array elements to function

Array elements can be passed to a function by value or by reference.

Example: A program to pass an array by value:

```
void main( )
{
void display(int[ ]);// Declaration
int marks[ ] = {10,15,20,25,30};
display (marks);// function call
}
void display(int n[ ] )// function definition
{
int i;
for(i = 0 ; i < 5 ; i++)
printf("%d ", n[ i ] );
}
```

Output:

10 15 20 25 30

Here, we are passing the entire array by name. The formal parameter to receive is declared as an array, so it receives entire array elements.

To pass the individual elements of an array, we have to use index of element with array name.

Example: display (marks[*i*]); sends only the ith element as parameter.

Example: A program to demonstrate call by reference:

```
void main( )
{
void display (int *);
int marks[ ] = {5, 10 15, 20, 25};
display(&marks[0]);
}
void display(int *p)
{
int i;
for(i = 0; i < 5; i++)
printf("%d ",*(p+i));
}
```

Output:

5 10 15 20 25

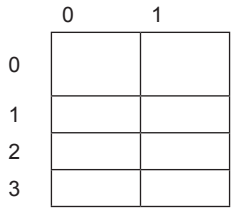
Here, we pass the address of very first element. Hence, the variable in which this address is collected (*p*) is declared as a pointer variable.

Note: Array elements are stored in contiguous memory location, by passing the address of the first element; entire array elements can be accessed.

Two-dimensional Arrays

In *C* a two-dimensional array looks like an array of arrays, i.e., a two-dimensional array is the collection of one-dimensional arrays.

Example: `int x[4][2];`



By convention, first dimension says the number of rows in array and second dimension says the number of columns in each row.

In memory, whether it is one-dimensional or a two-dimensional array, the array elements are stored in one continuous chain.

The arrangement of array elements of a two-dimensional array in memory is shown below:

X[0][0]	x[0][1]	x[1][0]	x[1][1]	x[2][0]	X[2][1]	X[3][0]	X[3][1]
6000	6002	6004	6006	6008	6010	6012	6014

Initialization

We can initialize two-dimensional array as one-dimensional array:

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

The nested braces can be used to show the exact nature of array, i.e.,

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

Here, we define each row as a one-dimensional array of two elements enclosed in braces.

Note: If the array is completely initialized with supplied values, then we can omit the size of first dimension of an array (the left most dimension).

- For accessing elements of multi-dimensional arrays, we must use multiple subscripts with array name.
- Generally, we use nested loops to work with multi-dimensional array.

Multidimensional Arrays

C allows array of two or more dimensions and maximum numbers of dimensions a C program can have depends on the compiler, we are using. Generally, an array having one dimension is called 1D array; array having two dimensions is called 2D array and so on.

Syntax:
type array-name[*d*1] [*d*2] [*d*3] [*d*4]…[*dn*];
where dn is the size of last dimension.

Example:
int table[5][5][20];
float arr[5][6][5][6][5];
In our example array "table" is a 3D. (A 3D array is an array of array of array)

Declaration and Initialization of 3D array

A 3D array can be assumed as an array of arrays; it is an array of 2D arrays and as we know 2D array itself is an array of 1D arrays. A diagram can help you to understand this.

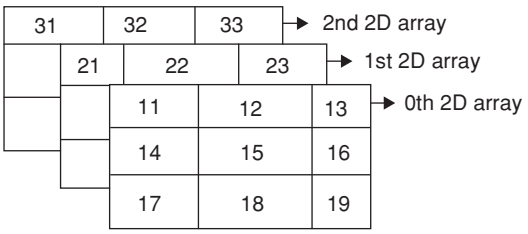


Figure 1 3D array conceptual view

Example:

```
void main( )
{
int i, j, k;
int arr [3] [3] [3] =
{
{11, 12, 13},
{14, 15, 16},
{17, 18, 19}
},
{21, 22, 23},
{24, 25, 26},
{27, 28, 29}
},
{31, 32, 33},
{34, 35, 36},
{37, 38, 39}
}
};
printf("3D Array Elements \n");
for (i = 0; i<3; i++)
{
for(j =0; j <3; j++)
{
for (k= 0; k<3; k++)
{
printf ("% d\t", arr[i][j][k]);
}
printf ("\n");
}
printf ("\n);
}
}
```

Output: 3D Array Elements

11	12	13
14	15	16
17	18	19
21	22	23
24	25	26
27	28	29
31	32	33
34	35	36
37	38	39

Syntax for 3D Array Declaration

data–type array–name [table] [row] [column];
To store values in any 3D array, first point to table number, row number and lastly to column number.

POINTERS

Pointer is a variable which contains address of another variable. *C*'s clever use of pointers makes it the excellent language.

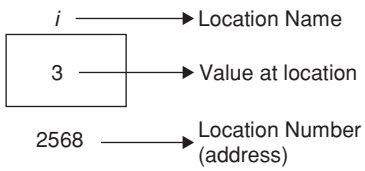
Consider the declaration:

```
int i = 3;
```

The declaration tells the *C* compiler to:

- Reserve space in memory to hold in integer value.
- Associate the name *i* with this memory location.
- Store the value 3 at this location.

Memory map is:



Computer may choose different location at different times for same variable. The important point is the address is a number.

The expression '&i' gives the address of variable '*i*'.

p = &i;

Assigns the address of '*i*' to variable '*p*'.

The variable '*p*' is declared as:

int *p;

* tells the compiler that variable '*p*' is an address variable.

Memory map of i, *p is –

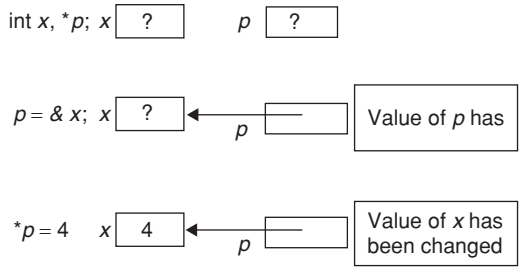
*p	i
2568	3
2720	2568

Now, pointer '*p*' is referring to the variable '*i*'.

The variable '*i*' can be accessed in two ways:

- By using the name of variable.
- By using the pointer variable referring to location '*i*'.

The operator '*' can also be used along with pointer variable in expressions. The operator '*' acts as indirection operator.



Usage of '*p*' refers to value of '*p*', where as '**p*' refers to value at the address stored in '*p*', i.e., value of '*i*'.

Example:

```
int *p;
float *x;
char *ch ;
```

Here, *p*, *x* and ch are pointer variables, i.e., variables capable of holding address. Since addresses are always whole numbers, pointers would always contain whole numbers.

The declaration float *x does not mean that *x* contains floating value, *x* will contain address of floating point variable. Similarly, 'ch' contains address of char value.

Pointer to Pointer

We know, pointer is a variable that contains address of another variable. Now this variable address might be stored in another pointer. Thus, we now have a pointer that contains address of another pointer, known as pointer to pointer.

Example:

```
void main()
{
int i = 3, *p, **q;
p = &i;
q =&p;
printf("\n Address of i = %u", &i);
printf("\n Address of i = %u", p);
printf("\n Address of i = %u", *q);
printf("\n Address of p= %u", &p);
printf(\n Address of p= %u, q);
printf("\n Address of q = %u", &q)
printf('\n value of i= %d",i);
printf('\n value of i= %d",*(&i));
printf('\n value of i= %d",*p);
printf('\n value of i= %d",**q);
}
```

If the memory map is

**q	*p	i
2010	2000	3
2050	2010	2000

Then the output is:

Address of *i* = 2000
Address of *i* = 2000
Address of *i* = 2000
Address of *p* = 2010
Address of *p* = 2010
Address of *q* = 2050
Value of *i* = 3
Value of *i* = 3
Value of *i* = 3
Value of *i* = 3

Note: We can extend pointer to a pointer to pointer. In principal, there is no limit on how far we can go on extending this definition.

Pointers for Inter-function Communication

We know that functions can be called by value and called by reference.

- If the actual parameter should not change in called function, pass the parameter-by value.

- If the value of actual parameter should get changed in called function, then use pass-by reference.
- If the function has to return more than one value, return these values indirectly by using call-by-reference.

Example: The following program demonstrates how to return multiple values.

```
void main( )
{
void areaperi(int, int *, int *);
int r;
float a,p;
printf("\n Enter radius of a circle");
scanf("%d", &r);
areaperi(r, &a, &p);
printf("Area = %f", a);
printf("\n Perimeter = %f", p);
}
void areaperi(int x, int *p, int *q)
{
*p = 3.14*x*x;
*q = 2 * 3.14*x;
}
```

Output:
Enter radius of circle 5
Area = 78:500000
Perimeter = 31.400000

Compatibility: Pointers have a type associated with them. They are not just pointer types, but rather are pointers to a specific type. The size of all pointers is same, which is equal to size of int. Every pointer holds the address of one memory location in computer, but size of variable that the pointer references can be different.

Pointer to Void (Generic Pointer)

A pointer to void is a generic type; this can point to any type. Its limitation is that the pointed data cannot be referenced directly. Since void pointer has no object type, so its length is undetermined; it cannot be dereference unless it is cast.

Example: The following example demonstrates generic pointer.

```
void main ( )
{
int a = 10;
float x = 5.7;
void *p;
p = &a;
printf("\n value of a = %d", *((int*)p));
p= &x;
printf ("\n value of x = % f", *((float *)p));
}
```

Output:
value of $a = 10$
value of $x = 5.700000$

Operations can be Performed on Pointers

1. Addition of a number to a pointer.

Example:
```
int i = 4, *j, *k;
j =&i;
j = j +1;
k = j +5;
```

2. Subtraction of a number from a pointer.

Example:
```
int i = 4, * j, * k;
j = &i; j = j -1;
k = j - 3;
```

3. Subtraction of one pointer from another. One pointer variable can be subtracted from another (provided both variables point to same array elements). The resulting value indicates the number of bytes (elements) separating (the corresponding array elements).

Example:
```
void main ( )
{
int a[ ] = {5,10,15,20,25} ,*i, *j;
i = &a[0];
j = &a[4];
printf("%d, %d", j-i,*j-*i);
}
```

Output: 4, 20
The expression *j-i* prints 4 but not 8. because j and i pointing to integers that are 4 integers apart.

4. Comparison of two pointer variables. Pointer variables can be compared provided both pointing to the same data type.

Notes: Do not attempt the following operations on pointers:
1. Addition of two pointers.
2. Multiplication of a pointer with a number or another pointer.
3. Division of a pointer with a number or another pointer.

Important points about pointer arithmetic

- A pointer when incremented always points to an immediately next location.
- A pointer when decremented always points to an element precedes the current element.

Notice the difference with:

```
(*p)++
```

Here, the expression would have been evaluated as the value pointed by p increased by one. The value of p would not be modified if we write

$$*p{+}{+} = *q{+}{+};$$

Because ++ has a higher precedence than *, both p and q are increased, but because both increase operators (++) are used as postfix and not prefix, the value assigned to $*p$ is

*q before both p and q are increased. And then both are increased, it would be equivalent to

```
*p = *q
++p;
++q;
```

Implementation of arrays in C

Array name is the pointer to the first element in array. The following discussion explains how pointers are used for implementing arrays in *C*.

```
int n[ ] = {10,20,30,40,50};
```

n	10	20	30	40	50
	5512	5514	5516	5518	5520

- We know that mentioning the array name gets the base address.

  ```
  int *p = n;
  ```

  Now 'p' points to 0th element of array '*n*'.
- 0th element can be accessed as *array_ name.

  ```
  int  x = *n;
  stores n[0] into 'x'.
  ```
- we can say that *array _ name and *(array _ name+0) are same. This indicates the following are same.

  ```
  num[i]
  *(num + i)
  *(i+num)
  ```

  (num is an array; i is an index)

Array of Pointers

The way there can be an array of ints or array of floats, similarly there can be an array of pointers. An array of pointers is the collection of addresses.

These arrays of pointers may point to isolated elements or an array element.

Example 1: Array of pointers pointing to isolated elements:

```
int i = 5, j=10, k =15;
int *ap[3];
ap[0] = &i; ap[1] = &j; ap[2] = &k;
```

Example 2: Array of pointers pointing to elements of an array:

```
int a[ ] = {0,20,45,50,70};
int *p[5], i ;
for(i = 0; i <5 ; i++ )
p[i] = &a[i] ;
```

Example 3: Array of pointers pointing to elements of different arrays;

```
int a[ ] = {5,10,20,25};
int b[ ] = {0,100,200,300,400};
int c[ ] = {50,150,250,350,450};
int *p[3];
p[0] = a; p[1] = b; p[2]=c;
```

Example 4: Array of pointers pointing to 0th element of each row of a two-dimensional array

```
int a[3][2] = {{1,2} {3,4}, {5,6}};
int *p[3];
p[0] = a[0]; p[1] =a[1];p[2] = a[2];
```

Pointer to Function

Function is a set of instructions stored in memory, so the function also contains the base address. This address can hold by using a pointer called pointer to function.

Syntax:

```
return_type  (*function_pointer)(parameter -
list );
```

Example: `int (*fp)(float, char, char);`

Example:

```
// pointer to functions
# include <iostream>
Using name space std;
int addition(int a, int b)
{
return (a + b);
}
int subtraction(int a, int b)
{
return (a - b) ;
}
int operation (int x, int y, int (*funtocall)
(int, int))
{
int g;
g = (*functocall)(x, y);
return (g);
}
int main( )
{
int m, n;
int (*minus)(int, int) = substraction;
m = operation(7, 5, addition);
n = operation(20, m, minus);
cout < < n;
return 0;
}
```

In the example, minus is a pointer to a function that has two parameters of type int. It is immediately assigned to point to the function subtraction, all in a single line.

Example: Program to demonstrate function pointer

```
int add(int, int);
int sub(int, int);
void main( )
{
Int (*fp) (int, int);
fp = add;
```

```
printf("\n 4+5=%d", fp(4,5));
fp = sub;
printf ("\n 4 - 5 = %d", fp(4,5));
}
int add(int x, int y)
{
return x + y;
}
int sub(int x,int y)
{
return x - y;
}
```

Output: 4 + 5 = 9
4 – 5 = –1

Pointer to structure The main usage of pointer to structure is we can pass structure as parameter to function as call by reference.

The other usage is to create linked lists and other dynamic data structures which depend on dynamic allocation.

Consider the declaration

struct employee

```
{
char name[20];
Int age;
float salary;
};
```

struct employee * *p*;

Variable of structures can be accessed using '.' Operator (or) → operator that is

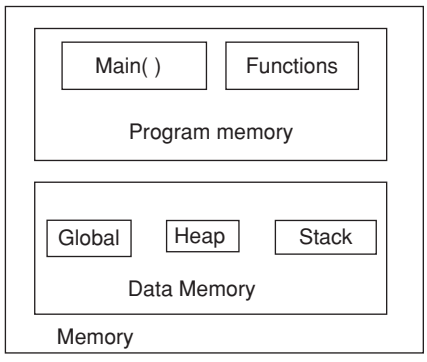
(**p*).age = 20 ; (or) *p* → age = 20;

(**p*).salary = 40, 231.0; (or) *p* → salary = 40,231.0;

Dynamic Memory Management

We can allocate the memory to objects in two ways—static and dynamic allocation. Static memory allocation requires declaration and definition of memory fully specified in the source program. The number of bytes required cannot be changed during run time. Dynamic memory allocation uses predefined functions to allocate and de-allocate memory for data dynamically during the execution of program.

We can refer to dynamically allocated memory only through pointers. Conceptual view of memory:



Memory Allocation Function

- Static memory allocation uses stack memory for variables.
- Dynamic memory management allocates memory from heap.

The following are the four memory management functions available in alloc.h and stdlib.h.

1. **Malloc (Block memory allocation):** Malloc function allocates block of memory that contained the number of bytes specified in parenthesis. It returns 'void' pointer to the first byte of allocated memory. The allocated memory is not initialized. If the memory allocation is not successful then it return NULL pointer.
   Declaration
   void *malloc (size_t size);
   The type size_t is defined as unsigned int in several header files including `stdio.h`.

   Syntax: pointer = (type*) malloc(size);
2. **Calloc (contiguous memory allocation):** Calloc is primarily used to allocate memory for arrays. It initializes the allocated memory with null characters.

   Declaration: void *calloc (size_t ele_count, size_t ele_size);

   Syntax: ptr = (type*)calloc(ele-count,ele-size);
3. **Realloc (reallocation of memory):** The realloc function is highly inefficient. When given a pointer to a previously allocated block of memory, realloc changes the size of block by deleting or extending the memory at the end of block. If the memory cannot be extended, then realloc allocates completely new block, copies the contents from existing memory location to new location, and deletes the old location.

   Declaration: void *realloc (void *ptr, size_t new_ size);

   Syntax: ptr = (type*)realloc(ptr, new_ size);
4. **Free (Releasing memory):** When the memory allocated by malloc, calloc or realloc is no longer needed, they can be freed using the function free( ).

   Declaration: void free(void *ptr);

   Syntax: free(ptr);

Free function de-allocates complete memory referenced by the pointer. Part of the memory block cannot be de-allocated.

Structures

Arrays are used to store large set of data and manipulate them but the disadvantage is that all the elements stored in an array are to be of the same data type. When we require using a collection of different data items of different data types, we can use a structure.

- Structure is a method of packing data of different types.
- A structure is a convenient method of handling a group of related data items of different data types.

Syntax for declaration

```
struct sturct_name
{
Data_type_1 var1;
Data_type_2 var2;
:
Data_type_n varn;
};
```

Example:

```
struct lib – books
{
char title [20];
char author[15];
int pages;
float price;
};
```

The keyword struct declares a structure to hold the details of four fields namely title, author, pages and price, these are members of the structures.

We can declare structure variables using the tag name anywhere in the program.

Example: struct lib – books book1, book2, book3;

- Declares book1, book2, book3 as variables of type struct lib _ books, each declaration has four elements of the structure lib_ books.

Memory map of book1:

Book1		
	Title	20 bytes
	Author	15 bytes
	Pages	2 bytes
	Price	4 bytes

- Memory will not be allocated to the structure until it is instantiated. i.e., till the declaration of a variable to structure.
- To access the members of a structure variable, *C* provides the member of (.) operator.

Example: To access author of book 1 – book1. author

Syntax: `structure_var.member_name;`

- The structures can also be initialized as any other variable of C.

Example: struct lib-books book4={"Let us *C*", "yashwanth", 450, 200.95};

Note: The values must provide in the same order as they appear in structure declaration.

- One structure variable can be assigned to another structure variable.
- Structure variables cannot be compared.

Example:

```
# include <stdio.h>
void main( )
{
Struct s1{
int id_no;
char name[20];
char address[20];
char combination[3];
    int age;
    } newstudent;
printf (" Enter student Information");
printf ("Enter student id – no");
scanf ("%d", &newstudent.id_no):
printf (" Enter the name of the student");
scanf ("%s", & newstudent.name);
printf (" Enter the address of the student");
scanf ("%s", &newstudent.address);
printf("Enter the combination of the
student")';
scanf("%s", &newstudent.combination");
printf (" Enter the age of student);
scanf ("%d ", &newstudent.age");
printf (" student information");
printf (" student id_no = %d", newstudent.
id – no);
printf("student name = %s", newstudent.
name);
printf("student address = %s", newstudent.
address);
printf ("students combination = %s", newstu-
dent. combination);
printf("Age of student = %d", newstudent.
age);
}
```

Nesting of Structures

The structures can be nested in two ways:

- Placing the structure variable as a member in another structure declaration.
- Declaration of the entire structure in another structure.

Example:

```
struct date
{
int day;
int month;
int year;
};
struct student
{
int id_no;
char name[20];
char address [20];
int age;
structure date doa;
} oldstudent, newstudent;
```

The structure 'student' contains another structure date as one of its members.

To access the day of date of admission (doa) of old student – oldstudent.doa.day.

Example:

```
struct outer
{
```

```
int o1;
float o2;
struct inner
{
int i1;
float i2;
};
} out1, out2;
```

The innermost members in a nested structure can be accessed by chaining all the concerned structure variables, from outermost to innermost; accessing i1 for out1-out1. inner.i1;

Array of Structures

It is possible to define an array of structures. For example, if we are maintaining information of all the students in the college and if 100 students are studying in the college, we need to use an array than single variables.

Example:

```
structure information
{
int id _ no;
char name[20];
char address[20];
char combination[3];
int age;
}
student[100];
```

Example:

```
# include <stdio.h>
{
struct info
{
int  id _ no;
char name[20];
char address[20];
char combination[3];
int age;
}
struct info std[100];
int, i ,n;
printf (" Enter the number of students");
scanf ("%d", &n);
scanf("Enter id_no, name, address, combination and age");
for (i = 0; i<n; i ++)
scanf(" %d %s %s %s %d", &std[i].id_no, std[i].name, std[i].address,
std[i]. combination,&std [i].age);
printf("student information");
for (i = 0 ; i < n; i ++)
printf("%d %s %s % s % d", std[i].id_no, std[i].name, std[i].address, std[i]. combination, std[i]. age);
```

Structures and Functions

- An entire structure can be passed as a parameter like any other variable.
- A function can also return a structure variable.

Example:

```
# include <stdio.h>
struct employee
{
int emp_id;
char name[25];
char department[10];
float salary;
};
void main( )
{
static struct employee emp1 = {
12, "shyam", "computer", 7500.00};
/* sending entire employee structure */
display(emp1);
}
/* function to pass entire structure variable */
display(empf)
struct employee empf
{
printf (" %d %s % s %f", empf.empid, empf.name, empf.department, empf.salary);
}
```

UNION

Union, like structure contains members whose individual data types may differ from one another. The members that compose union all share the same storage area within the computer's memory whereas each member within a structure is assigned its own unique storage area. Thus, unions are used to conserve memory.

Declaration

```
union item
{
int m;
float p;
char c;
}Code;
```

This declares a variable code of type union item.

The union contains three members each with a different data type. However, we can use only one of them at a time. The compiler allocates a piece of storage that is large enough to access a union member; we can use the same syntax that we use to access structure members, i.e.,

```
Code.m
Code.p
Code.c
```

are all valid member variables. During accessing, we should make sure that we are accessing the member whose value is currently stored.

Example:

```
union marks
{
float perc;
char grade;
}
main( )
{
union marks student1;
student1.perc = 98.5;
printf("marks are %f address is %16ℓu", stu-
dent1.perc, &student1. perc);
student1. grade = 'c';
printf("grade is %c address is %16ℓu", stu-
dent1. grade, &student1. grade);
}
```

Example:

```
# include <stdio.h>
void main ( )
{
Union u_example
{
float decval;
int p_num;
double my_value;
}U1;
U1.my_value = 125.5;
U1.pnum = 10;
U1.decval = 1000.5f;
printf("decval = %f pnum = %d my_value = % lf
", U1. decval, U1.pnum, U1.my_value);
printf(" U1 size = %d decval size =%d,
pnum size = %d my-value size = % d",
sizeof (U1), sizeof (U1.decval), sizeof
(U1.pnum), sizeof (U1.my_value));
      }
```

Bit Fields

When a program variable '*x*' is declared as int, then '*x*' takes the values from (-2^{15}) to $(2^{15} - 1)$, if *x* in the program takes only two values, 1 and 0, which requires only one bit, then the remaining 15 bits are waste.

In order to not to have this wastage, we can use bit fields with the several variables with the small enough maximal values, which can pack into a single memory location

Example:

```
struct student
{
Int gender : 1 ; // gender takes only 0,1
values
Int marriage : 2 ; // marriage takes 4(0, 1,
2, 3) values
Int marks : 7 ; // marks takes values from
0 – 127
}
```

Exercises

Practice Problems I

Directions for questions 1 to 15: Select the correct alternative from the given choices.

1. Output of the following C program is

```
intF(int x, int *py, int **pz)
{
int y, z;
** pz+= 1;
z = *pz;
*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) 30 (B) 22
(C) 20 (D) Error

2.

```
main( )
{
    char  *ptr;
    ptr = "Hello World";
     printf("%c\n",*&*ptr);
}
```

Output of the above program is
(A) Garbage value
(B) Error
(C) H
(D) Hello world

3.

```
#include <stdio.h>
main( )
{
register a =10;
char b[ ] = "Hi";
printf("%s %d ", b, a);
}
```

Output is
(A) Hi 10 (B) Error
(C) Hi (D) Hi garbage value

4.
```
main ( )
{
    int fun( ) ;
    (*fun)( ) ;
}
int fun( )
{ printf("Hello") ;
}
```
(A) Hello (B) Error
(C) No output (D) H

5. Let B be a two-dimensional array declared as B : array[1...10] [1...15] of integer;

Assuming that each integer takes one memory location the array is stored in row major order and the first element of the array is stored at location 100, what is the address of the element $B[i]\ [j]$?

(A) $15i + 10j + 84$ (B) $15i + j - 16$
(C) $15i + j$ (D) $15i + j + 84$

6. Consider the following C program which is supposed to compute the transpose of a given 4×4 matrix M. Note that, there is a Y in the program which indicates some missing statements. Choose the correct option to replace Y in the program.

```
# include <stdio.h>
int M[4][4] = { 8, 10, 9, 16, 12, 13, 11,
15, 14, 7, 6, 3, 4, 2, 1, 5 };
main( )
{
   int i, j, temp;
    for (i = 0; i<4; ++i)
     {
            Y
     }
     for (i=0; i<4; ++i)
     for (j=0; j<4; ++j)
      printf("%d", M[i] [j]);
}
```
(A)
```
for (j=0; j<4; ++j)
    {
       M[j] [i] = temp;
        temp = M[j][i];
         M[j][i] = M[i][j];
        }
```
(B)
```
for (j=0; j<4; ++j)
    {
       temp = M[j][i];
       M[i][j] = M[j][i];
        M[j][i] =  temp;
        }
```
(C)
```
for (j=i; j<4; ++j)
    {
       temp = M[i][j];
       M[i][j] = M[j][i];
        M[j][i] =  temp;
        }
```
(D)
```
for (j=i; j<4; ++j)
        {
       M[i][j] = temp;
        temp = M[j][i];
       M[j][i] = M[i][j] ;
        }
```

7. Consider the C program shown below:

```
# include <stdio.h>
# define print(a) printf("%d", a)
int a;
void z(int n)
{
  n += a;
print (n);
}
void x(int *p)
{
   int a = *p+2;
   z(a) ;
*p = a;
   print(a);
}
main(void)
  {
     a = 6;
     x(&a);
     print(a);
  }
```
The output of this program is
(A) 14 8 6 (B) 16 6 6
(C) 8 6 6 (D) 22 11 12

8. Consider the program below:

```
# include <stdio.h>
int fun(int n, int *p)
{
    int x,y;
    if (n<=1)
     {
         *p = 1;
          return 1;
    }
x = fun(n-1, p);
y = x +p;
*p = x;
return y;
}
int main( )
    {
        int a =15;
    printf( "%d\n", fun(5, &a));
    return 0;
    }
```
The output value is
(A) 14 (B) 15
(C) 8 (D) 95

9. Consider the following C program segment

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

The output of the program is
(A) string
(B) gnirt
(C) gnirts
(D) No output is printed

10.

```
# include <stdio.h>
main( )
{
    struct AA
       {
            int A = 5;
             char name[ ] = "ANU";
       };
    struct  AA  *p  =  malloc(sizeof(struct
    AA));
                  printf("%d",p->A);
                  printf("%s",p->name);
}
```

Output of the program is
(A) 5 ANU
(B) Runtime error
(C) Compiler error
(D) Linker error

11. The declaration

```
union u_tag {
int ival;
float fval;
char sval;
} u;
```

denotes *u* is a variable of type *u*_tag and
(A) *u* can have a value of int, float and char
(B) *u* can represent either integer value, float value or character value at a time
(C) *u* can have a value of float but not integer
(D) None of the above

12. If the following program is run from command line as myprog 1 2 3, what would be the output?

```
main (int argc, char *argv[ ])
{
int i;
i = argv [1] + argv [2] – argv [3];
printf ("%d", i);
}
```

(A) 123 (B) 6
(C) 0 (D) Error

13. The following C program is run from the command line as
myprog one two;
what will be the output?

```
main (int argc, char *argv [ ])
{
printf ("%c",**++argv);
}
```

(A) m (B) o
(C) myprog (D) one

14. The following program

```
change(int *);
main( ) {
int a = 4;
change(a);
printf ("%d", a);
}
change(a)
int a;
{
printf("%d", a);
}
```

Outputs
(A) 44 (B) 55
(C) 34 (D) 22

15. What is the output of the following program:

```
main( )
{
    const int x = 10;
    int *ptrx;
    ptrx = &x;
    *ptrx = 20;
    printf ("%d", x);
}
```

(A) 5 (B) 10
(C) Error (D) 20

Practice Problems 2

Directions for questions 1 to 11: Select the correct alternative from the given choices.

1. The following program segment

```
int *i;
*i = 10;
```

(A) Results in run time error
(B) Is a dangling reference
(C) Results in compilation error
(D) Assigns 10 to *i*

2. A $m \times n$ matrix is stored in column major form. The expression which accesses the (ij)th entry of the same matrix is
(A) $n \times (j-1) + i$
(B) $m \times (j-1) + i$
(C) $n \times (m-1) + ij$
(D) $m \times (n-1) + j$

3. int $* S[a]$ is 1D array of integers, which of the following refers to the third element in the array?
(A) $*(S+2)$ (B) $*(S+3)$
(C) $S+2$ (D) $S+3$

4. If an array is declared as char a[10][12]; what is referred to by *a*[5]?
(A) Pointer to 3rd Row
(B) Pointer to 4th Row
(C) Pointer to 5th Row
(D) Pointer to 6th Row

5. The following code is run from the command line as myprog 1 2 3. What would be the output?

```
main(int argc, char *argv[ ])
{
    int i, j = 0;
    for (i = 1; i < argc; i++)
    j = j + atoi (argv [i]);
    printf ("%d", j);
}
```

(A) 123 (B) 6
(C) Error (D) "123"

6. What will be the following C program output?

```
main (int argc, char *argv[ ], char *env
[ ]) {
int i;
for(i = 1; i < argc; i++)
printf ("%s", env[i]);
}
```

(A) List of all arguments
(B) List of all path parameters
(C) Error
(D) List of environment variables

7. The declaration

```
enum colors {
    red,
    blue,
    yellow = 1,
    green
};
```

assigns the value 1 to
(A) Red and Yellow
(B) Blue
(C) Red and blue
(D) Blue and yellow

8. What would be the output of the following program?

```
sum = 0;
for (i = -10; i < 0; i++)
    sum = sum + abs(i);
printf ("%d", sum);
```

(A) 100 (B) −505
(C) 55 (D) −55

9. An integer occupies 2 bytes of memory, float occupies 4 bytes and character occupies 1 byte. A structure is defined as:

```
struct tab {
    char a;
    int b;
    float c;
} table [10];
```

Then the total memory requirement (in bytes) is
(A) 14 (B) 70
(C) 40 (D) 100

10. What are the values of u1 and u2?

```
int u1, u2;
int x = 2;
int *ptr;
u1 = 2*(x + 10);
ptr = &x;
u2 = 2*(*ptr + 10);
```

(A) $u1 = 8, u2 = 16$
(B) $u1 = 23, u2 = 24$
(C) $u1 = 24, u2 = 24$
(D) None of the above

11. What is the output?

```
func(a, b)
int a, b;
{
return (a = (a = = b));
}
main ()
{
int process(), func();
printf("The value of process is %d", pro-
cess (func,3,6));
}
process (pf, val1, val2)
int (*pf) ();
```

```
int val1, val2;
{
return ((*pf) (val1, val2));
}
```

(A) The value of process is 0
(B) The value of process is 3
(C) The value of process is 6
(D) Logical error

Previous Years' Questions

1. 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 statement is TRUE? **[2014]**

(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.

2. 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 = 1; 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 My X is the **[2014]**

(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*.

3. The output of the following C program is _____ **[2015]**

```
void f1 (int a, int b)  {
      int c;
      c=a; a=b; b=c;
}
void f2(int *a, int *b)  {
      int c;
      c=*a; *a=*b; *b=c;
}
int main ( ) {
      int a=4, b=5, c=6;
      f1 (a, b);
      f2 (&b, &c);
      printf("%d", c-a-b);
}
```

4. What is the output of the following C code? Assume that the address of *x* is 2000 (in decimal) and an integer requires four bytes of memory. **[2015]**

```
int main ( ) {
  unsigned int x[4] [3] =
  { {1, 2, 3}, {4, 5, 6}, {7, 8, 9},
  {10, 11, 12}};
  printf ("%u, %u, %u", x + 3, *(x +
  3), *(x + 2) + 3);
}
```

(A) 2036, 2036, 2036 (B) 2012, 4, 2204
(C) 2036, 10, 10 (D) 2012, 4, 6

5. Consider the following function written in the C programming language. **[2015]**

```
void foo(char *a {
      if ( *a && *a != ' '){
            foo(a + 1);
            putchar(*a);
      }
}
```

The output of the above function on input "ABCD EFGH" is

(A) ABCD EFGH (B) ABCD
(C) HGFE DCBA (D) DCBA

6. Consider the following C program segment. **[2015]**

```
#include <stdio.h>
int main()
{
   char s1[7] = "1234", *p;
```

```
    p = s1 + 2;
    *p = '0';
    printf("%s", s1);
}
```

What will be printed by the program?

(A) 12 (B) 120400
(C) 1204 (D) 1034

7. Consider the following C program **[2015]**

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

8. Consider the following C program. **[2016]**

```
void f (int, short);
void main( )
{
int i = 100;
short s = 12;
short *p = &s;
_____; // call to f( )
}
```

Which one of the following expressions, when placed in the blank above, will **NOT** result in a type checking error?

(A) $f(s,*s)$ (B) $i = f(i,s)$
(C) $f(i,*s)$ (D) $f(i,*p)$

9. Consider the following *C* program. **[2016]**

```
# include<stdio.h>
void mystery (int *ptra, int *ptrb) {
int *temp;
temp = ptrb;
ptrb = ptra;
ptra = temp;

}
int main ( ) {
int a = 2016, b = 0, c = 4, d = 42;
mystery (&a, &b);
if (a < c)
    mystery(&c, &a);
mystery (&a, &d);
printf("%d\n", a)
}
```

The output of the program is _____.

10. The following function computes the maximum value contained in an integer array p [ ] of size n ($n >= 1$). **[2016]**

```
int max (int *p, int n) {
    int a = 0, b = n – 1;
    while (_____) {
            if (p [a] <= p [b]) {a = a+1;}
            else                { b = b – 1;}
    }
    return p[a];
}
```

The missing loop condition is

(A) $a != n$
(B) $b != 0$
(C) $b > (a + 1)$
(D) $b != a$

11. The value printed by the following program is ___. **[2016]**

```
void f (int* p, int m) {
m = m +5;
*p = *p + m;
return;
}
void main () {
int i = 5, j = 10;
f(&i, j);
print f ("%d", i +j);
}
```

12. Consider the following program: **[2016]**

```
int f (int *p, int n)
{   if (n <= 1) return 0;
else return max (f (p +1, n – 1), p [0] – p [1] );
}
int main ()
{
int a[ ] = {3,5,2,6,4};
printf ("%d", f(a,5));
}
```

Note: *max (x,y) returns the maximum of x and y.*

The value printed by this program is _____

13. Consider the following C code:

```
# include <stdio.h>
int *assignval (int *x, int val) {
     *x = val;
     return x;
}
void main ( )  {
    int *x = malloc (sizeof (int));
    if (NULL == x) return;
     x = assignval (x, 0);
     if (x)  {
          x  =  (int  *)  malloc
          (sizeof (int));
          if (NULL == x) return;
          x = assignval (x, 10);
     }
    printf("%d\n", *x);
    free (x);
}
```

The code suffers from which one of the following problems: **[2017]**

(A) compiler error as the return of malloc is not type-cast appropriately

(B) compiler error because the comparison should be made as x == NULL and not as shown

(C) compiles successfully but execution may result in dangling pointer

(D) compiles successfully but execution may result in memory leak

14. Consider the following C program.

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

Recall that strlen is defined in string.h as returning a value of type size_t, which is an unsigned int. the output of the program is _____. **[2017]**

15. Given the following binary number in 32-bit (single precision) IEEE-754 format:

00111110011011010000000000000000

The decimal value closest to this floating-point number is **[2017]**

(A) 1.45×10^{1} (B) 1.45×10^{-1}

(C) 2.27×10^{-1} (D) 2.27×10^{1}

16. Match the following:

(P) static char var;	(i) Sequence of memory locations to store addresses
(Q) m = malloc (10); m = NULL;	(ii) A variable located in data section of memory
(R) char *ptr [10];	(iii) Request to allocate a CPU register to store data
(S) register int var1;	(iv) A lost memory which cannot be freed

[2017]

(A) P → (ii), Q → (iv), R → (i), S → (iii)

(B) P → (ii), Q → (i), R → (iv), S → (iii)

(C) P → (ii), Q → (iv), R → (iii), S → (i)

(D) P → (iii), Q → (iv), R → (i), S → (ii)

17. Consider the following function implemented in C:

```
void printxy (int x, int y) {
     int *ptr;
     x = 0;
     ptr = &x;
     y = *ptr;
     *ptr = 1;
     printf ("%d, %d" x, y);
}
```

The output of invoking printxy (1, 1) is **[2017]**

(A) 0, 0 (B) 0, 1

(C) 1, 0 (D) 1, 1

18. Consider the following snippet of a C program. Assume that swap (&x, &y) exchanges the contents of x and y.

```
int main ()  {
      int array[] = {3, 5, 1, 4, 6, 2};
      int done = 0;
      int i;
while (done == 0)  {
    done = 1;
    for (i=0; i <=4; i++)  {
        if (array[i] < array[i+1]) {
            swap(&array[i], &array[i + 1]) ;
            done = 0;
        }
    }
    for (i=5; i >=1; i--)  {
         if (array[i] > array[i-1]) {
              swap(&array[i], &array[i-1]);
              done = 0;
         }
     }
  }
```

```
    printf{"%d", array[3]);
}
```

The output of the program is _____. **[2017]**

19. Consider the following C Program.

```
#include<stdio.h>
#include<string,h>
int main ()  {
            char* c = "GATECSIT2017";
            char* p = c;
            printf{"%d",
            (int) strlen(c+2[p]-6[p]-1)) ;
            return 0;
}
```

The output of the program is _____. **[2017]**

20. Consider the following C program.

```
#include<stdio.h>
struct Ournode {
    char x, y, z;
} ;
Int main () {
    struct Ournode p = {'1', '0', 'a'+2};
    struct Ournode *q = &p;
    printf ("%c, %c", *( (char*)q+1),
                      * ( (char*)q+2) );
    return 0;
}
```

The output of this program is: **[2018]**

(A) 0, c
(B) 0, a+2
(C) '0', 'a+2'
(D) '0', 'c'

21. Consider the following C program:

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

The output of the program above is: **[2018]**

(A) Hi Bye Bye Hi (B) Hi Bye Hi Bye
(C) Bye Hi Hi Bye (D) Bye Hi Bye Hi

Answer Keys

Exercises

Practice Problems 1

1. B **2.** C **3.** A **4.** A **5.** D **6.** C **7.** A **8.** C **9.** D **10.** C
11. B **12.** D **13.** B **14.** A **15.** D

Practice Problems 2

1. B **2.** B **3.** A **4.** D **5.** B **6.** D **7.** D **8.** C **9.** B **10.** C
11. A

Previous Years' Questions

1. D **2.** A **3.** –5 **4.** A **5.** D **6.** C **7.** 140 **8.** D **9.** 2016 **10.** D
11. 30 **12.** 3 **13.** D **14.** 3 **15.** C **16.** A **17.** C **18.** 3 **19.** 2 **20.** A
21. A

Chapter 4

Linked Lists, Stacks and Queues

LEARNING OBJECTIVES

- *Data structure*
- *Linked list*
- *Single-linked list*
- *Double-linked list*
- *Circular linked list*
- *Double circular-linked list*
- *Stack*
- *Queue*
- *Double-ended queue*
- *Circular queue*
- *Priority queue*
- *Array implementation*
- *Linked list implementation*
- *Linked list implementation of priority queue*

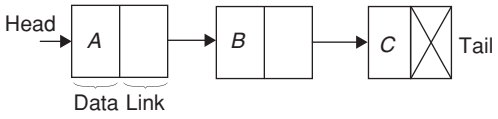
Data Structure

Data structure represents the logical arrangement of data in computer memory for easily accessing and maintenance.

Linked List

A linked list is a data structure that consists of a sequence of nodes, each of which contains data field and a reference (i.e., link) to next node in sequence.

- Generally node of linked list is represented as self-referential structure.
- The linked list elements are accessed with special pointer(s) called head and tail.



- The principal benefit of a linked list over a conventional array is that the list elements can easily be added or removed without reallocation or reorganization of the entire structure because the data items need not be stored contiguously in memory or on disk.
- Linked lists allow insertion and removal of nodes at any point in the list.
- Finding a node that contains a given data, or locating the place where a new node should be inserted may require scanning most or all of the list elements.
- The list element does not have to occupy contiguous memory.
- Adding, insertion or deletion of list elements can be accomplished with the minimal disruption of neighbouring nodes.

Single-linked List

List in which each node contains only one link field.

Node structure

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

typedef struct node Node;

Creating a linked list with two nodes of type list node

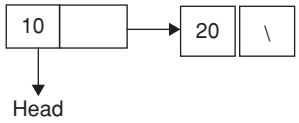
Creating a linked list with 2 nodes

struct node

```
{
Int ele;
struct node * next ;
};
typedef struct node Node ;
Node * ptr1, * ptr2;
ptr1 = getnode ();
ptr2 = getnode ();
if((ptr1) && (ptr2))
{
Printf("No memory");
exit(1);
}
Ptr1 → ele = 10;
```

```
Ptr1 → next = ptr2;
Ptr2 → ele = 20;
Ptr2 → next = NULL;
Head = ptr1;
```

the linked list appears as below



Operations on SLL (single-linked list)

- Insert at Head
- Insert at Tail
- Insert in Middle
- Delete Head
- Delete Tail
- Delete Middle
- Search
- Display

Declare two special pointers called head and tail as follows:
Node *Head, *Tail;
Head = Tail = NULL;

Head or tail is NULL represents list is empty.
Steps for Insertion:

1. Allocate memory
2. Read data
3. Adjust references

Insert head element

```
1. void ins _ Head (int x)
2. {
3. Node *temp;
4. temp = (Node *) malloc(sizeof (Node));
5. temp → ele = x;
6. temp → next = Head;
7. Head = temp;
8. if (Tail = = NULL)
9. Tail = Head;
10 }
```

- Step 4 allocates memory
- Step 5 read data
- Steps from 6 to 9 adjust reference
- 'if' condition represents first insertion

Insert tail element

```
1. void ins_tail (int x)
2. {
3. Node *temp;
4. temp = (Node *) malloc (sizeof (Node));
5. temp → ele = x;
6. temp → next = NULL;
7. Tail = temp;
8. if (Head = = NULL)
9. Head = Tail;
10. }
```

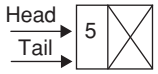
- Step 4 allocates memory
- Step 5 read data
- Steps from 6 to 9 adjust reference
- 'if' condition represents first insertion

Insert in middle/random position of list

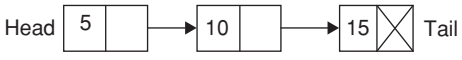
```
1. void ins _ mid (int n, int pos)
2. {
   int i = 1;
3. Node * temp, N, P; //N,P represent
   previous //& next nodes
4. if (Head == NULL)
5. {
6. ins _ head(n);
7. return;
8. }
9. temp = (Node *) malloc(sizeof(Node));
10. temp → ele = n;
11. P = head;
12. while (i < pos -1)
13. {
    P = P → next;
    i++;
    }
14. N = P → next;
15. temp → next = N;
16. P → next = temp;
17. }
```

- step 4 checks, whether the insertion is into an empty list.
- If list is empty, invokes ins–head( ) function.
- If list is not empty, then step 9 allocates memory.
- Step 10 reads data.
- Steps from 11 to 14 make the reference to the previous and next nodes of new node to be inserted.
- Steps 15 and 16 create the reference to new node from previous node and from new node to next node.

Example 1: Head = Tail = NULL
$n = 5$, P = NULL;
Here the list is empty. So,

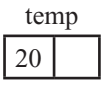


Example 2:

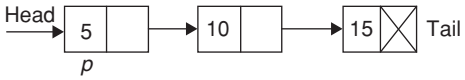


Insert element (n) 20 at position(pos) 3.
In current list, element 5 is the first element, 10 is the second and 15 is the third element.
To insert an element at pos = 3, the new node has to be placed between elements 10 and 15.
Condition in step 4 is false so step 9 executes and allocates memory.

On completion of step 10 –



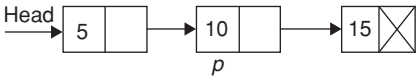
Step 11



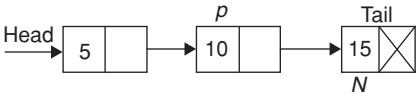
Step 12, 13

```
While (i < pos - 1)
{
P = P → next;
i++;
}
i < pos
1 < 2
```

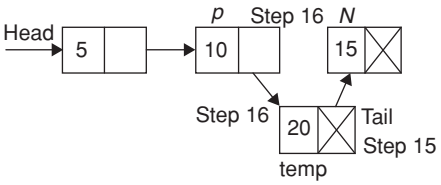
Condition true, so



i becomes 2,
2 < 2 // condition false
Step 14 makes a reference to next of previous element.



Steps 15 and 16 execute as follows:



Now the element 20 becomes the 3rd element in the list.

Deletion

- Identify the node
- Adjust the links, such that deallocation of that node does not make the list as unconnected components.
- Return/display element to delete.
- Deallocate memory.

Delete head element

```
1. void del _ head()
2. {
3. int x;
   Node * temp;
4. if (Head = = NULL)
5. {
6. printf("List empty");
7. return;
8. }
9. x = Head → ele;
10. temp = Head;
11. if (Head = = Tail)
12. Head = Tail = NULL;
13. else
14. Head = Head → next;
15. printf ("Deleted element "%d", x);
16. free(temp);
17. }
```

Step 4 – Checks for list empty
Step 9 – Reads element to delete
Step 10 – Head referred by temp pointer
Step 11 – Checks for last deletion
Step 14 – Moves the head pointer to next element in the list
Step 15 – Displays element to delete
Step 16 – Deallocates memory

Delete tail element

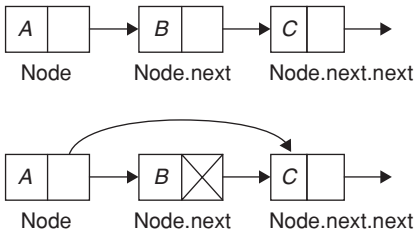
```
1. void del _ tail()
2. {
3. int x;
4. Node * temp;
5. if (Head = = NULL)
6. {
7. printf("\n list empty")
8. return ;
9. }
10. temp = Head;
11. while(temp → next ! = Tail)
12. temp = temp → next;
13. x = Tail → ele;
14. Tail = temp;
15. temp = temp → next;
16. Tail → next = NULL;
17. printf("\n Deleted element : %d", x)
18. free (temp);
19. }
```

Step 4 – Checks for list empty
Step 10, 11, 12 – Move the temp pointer to last but one node of the list
Step 13 – Reads tail element to delete
Step 14 – Moves tail pointer to last but one node
Step 15 – Moves the temp pointer to last node of the list
Step 16 – Removes the reference from tail node to temp node, i.e., tail node becomes the last element
Step 17 – Displays elements to delete
Step 18 – Deallocate memory

Delete middle element

```
1. void del _ mid (int pos)
2. {
3. int i = 1, x;
4. Node * temp P, N;
5. if(Head = = NULL)
6. {
7. printf ("\n list empty")
8. return;
9. }
10. P = head;
11. while (i < pos -1)
12. {
    P = P → next;
    i++ ;
    }
13. temp = P → next;
14. N = temp → next;
15. P → next = N;
16. x = temp → ele;
17. printf("\n Element to Delete %d", x);
18. free(temp);
19. }
```

Step 5 – Checks for empty list
Step 10, 11, 12 – Move previous pointer *P* to previous node of node to delete.
Step 13 – Temp points to node to delete
Step 14 – N points to temp next
Step 15 – Creates link from *P* to *N*
Steps 16, 17, 18 – Read and display elements to delete and deallocate memory.



Linked list using dynamic variables

Node in the linked list contains data part that is ele and link part which points to the next node, and some other external pointer will be pointing to this as these take some storage, a programmer when creating a list, should check with the available storage. For this we make use of get node ()

Function which is defined as follows:

```
struct node
{
int ele
struct node * next ;
};
typedef struct node Node;
Node getnode ()
{
Node ptr;
ptr = (Node *) malloc (size of (struct node)):
return (ptr);
}
```

If ptr returns NULL, then it is underflow (there is no available memory) otherwise, it returns start address of memory location.

Search an element

```
1. void search (int x)
2. {
3. Node * temp = head;
4. int  c = 1;
5. while (temp! = NULL)
6. {
7. if (temp → ele = = x)
8. {
9. printf("\n Element found at % d", c);
10. break;
11. }
12. c++;
13. }
14. if (temp = = NULL)
15. printf("\n search unsuccessful");
16. }
```

Step 7 – Checks temp data with search element. Repeats this step until the element is found or reaches the last node
Step 9 – Displays the position of search element in the list, if found
Step 14, 15 – Represents search element not exists in list

Display

```
1. void display ( )
2. {
3. Node *temp = Head;
4. printf("\n list elements: ");
5. while (temp ! = NULL)
6. {
7. printf("%d", temp → ele);
8. temp = temp → next;
9. }
10. }
```

Step 7 – Displays temp data
Step 8 – Moves temp pointer to next node

Algorithm to reverse direction of all links of singly liked list

Consider a linked list '*L*' with head as pointer pointing to the first node contains data element 'ele' and a pointer called 'next' which points to the next node.

Reverse is the routine which will reverse the list, there are three node pointers *P*, *Q*, *R* with *P* pointing to the first node, *Q* pointing to NULL.

```
1. START
2. if (P = NULL)
   1. print ("List is null");
   2. Exit
3. While (P)
4. R = Q;
5. Q = P;
6. P = P → next;
7. Q → next = R
8. End While
9. Head = Q;
10. STOP
```

Double-linked List (DLL)

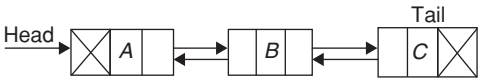
Double-linked list is a linked list in which, each node contains data part and two link fields.

Node structure:

```
struct Dnode
{
struct Dnode *prev;
int ele;
struct Dnode *next;
};
```

- prev – points to previous node in list
- next – points to next node in list
- The operations which can be performed in SLL can also be preformed on DLL.
- The major difference is that we have to adjust double reference as compared to SLL.
- We can traverse or display the list elements in forward as well as in reverse direction.

Example:



Circular-linked List (CLL)

Circular-linked list is completely same as SLL, except, in CLL the last (Tail) node points to first (Head) node of list.

So, the Insertion and Deletion operation at Head and Tail are little different from SLL.

Double Circular-linked List (DCL)

Double circular-linked list can be traversed in both directions again and again. DCL is very similar to DLL, except the last node's next pointer points to first node of list and first node's previous pointer points to last node of list.

So, the insertion and deletion operations at head and tail in DCL are little different in adjusting the reference as compared to DLL.

Storing ordered table as linked list: The table is stored as a linked list, it is retrieved and stored with two pointers, one pointer will point to node holding a record having the smallest key and other pointer performs the search.

Stack

A stack is a last in first out (LIFO) abstract data type and data structure. A stack can have any abstract data type as an element, but is characterized by only two fundamental operations.

√ PUSH
√ POP

- The PUSH operation adds an item to the top of the stack, hiding any items already on the stack or initializing the stack if it is empty.
- The POP operation removes an item from the top of the stack, and returns the poped value to the caller.
- Elements are removed from the stack in the reverse order to the order of their insertion. Therefore, the lower elements are those that have been on the stack for longest period.

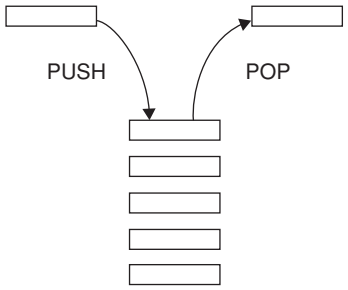


Figure 1 Simple representation of a stack

Implementation

A stack can be easily implemented either through an array or a linked list. The user is only allowed to POP or PUSH items onto the array (or) linked list.

1. **Array Implementation:** Array implementation aims to create an array where the first element inserted is placed st[0] which will be deleted last.

   The program must keep track of position top (last) element of stack.

   Operations

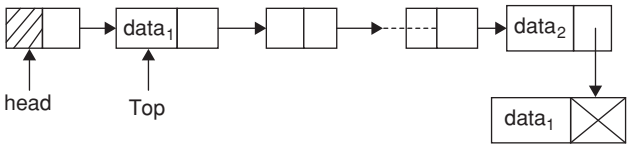
   Initially Top = –1;//represents stack empty

```
(i) Push (S, N, TOP, x)
    {
    if (TOP = = N - 1)
    printf("overflow");
    else
    TOP = TOP + 1;
    S[TOP] = x;
    }
(ii) POP (S, N, TOP, x)
    {
    if (TOP = = -1)
    printf("underflow");
    else
    x = S[TOP]
    TOP = TOP - 1
    return x;
    }
```

2. **Dynamic Implementation:** The Array implementation is also called static implementation, because the stack size is fixed.
   The stack implementation using linked list is called dynamic implementation, because the stack size can grow and shrink as the elements added or removed from the stack.
   - The PUSH operation on stack is same as insert head in SLL.
   - The POP operation is same as delete head in SLL.

Algorithm to add and delete to a link stack and link queue

Link stack:



The linked stack with head and top pointers is shown above

The algorithm to push the elements into stack is given below, the method push (item)

Steps:

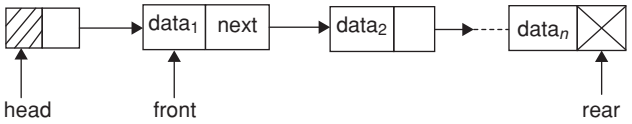
```
1. ptr = getnode (Node)
2. ptr.data = item
3. ptr.next = Top
4. Top = new
5. Head.next = Top
6. Stop.
```

for deletion of elements from stack, its algorithm is pop(), it is given below

Steps:

```
1. if (Top = NULL)
   1. print "stack is empty"
   2. exit
2. Else
   1. ptr = Top.next
   2. item = Top.data
   3. Head.next = ptr
   4. Top = ptr
3. End if
4. Stop.
```

Linked queue representation



The linked queue with head, front and rear point is shown above.

The algorithm to enqueue the elements into queue is given below, the method enqueue (item)

Steps:

```
1. ptr = getNode (Node)
2. ptr.data = item
3. ptr.next = NULL
4. if (front = NULL)
      front = ptr
      else
      rear.next = ptr;
5. end if
6. rear = ptr
7. Stop
```

For deletion of elements from queue that is ptr dequeue () is given below

Steps:

```
1. if (front = NULL)
   1. print "underflow".
   2. exit
2. ptr = front;
3. front = ptr.next
4. Head.next = front
5. item = ptr.data
6. free(ptr)
7. end.
```

Uses of Stack

- Function calls: When a function is called all local storage for the function is allocated on system 'stack', and return address also pushed on to system stack.
- Recursion stacks can be used to implement recursion if the programming language does not provide recursion facility.
- Reversing a list
- Parsing: Stacks are used by compilers to check the syntax of program.
- For evaluating expressions.

Expression Notations

Infix expression: Here binary operator comes between the operands.

Postfix expression: Here the binary operator comes after both the operands.

Example: *ab+*

Prefix expression: Here the binary operator comes before both the operands.

Example: *+ab*

Infix to postfix conversion

- If operand, output to postfix expression
- If operator, push it onto stack
- In case of parenthesis, when an opening parenthesis is read, it is pushed onto stack and when a closing parenthesis is read, all operators up to the first opening parenthesis must be popped from the stack into the post fix notation.

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

i/p	Postfix notation	Stack
(		(
A	A	(
+	A	(+
(	A	(+(
B	AB	(+(
–	AB	(+(–
C	ABC	(+(–
)	ABC–	(+
)	ABC–+	–
*	ABC–+	*
D	ABC–+D	*
	ABC–+D*	

Evaluation of postfix expression

We use operand stack for evaluation. Scan the post fix expression,

- When an operand encounters while scanning, push on to stack.
- While scanning post fix expression, if operator found then
  - Pop top two operands from stack
  - Perform the operation on those two operands
  - Push, result on to stack top
- Finally, the stack contains only one value, which represents result of the expression.

Example: 6 2 3 + – 3 8 2 / + 2 3 +

Symbol	OP1	OP2	Value	Operand stack
6				6
2				6, 2
3				6, 2, 3
+	2	3	5	6,5
–	6	5		1
3				1, 3
8				1, 3, 8
2				1, 3, 8, 2
/	8	2	4	1, 3, 4
+	3	4	7	1, 7
*	1	7	7	7
2				7, 2
*	7	2	14	14
3				14, 3
+	14	3		17

Result is 17.

Performing add, delete operations on stack (multiple stack)

Let us consider an array whose size is 'max' with multiple stack A, B having top A and top B, push and pop operations on one stack A is given below.

Algorithm for push A(x)

Initially A[Max], top $A = -1$, top B = MAX;

```
1. if (top A = top B)
   a. print " overflow"
   b. exit
2. top A = top A + 1
3. A[top A] = x
4. stop
```

Algorithm for pop A(x)

```
1. if (top A = - 1)
   a. print "underflow"
   b. exit
2. y = A[top A]
3. top A = top A - 1
4. return y
5. stop
```

Algorithm for push B(x)

```
1. if (top B - 1 = top A)
   a. print "overflow"
   b. exit
2. top B = top B -1
3. A[top B] = x
4. stop
```

Algorithm for pop B(x)

```
1. if (top B = max)
   a. print "underflow"
   b. exit
2. y = A [top B]
3. top B = top B - 1
4. return y
5. stop
```

QUEUE

A queue is an ordered collection of items from which items may be deleted at one end (called that front of queue) and into which items may be inserted at the other end (called rear of queue).

Queue is a linear data structure maintains the data in first in–first out (FIFO) order.

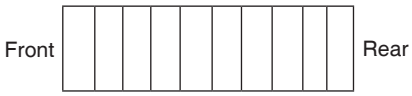
Implementation

Queue can be implemented in the following ways:

1. Array static implementation: queue cannot be extended beyond the array size.
2. Linked list dynamic implementation: Queue size increases as the elements added/inserted to queue. Queue shrinks when an element deleted from queue.

Array Implementation

const int SIZE = 10;
int *q*[SIZE];
int *f* = –1, *r* = –1; //*f* = *r* = –1 represents queue empty



Insertion

```
1. void insert (int x)
2. {
3. if (r = = SIZE -1)
4. {
5. printf("Q FULL")
6. return;
7. }
8. r++;
9. q[r] = x;
10. if (f = = -1)
11. f = r;
12. }
```

Step 3 – Checks for queue full
Step 8 – Increments rear (*r*)
Step 9 – Inserts '*x*' into queue
Step 10 – Checks whether insertion is first
Step 11 – If first insertion, updates front (*f*)

Deletion

```
1. void deletion()
2. {
3. int x;
4. if (f = = -1)
5. {
6. if ("\n Q Empty");
7. return;
8. }
9. x = q[f];
10. if (f = = r)
11. f = r = -1;
12. else
13. f++;
14. printf("\n deleted element %d", x);
15. }
```

Step 4 – Checks for queue empty
Step 9 – Deletes '*q*' front element
Step 10 – Checks whether queue having only one element
Step 11 – Rear and front initializes to –1, if queue is having only one element
Step 13 – Queue front points to next element
Step 14 – Deleted element is printed

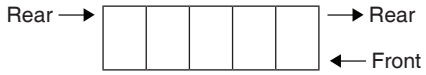
Display

```
1. void display( )
2. {
3. int i = f;
4. if (f = = -1)
5. {
6. printf("Queue Empty");
7. return;
8. }
9. printf ("\n Queue Elemetns");
10. for(; i < = r; i++)
11. printf(" %d", q[i]);
12. }
```

Step 4 – Checks for '*q*' empty
Step 10 and 11 – Display '*q*' elements

Double-ended Queue

A double-ended queue (deque) is an abstract data structure that implements a queue for which elements can only be added to or removed from the front (head) (or) rear (tail) end.



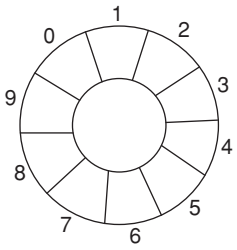
Insertions and deletions are possible at both ends.

Linked List Implementation Double-ended Queue

- Insert – Front is same as insert – Head
- Insert – Rear is same as insert – Tail
- Delete front is same as delete – Head
- Delete – Rear is same as delete – Tail

Circular Queue

As the items from a queue get deleted, the space for that item is reclaimed. Those queue positions continue to be empty. This problem is solved by circular queues. Instead of using a linear approach, a circular queue takes a circular approach; this is why a circular queue does not have a beginning or end.



The advantage of using circular queue over linear queue is efficient usage of memory.

Algorithm to implement addition and deletion from circular queue

Circular Queue Insertion:
To add an element '*X*' to a Queue '*Q*' of size '*N*' with front and rear pointers as '*F*' and '*R*' is done with insert (*X*), Initially *F* = *R* = 0.
Insert (*X*)

Steps:

```
1. if (((R = N ) & & (F = 1)) or ((R +
   1) = F))
   a. print "overflow"
   b. exit
2. if (R = N)
   then R = 0;
   Else
      R = R + 1;
3. Q[R] = x;
4. if (F = 0)
   F = 1
5. Stop.
```

To delete an element we implement an algorithm delete (). '*y*' contains the deleted element.

delete()

Steps:

```
1. if (F = 0)
   a. print "underflow"
   b. exit
2. y = Q[F]
3. if (F = R)
   F = R = 0
   else
   If (F = N)
   F = 1
   Else
   F = F + 1
4. Return y
5. Stop.
```

Priority Queue

In priority queue, the intrinsic ordering of elements does determine the results of its basic operations.

There are two types of priority queues.

- Ascending priority queue is a collection of items in which items can be inserted arbitrarily and from which only the smallest items can be removed.
- Descending priority queue is similar but allows deletion of the largest item.

Array Implementation

- The insertion operation on priority queue selects the position to the element to insert.
- Makes the position empty/free by moving the existing element (if required).
- Place the element in required position.
- Deletion operation simply deletes front of queue.

Linked-list Implementation

- Insertion operation create a node
- Reads element into node
- Find out the location
- Insert the node into list, by adjusting the reference
- Deletion operation simply deletes head elements, making the head next as head element

Linked-list Implementation of Priority Queue

- Insertion in queue is same as insert-tail of queue
- Deletion from queue is same as delete head

EXERCISES

Practice Problems I

Directions for questions 1 to 16: Select the correct alternative from the given choices.

1. If the array representation of a circular queue contains only one element then

(A) front = rear (B) front = rear + 1
(C) front = rear − 1 (D) front = rear = NULL

2. The five items *P*, *Q*, *R*, *S* and *T* are pushed in a stack, one after another starting from *P*. The stack is popped four times, and each element is inserted in a queue. The 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) *P* (B) *Q*
(C) *R* (D) *S*

3. What are the contents of the stack (initially the stack is empty) after the following operations?

PUSH (A)
PUSH (B)
PUSH (C)
POP
PUSH(D); POP; POP;
PUSH(E)
PUSH(F)
POP

(A) ABE (B) AE
(C) A (D) ABCE

4. Consider the below code, which deletes a node from the beginning of a list:

```
void deletefront()
{
if(head == NULL)
return;
else
{
.........
.........
.........
}
}
```

Which lines will correctly implement else part of above code?

(A)
```
if (head → next = = NULL)
    head = head → next;
```
(B)
```
if (head = = tail)
    head = tail = NULL;
    else
    head = head → next;
```
(C)
```
if (head = = tail = = NULL)
    head = head → next;
```
(D)
```
head = head → next;
```

5. When a new element is inserted in the middle of linked list, then the references of _____ to be adjusted/ updated.
(A) those nodes that appear after the new node
(B) those nodes that appear before the new node
(C) head and tail nodes
(D) those nodes that appear just before and after the new node

6. The following C function takes double-linked list as an argument. It modifies the list by moving the head (first) element to tail of the list.
```
typedef struct node
{
    struct node *p;
    int data;
    struct node *n;
} Node;
Node * Move - to - last (Node *head)
{
Node * temp, * prev, * next;
if (head = = NULL)||(head → n = = NULL))
return head;
temp = head;
prev = head;
head = head → n;
while (prev → n! = NULL)
{
X;
}
Y;
return head;
}
```
(A)
```
X: prev = prev → n;
Y: prev → n = temp;
temp → p = prev;
temp → n = NULL;
head → P = NULL;
```
(B)
```
X: next = prev → n;
Y: prev → n = temp;
temp → p = prev;
```
(C)
```
X: prev = prev → n;
Y: prev → n = temp;
temp → n = NULL;
head → p = NULL;
```
(D)
```
X: next = prev → n;
prev = prev → n;
Y: prev → n = Next;
next → n = head;
temp → n = NULL;
```

7. Which of the following program segment correctly inserts an element at the front of the linked list. Assume that Node represents linked list node structure, value is the element to be inserted.
(A)
```
temp = (Node *)malloc (sizeof (Node));
temp → data = value;
temp → next = head;
head = temp;
```
(B)
```
temp = (Node *)malloc(sizeof (Node*)
);
temp → data = value;
temp → next = head;
head = temp;
```
(C)
```
temp = (Node *)malloc (sizeof (Node));
head = temp;
temp → next = head;
temp → data = value;
```
(D)
```
temp = (Node *)malloc (sizeof (Node
*);
temp → data = value;
head = temp;
temp → next = head;
```

8. Consider the following program segment:
```
struct element
{
    int x;
    struct element *link;
}
void shuffle(struct element *head)
{
struct *p, *q;
int t;
if (!head || !head → link) return;
p= head ; q = head → link;
while(q)
{
t = p → x;
p→ x = q → x;
q → x = t;
p = q → link;
q = p?  p : 0;
}
}
```
The function called with list containing 10, 15, 20, 25, 30, 35, 40 in given order. What will the order of elements of the list, after executing the function shuffle?

(A)	10	15	20	25	30	35	40
(B)	40	35	30	25	20	15	10
(C)	20	15	10	25	40	35	30
(D)	15	10	25	20	35	30	40

9. Primary ADT's are
(A) Linked list only (B) Stack only
(C) Queue only (D) All of these

10. Linked list uses NULL pointers to signal
(A) end of list (B) start of list
(C) Either (A) or (B) (D) Neither (A) nor (B)

11. Which of the following is essential for converting an infix to postfix form efficiently?
(A) Operator stack (B) Operand stack
(C) Both (A) and (B) (D) Parse tree

12. Stacks cannot be used to
(A) Evaluate postfix expression
(B) Implement recursion
(C) Convert infix to postfix
(D) Allocate resource like CPU by the operating system

13. Linked list can be sorted
(A) By swapping data only
(B) By swapping address only
(C) Both (A) and (B)
(D) None of these

14. Linked list are not suitable for implementing
(A) Insertion sort
(B) Binary search
(C) Radix sort
(D) Polynomial manipulation

15. Insertion of node in a double-linked list requires how many changes to previous (prev) and next pointers?
(A) No changes (B) 2 next and 2 prev
(C) 1 next and 1 prev (D) 3 next and 3 prev

16. Minimum number of stacks required to implement a queue is
(A) 1 (B) 2
(C) 3 (D) 4

Practice Problems 2

Directions for questions 1 to 11: Select the correct alternative from the given choices.

1. Stack is useful for implementing _____.
(A) radix sort
(B) breadth first search
(C) quick sort
(D) recursion

2. Which is true about linked list?
(A) A linked list is a dynamic data structure.
(B) A linked list is a static structure.
(C) A stack cannot be implemented by a linear linked list.
(D) None of the above

3. The process of accessing the data stored in a tape is similar to manipulating data on a _____.
(A) stack (B) list
(C) queue (D) heap

4. Which of the following is used to aid in evaluating a prefix expression?
(A) Queue (B) Heap
(C) Stack (D) Hash

5. Select the statement which best completes the sentence — 'Abstract data type is…'
(A) a data type which is abstract in nature
(B) a kind of data type
(C) data structure
(D) a mathematical model together with a set of operations defined on it

6. Which of the following data structures may give an overflow error, even through the current number of elements in it is less than its size?
(A) Simple queue (B) Circular queue
(C) Stack (D) None of these

7. In a circular linked list, insertion of a record involves the modification of _____.
(A) no pointer (B) four pointers
(C) two pointers (D) All of the above

8. Among the following, which one is not the right operation on a stack?
(A) Remove the item that is inserted latest into the stack.
(B) Add an item to the stack.
(C) Remove the first item that is inserted into the stack, without deleting other elements.
(D) None of the above

9. Among the following which one is not the right operation on dequeue?
(A) Inserting an element in the middle of a dequeue.
(B) Inserting an element at the front of a dequeue.
(C) Inserting an element at the rear of a dequeue.
(D) None of the above

10. A linear list in which elements can be added or removed at either end but not in the middle is _____.
(A) queue
(B) dequeue
(C) array
(D) tree

11. The post fix notation of $A/B * * C + D * E - A * C$ is
(A) $ABC * * /DE * + AC * -$
(B) $ABC * * D/E * + AC + -$
(C) $ABC * * /DE * AC + -$
(D) $ABC * * /DE * + AC + -$

Previous Years' Questions

1. An abstract data type (ADT) is **[2005]**
(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

2. An implementation of a queue Q, using two stacks S_1 and S_2, is given below:

```
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 ($\geq n$) 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? **[2006]**
(A) $n + m \geq x < 2n$ and $2m \geq y \geq n + m$
(B) $n + m \geq x < 2n$ and $2m \geq y \geq 2n$
(C) $2m \geq x < 2n$ and $2m \geq y \geq n + m$
(D) $2m \geq x < 2n$ and $2m \geq y \geq 2n$

3. The following postfix expression with single digit operands is evaluated using a stack:
8 2 3 ∧ / 2 3 * + 5 1 * –
Note that ∧ is the exponentiation operator. The top two elements of the stack after the first * is evaluated are: **[2007]**
(A) 6 and 1 (B) 5 and 7
(C) 3 and 2 (D) 1 and 5

4. 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?

```
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;
}
}
```

[2008]
(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

5. 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]**
(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 = = FRONT
(D) Full: (FRONT + 1) mod n = = REAR
Empty: REAR = = FRONT

6. Consider the C program below **[2015]**

```
#include <stdio.h>
int *A, stkTop;
int stkFunc (int opcode, int val)
{
  static int size=0, stkTop=0;
  switch (opcode) {
    case -1: size = val; break;
    case 0: if (stkTop < size)
  A[stkTop++] =
        val; break;
    default: if (stkTop) return A[-
    -stkTop];
  }
  return -1;
}
    int main ( )
    {
    int B[20]; A = B; stkTop = -1;
    stkFunc (-1, 10);
    stkFunc (0, 5);
    stkFunc (0, 10);
    printf ("%d\n", stkFunc(1, 0) +
    stkFunc(1, 0));
    }
```

The value printed by the above program is _____

7. The result of evaluating the postfix expression 10 5 + 60 6/* 8 – is **[2015]**

(A) 284 (B) 213
(C) 142 (D) 71

8. Let Q denote a queue containing sixteen numbers and S be an empty stack.

Head (Q) returns the element at the head of the queue **Q without** removing it from **Q**. Similarly Top(S) returns the element at the top of **S without** removing it from S.

Consider the algorithm given below.

```
while Q is not Empty do
        if S is Empty OR Top(S) ≤ Head (Q)
        then
                x : = Dequeue (Q)
                Push (S, x);
        else
                x : = Pop (S);
                enqueue (Q, x);
        end
end
```

The maximum possible number of iterations of the while loop in the algorithm is ____ . **[2016]**

9. The attributes of three arithmetic operators in some programming language are given below.

Operator	Precedence	Associativity	Arity
+	High	Left	Binary
–	Medium	Right	Binary
*	Low	Left	Binary

The value of the expression

2 – 5 + 1 – 7 * 3 in this language is _____. **[2016]**

10. A circular queue has been implemented using a singly linked list where each node consists of a value and a single pointer pointing to the next node. We maintain exactly two external pointers **FRONT** and **REAR** pointing to the front node and the rear node of the queue, respectively. Which of the following statements is/are CORRECT for such a circular queue, so that insertion and deletion operations can be performed in O (1) time?

I. Next pointer of front node points to the rear node.
II. Next pointer of rear node points to the front node.

[2017]

(A) I only (B) II only
(C) Both I and II (D) Neither I nor II

Answer Keys

Exercises

Practice Problems 1

1. A **2.** C **3.** B **4.** B **5.** D **6.** A **7.** A **8.** D **9.** D **10.** A
11. A **12.** D **13.** C **14.** B **15.** B **16.** B

Practice Problems 2

1. D **2.** A **3.** C **4.** C **5.** D **6.** A **7.** C **8.** C **9.** A **10.** B
11. A

Previous Years' Questions

1. C **2.** A **3.** A **4.** B **5.** **6.** 15 **7.** C **8.** 256 **9.** 9 **10.** B

Chapter 5

Trees

LEARNING OBJECTIVES

- *Tree*
- *2-Tree*
- *Binary tree*
- *Properties of binary trees*
- *Complete binary tree*
- *Full binary tree*
- *Binary tree representation*
- *Linked representation*
- *Binary search tree*
- *Binary tree traversing methods*
- *AVL tree*
- *Binary heap*
- *Max-heap*
- *Min-heap*
- *Expression tree*

Tree

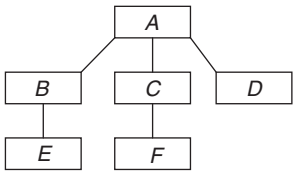
Tree is non-linear data structure designated at a special node called root and elements are arranged in levels without containing cycles.

(or)

The tree is

1. Rooted at one vertex
2. Contains no cycles
3. There is a sequence of edges from any vertex to any other
4. Any number of elements may connect to any node (including root)
5. A unique path traverses from root to any node of tree
6. Tree stores data in hierarchical manner
7. The elements are arranged in layers

Example:



- Root node is *A*.
- *A*'s children are *B*, *C* and *D*.
- *E*, *F* and *D* are leaves.
- Nodes *B*, *C* are called as intermediate nodes.
- *A* is parent of *B*, *C* and *D*.
- *B* is parent of *E* and *C* is parent of *F*.
- Number of children of a node is called degree of node.

2-Tree

A tree in which every node contains either 0 or 2 children.

Binary Tree

It is a special type of tree where each node of tree contains either 0 or 1 or 2 children.

(or)

Binary Tree is either empty, or it consists of a root with two binary trees called left-sub tree and right sub-tree of root (left or right or both the sub trees may be empty).

Properties of binary tree

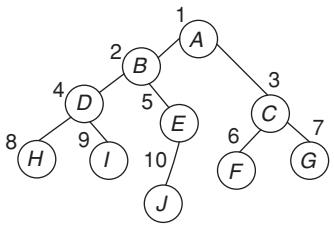
- Binary tree partitioned into three parts.
- First subset contains root of tree.
- Second subset is called left subtree.
- Another subset is called right subtree.
- Each subtree is a binary tree.
- Degree of any node is 0/1/2.
- The maximum number of nodes in a tree with height '*h*' is $2^{h+1}-1$.
- The maximum number of nodes at level '*i*' is 2^{i-1}.
- For any non-empty binary tree, the number of terminal nodes with n_2, nodes of degree 2 is $N_0 = n_2 + 1$
- The maximum number of nodes in a tree with depth *d* is $2^d - 1$.

Types of binary tree

Complete binary tree It is a binary tree, in which at every level, except possibly the last, is completely filled and all nodes at the last level are as left as possible.

Example:

Level	Height	Depth
1	3	1
2	2	2
3	1	3
4	0	4



For the given tree:

- Having 4 levels
- Height of the tree is 3
- Depth of the tree is 4
- The numbers at each node represents level order index.
- The level order Index, are assigned to nodes in the following manner
- Root of the tree is '1'
- For a node '*x*', the LOI is (2 * LOI (parent)), if '*x*' is left child of its parent.
- For a node '*y*', the LOI (2 * LOI (Parent) +1), if '*y*' is right child of its parent.

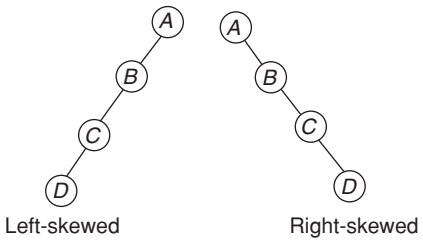
Now complete binary tree can be defined as a binary tree, which contains a sequence of numbers to its nodes as LOI's without any break in sequence.

Full binary tree It is a binary tree, for which all leaf nodes are at same level and all intermediate nodes contains exactly 2 children.
(or)
A tree with depth '*K*' contains exactly $2^K - 1$ nodes.

Strictly binary tree A binary tree in which every node contains exactly 0 or 2 children.

Skewed binary tree A binary tree in which elements are added only in one direction.

Example:



Application

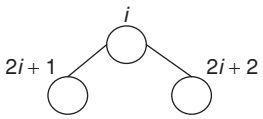
- A binary tree is useful data structure when two way decisions must be made at each point of process.

Binary tree representation

The binary trees can be represented in two ways.

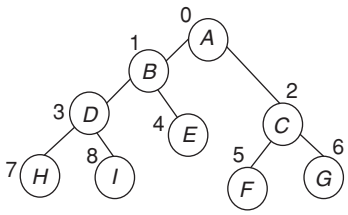
- Array
- Linked list

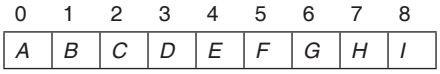
Array representation The elements of a binary tree are placed in an array using the level order index of each element.



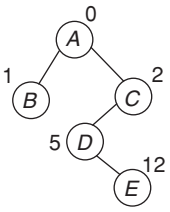
When LOI of Root is 0:

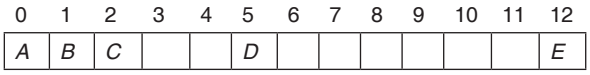
Example 1:





Example 2:

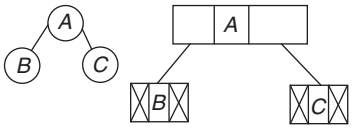




Linked representation Each node contains one data field and two link fields. Fist link point to the left child and another point to the right child.

In absence of any child, corresponding link field contains NULL.

Example:



Trade-off's between array and linked, representations

- Array representation is somewhat simpler. It must ensure elements are placed in array at proper position.
- Linked representation requires pointer to its left and right child.
- Array representation saves memory for almost complete binary trees.
- Linked representation allocates the number and nodes equal to the number of elements in tree.
- Array representation does not work efficiently for skewed binary trees.
- Array representation limits the size of binary tree to the array size.
- In linked representation, tree can be extended by adding an element dynamically and can be shrinked by deleting an element dynamically.

Binary search tree

It is a special type of binary tree that satisfies the following properties.

- All the elements of left sub tree of root are smaller than root.
- All the elements of right sub tree of root are greater than root.
- The above two properties satisfy for each subtree.

Example:

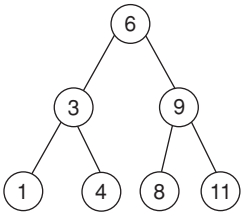


Figure 1 A data structure to encode binary search tree

The binary search tree node contains three fields, data field, left child, right child. Left child is a pointer which points to the predecessor of the node and right child is a pointer which points to the successor of the node.

A data structure to encode binary search tree is

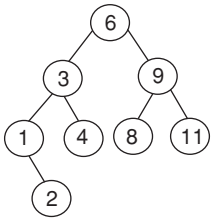
Left child	Data	Right child

The declaration is

```
Struct node
{
Struct node * left child;
Int data;
Struct node * Right child;
};
```

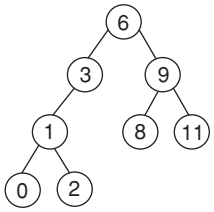
Insertion If a value to be inserted is smaller than the root, value, it must go in the left subtree, if larger it must go in the right subtree. This reasoning applies recursively until we reach a node where the required subtree does not exist and that is where we place the new value.

Example: It must go in 6's left subtree, 3's left subtree, 1's right subtree, 1 has no right subtree, so we make a singleton with 2 and it becomes 1's right subtree.

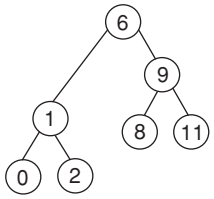


Deletion:

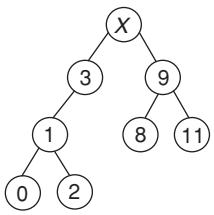
1. If a leaf node has to be deleted, just delete it and the rest of the tree is exactly as it was, so it is still a BST.
2. Suppose the node we are deleting has only one sub tree
   Example, In the following tree, '3' has only one sub-tree



To delete a node with 1 subtree, we just 'link past' the node, i.e., connect the parent of the node directly to the node's only subtree. This always works, whether the one subtree is on the left or on the right. Deleting 3 gives us.



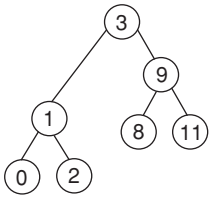
3. Deletion of node which has 2 subtrees
   Example: Delete 6.



Choose value '*X*'

1. Everything in the left subtree must be smaller than *X*.
2. Everything in the right subtree must be bigger than *X*.

We must choose *X* to be the largest value in the left subtree. In our example, 3 is the largest value in the left subtree. So we replace root node 6 with 3.



Note: We could do the same thing with the right subtree. Just use the smallest value in the right subtree.

Notes:

- The largest element in left subtree is the right most element.
- The smallest element in right subtree is the left most element.

Binary tree traversing methods

The binary tree contains 3 parts:

V – root

L – Left subtree

R – Right subtree

Pre-order: (*V*, *L*, *R*)

- Visit root of the tree first
- Traverse the left - subtree in pre-order
- Traverse the right - subtree in preorder

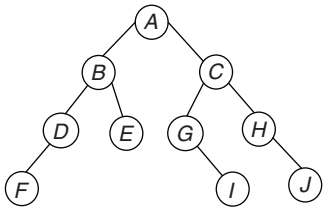
In-order: (*L*, *V*, *R*)

- Traverse the left – subtree in in-order
- Visit Root of the tree
- Traverse right - sub tree in in-order

Post-order: (*L*, *R*, *V*)

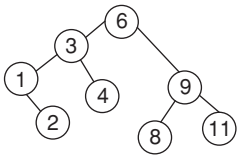
- Traverse the left subtree in post-order.
- Traverse the Right - subtree in post-order
- Visit root of the tree

Example 1:



Pre-order: *A B D F E C G I H J*
In-order: *F D B E A G I C H J*
Post-order: *F D E B I G J H C A*
Pre-order, In-order and post-order uniquely identify the tree.

Example 2:



Pre-order: 6 3 1 2 4 9 8 11
In-order: 1 2 3 4 6 8 9 11
Post-order: 2 1 4 3 8 11 9 6

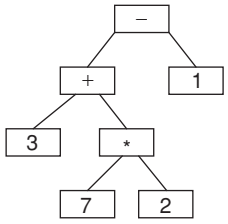
Points to remember

- Pre-order traversal contains root element as first element in traverse list.
- Post-order traversal contains root element as last in traversal list.
- For BST, in-order traversal is a sorted list.
- A unique binary tree can constructed if either pre-order or post-order traversal list provided with In order traversal list.
- If either pre-order or post-order only given then BST cannot be constructed.

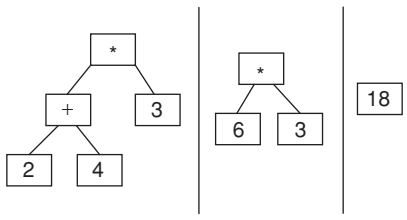
Applications

1. Binary trees can represent arithmetic expressions.
   - An infix expression will have a parent operator and two children operands.

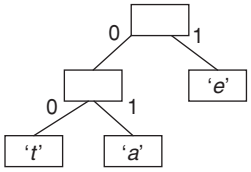
Consider the expression ((3 + (7 ∗ 2)) −1)
Each parenthesised expression becomes a tree.
Each operand is a leaf, each operator is an internal node.



2. To evaluate the expression tree:
   Take any two leaves
   Apply the parents operator to them
   Replace the operator with the value of the sub expression.



3. Binary trees in a famous file compression algorithm Huffman coding tree
   - Each character is stored in a leaf
   - The code is found by following the path 0 go left, 1 go right.
   - *a* is 01
   - *e* is 1



AVL Tree

An AVL tree is a self-balancing binary search tree, in which the heights of the two child subtrees of any node differ by atmost one.

Insertions and deletions may require the tree to be rebalanced by one or more tree rotations.

- The balance factor of a node is the height of its left subtree minus the height of its right subtree (sometimes opposite) and a node with balance factor—1, 0 or –1 is considered balanced. A node with any other balance factor is considered unbalanced and requires rebalancing the tree.
- The balance factor is either stored directly at each node or computed from the heights of the subtrees.

Insert operations

Step I: Insert a node into the AVL tree as it is inserted in a BST.

Step II: Examine the search path to see if there is a pivot node.

Three cases may arise

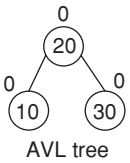
Case I: There is no pivot node. No adjustment required.

Case II: The pivot node exists and the subtree of the pivot node to which the new node is added has smaller height. No adjustment required.

Case III: The pivot node exists and the subtree to which the new node is added has the larger height, Adjustment required.

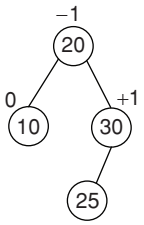
Example: The numbers at each node represents balance factor.

Example 1:



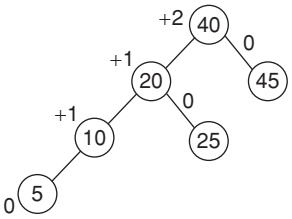
AVL tree

Example 2:



AVL tree

Example 3:



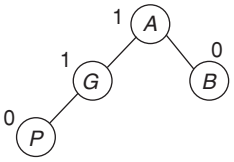
Not an AVL tree

Example 3 is not an AVL tree, because the balance factor of root node is +2.

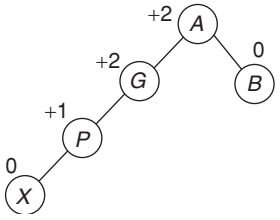
AVL tree becomes height in-balanced tree in following cases:

1. Left-Left case: An insertion in left subtree of left child of pivot node.

Example:

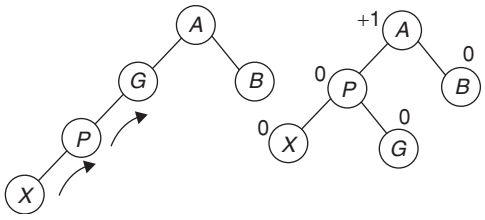


Insert '*X*' as left to node '*P*'. Here '*G*' is pivot node.



Solution:

To make the tree as balanced tree, perform **Left–Left Rotation** as follows:

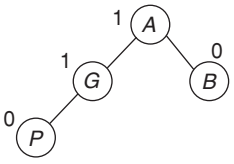


In left–left rotation

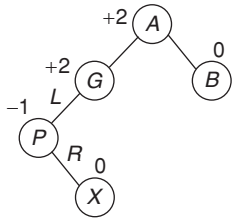
- Intermediate node '*P*' becomes root of subtree.
- Root of subtree '*G*' (pivot) becomes right subtree.
- New node '*X*' remains same as left child of '*P*'.

Left–Right Case

An insertion of left subtree of right child of pivot node.

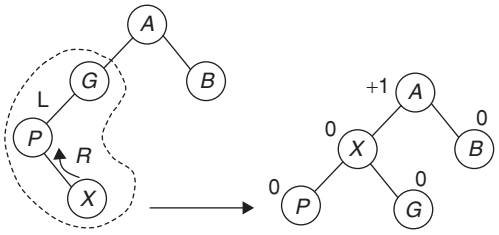


Example 1: Insert '*X*' as right child of '*P*'.



Is not an AVL tree. Height in-balance at node '*G*'.

Solution:
Perform Left–Right Rotation, to balance the height of tree.



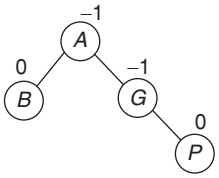
In Left–Right rotation:

- New node '*X*' becomes root of subtree.
- Root of subtree '*G*' (pivot) becomes right child of '*X*'.
- Intermediate node '*P*' becomes left child of new node.

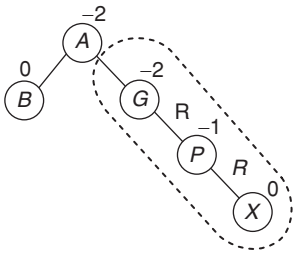
Right–Right case

An insertion of right subtree of right child of pivot node.

Example:

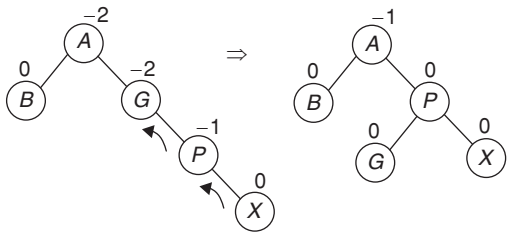


Insert '*X*' as right child of '*P*'



Is not an AVL tree, because of height in-balance at node '*G*'.

Solution:
To make the tree as balanced tree, perform the right–right rotation as follows:

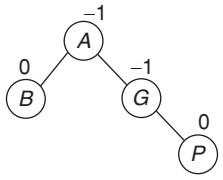


In Right–Right rotation:

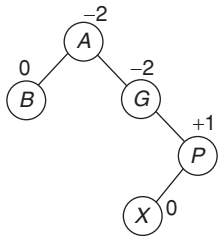
- Intermediate node '*P*' becomes root of subtree.
- Root of subtree '*G*' (pivot) becomes left child of '*P*'.
- New node '*X*' remains as right child to '*P*'.

Right–Left case

An insertion of right subtree of left child of pivot node.

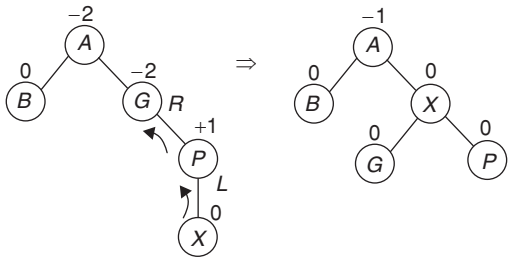


Insert '*X*' as left child of '*P*'



Is not AVL tree, because height in-balance at node '*G*'.

Solution:
To make the above tree as balanced, perform Right–Left rotation as follows:



In Right–Left Rotation:

- New node '*X*' becomes root of subtree.
- Root of subtree '*G*' (pivot) becomes left child of '*X*'.
- Intermediate node '*P*' becomes right of '*X*'.

Note: Left–Right and Right–Left rotation are also called as double rotations.

Binary Heap

A binary heap is a heap data structure created using a binary tree. It can be seen as a binary tree with two additional constraints.

The shape property: The tree is a complete binary tree; that is, all levels of the tree, except possibly the last one (deepest) level of the tree is not complete, the nodes of that level are filled, from left to right.

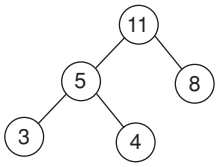
Max-Heap

A heap in which each node is greater than or equal to its children is called max-heap. Max-Heap generally used for heap sort.

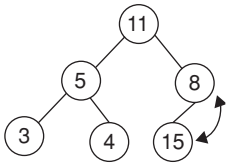
Min-Heap

A heap in which, each node is smaller than or equal to its children is called Min-Heap. Min-heap generally used to implement priority queue.

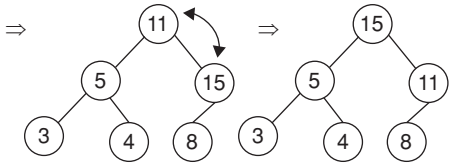
Note: By default heap represent Max-Heap:



Insert 15:

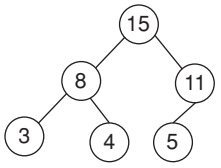


Is not satisfying heap property. So Heapify

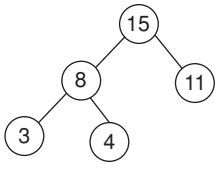


Delete 5: Deletion of a node from heap is always deletes a leaf node.

So interchange the value of last leaf node with node 5.



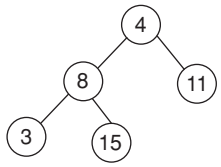
Now delete node '5'



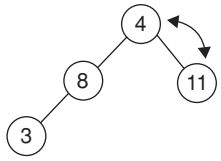
Is satisfying heap property.

Delete 15:

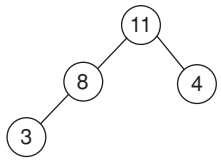
Interchange 4 and 15



Now delete Node '15'



Is not satisfying heap property. So heapify



Note: Insertion or deletion operation on a heap may require heapify process.

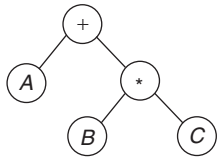
Expression Tree

The expressions can also represented by using a binary tree called expression tree.

Expression tree contains:

- Operators as intermediate nodes.
- Operands as leaf nodes (or) childs to operator nodes.
- The operator at lowest level will be having highest priority.

Example: $A + B * C$



Traversing:

Pre-order: $+ A * B C$

In-order: $A + B * C$

Post-order: $A B C * +$

Note: In-order traversal of expression tree generates In-fix expression. Similarly pre-order and post-order generates prefix and postfix, respectively.

EXERCISES

Practice Problems I

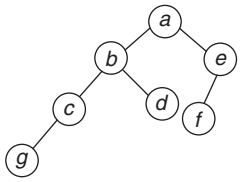
Directions for questions 1 to 15: Select the correct alternative from the given choices.

1. A binary tree T has n leaf nodes. The number of nodes of degree two in T is ____.
(A) n (B) $n - 1$
(C) $\log n$ (D) $n + 1$

2. How many numbers of binary tree can be created with 3 nodes which when traversed in post-order gives the sequence C, B, A?
(A) 3 (B) 5
(C) 8 (D) 15

3. A binary search tree contains the values 3, 6, 10, 22, 25, 30, 60, 75. The tree is traversed in pre-order and the values are printed out. Which of the following sequence is a valid output?
(A) 25 6 3 10 22 60 30 75
(B) 25 6 10 3 22 75 30 60
(C) 25 6 75 60 30 3 10 22
(D) 75 30 60 22 10 3 6 25

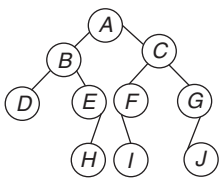
4. Figure shows a balanced tree. How many nodes will become unbalanced when a node is inserted as a child of the node 'g'?



(A) 7 (B) 2
(C) 3 (D) 8

5. A full binary tree with n non-leaf nodes contains
(A) $2n$ nodes (B) $\log_2 n$ node
(C) $n + 1$ nodes (D) $2n + 1$ nodes

6. Which of the following list of nodes corresponds to a post order traversal of the binary tree shown below?



(A) *A B C D E F G H I J* (B) *J I H G F E D C B A*
(C) *D H E B I F J G C A* (D) *D E H F I G J B C A*

7. Which of the following sequence of array elements forms as heap?
(A) {23, 17, 14, 6, 13, 10, 1, 12, 7, 5}
(B) {23, 17, 14, 6, 13, 10, 1, 5, 7, 12}
(C) {23, 17, 14, 7, 13, 10, 1, 5, 6, 12}
(D) {23, 17, 14, 7, 13, 10, 1, 12, 5, 6}

8. What is the maximum height of any AVL tree with 7 nodes? Assume that the height of a tree with a single node is 0.
(A) 2 (B) 3
(C) 4 (D) 5

9. A binary search tree is generated by inserting in order the following integers:

55, 15, 65, 5, 25, 59, 90, 2, 7, 35, 60, 23.

The number of nodes in the left subtree and right subtree of the root respectively are
(A) 8, 3 (B) 7, 4
(C) 3, 8 (D) 4, 7

10. In a complete binary tree of n nodes, how far are the most distant two nodes? Assume each in the path counts as 1.
(A) about $\log_2 n$ (B) about $2\log_2 n$
(C) about $3\log_2 n$ (D) about $4\log_2 (n)$

11. A complete binary tree of level 5 has how many nodes?
(A) 20 (B) 63
(C) 30 (D) 73

Common data for questions 12 and 13: A 3-ary max-heap is like a binary max-heap, but instead of 2 children, nodes have 3 children. A 3-ary heap can be represented by an array as follows:

The root is stored in the first location, $a[0]$, nodes in the next level from left to right is stored from $a[1]$ to $a[3]$ and so on. An item x can be inserted into a 3-ary heap containing n items by placing x in the location $a[n]$ and pushing it up the tree to satisfy the heap property.

12. Which one of the following is a valid sequence of elements in an array representing 3-ary max-heap?
(A) 1, 3, 5, 6, 8, 9 (B) 9, 6, 3, 1, 8 , 5
(C) 9, 3, 6, 8, 5, 1 (D) 9, 5, 6, 8, 3, 1

13. Suppose the elements 7, 2, 10 and 4 are inserted, in that order, into the valid 3-ary max-heap found in the above question. Which one of the following is the sequence of items in the array representing the resultant heap?
(A) 10, 7, 9, 8, 3, 1, 5, 2, 6, 4
(B) 10, 9, 8, 7, 6, 5, 4, 3, 2, 1
(C) 10, 9, 4, 5, 7, 6, 8, 2, 1, 3
(D) 10, 8, 6, 9, 7, 2, 3, 4, 1, 5

14. Consider the nested representation of binary trees : (X Y Z) indicated Y and Z are the left and right subtrees respectively, of node X(Y and Z may be null (or) further nested) which of the following represents a valid binary tree?

(A) (1 2 (4 5 6 7)) (B) (1(2 3 4)5 6)7
(C) (1(2 3 4) (5 6 7)) (D) (1(2 3 NULL)(4 5))

15. A scheme for storing binary trees in an array X is as follows:

Indexing of X starts at 1 instead of 0. The root is stored at $X[\ 1\ ]$. For a node stored at $X[i]$, the left child, if any, is stored in $X[\ 2i\ ]$ and the right child, if any, in $X[2i + 1]$. To store any binary tree on 'n' vertices the minimum size of X should be

(A) $2n$ (B) n
(C) $3n$ (D) n^2

Practice Problems 2

Directions for questions 1 to 15: Select the correct alternative from the given choices.

1. A binary search tree contains the values 1, 2, 3, 4, 5, 6, 7, 8. The tree is traversed in pre-order and the values are printed out. Which of the following is a valid output?

(A) 53124786 (B) 53126487
(C) 53241678 (D) 53124768

2. A binary search tree is generated by inserting in order the following integers : 50, 15, 62, 5, 20, 58, 91, 3, 8, 37, 60, 24. The number of nodes in the left subtree and right subtree of the root respectively are:

(A) (4, 7) (B) (7, 4)
(C) (8, 3) (D) (3, 8)

3. A full binary tree (with root at level 0) of height h has a total number of nodes equal to:

(A) 2^h (B) $2^{h+1} - 1$
(C) $2^h - 1$ (D) 2^{h-1}

4. The number of null pointers of a binary tree of n nodes is :

(A) $n + 1$ (B) $n(n + 1)$
(C) n^2 (D) $2n$

5. Which of the following is false?

(A) A tree with n nodes has $(n - 1)$ edges.
(B) A labeled rooted binary tree can be uniquely constructed, given its post-order, in-order traversal results.
(C) The complete binary tree with n internal nodes has $(n + 1)$ leaves.
(D) The maximum number of nodes in a binary tree of height h is $(2^{h+1} - 1)$.

6. The maximum number of nodes in a binary tree at level i is

(A) 2^i (B) $2^i - 1$
(C) $2^i + 1$ (D) $\log_2 i + 1$

7. The number of leaf nodes in a rooted tree of n nodes, with each node having 0 or 3 children is

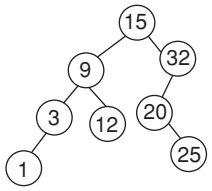
(A) $\frac{n}{3}$ (B) $\frac{(n-1)}{3}$
(C) $\frac{(n-1)}{2}$ (D) $\frac{(2n+1)}{3}$

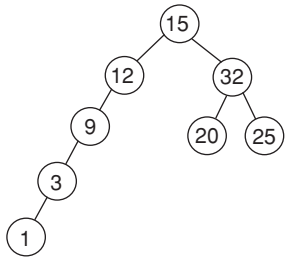
8. A complete n-ary tree is one in which every node has 0 or n children. If x is the number of internal nodes of a complete n-ary tree, the number of leaves in it is given by

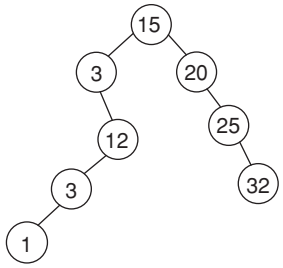
(A) $x(n - 1) + 1$
(B) $xn + 1$
(C) $xn - 1$
(D) $x(n + 1) - 1$

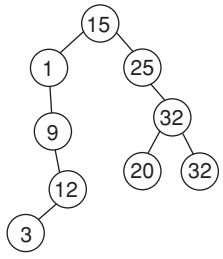
Common data for questions 9 and 10:

9. Insert the keys into a binary search tree in the order specified 15, 32, 20, 9, 3, 25, 12, 1. Which one of the following is the binary search tree after insertion of all elements?

(A)


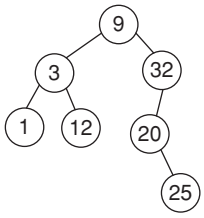
(B)


(C)


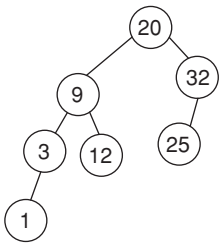
(D)


10. Which of the following is the binary tree after deleting 15?

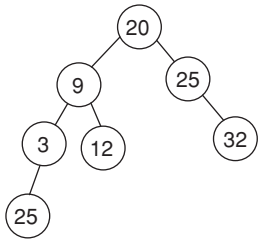
(A)



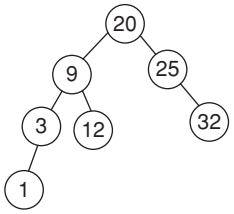
(B)



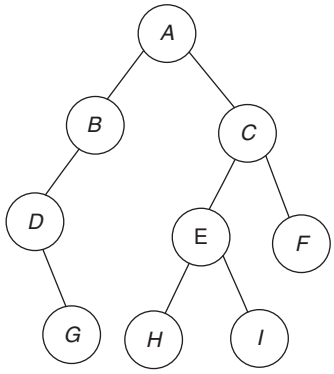
(C)



(D)



For questions 11, 12 and 13 below, use this figure



11. What is the post-order expression?
(A) ABDGCEJHIF (B) GDBHIEFCA
(C) DGBAHEICF (D) ABHIEFCDG

12. What is the pre-order expression?
(A) ABDGCEHIF (B) ABHIEFCDG
(C) DGBAHEIFCF (D) GDBHIEFCA

13. What is the in-order expression?
(A) ABDGCEHIF (B) GDBHIEFCA
(C) DGBAHEICF (D) ABHIEFCDG

14. In a 3-ary tree every internal node has exactly 3 children. The number of leaf nodes in such a tree with 6 internal nodes will be
(A) 13 (B) 12 (C) 11 (D) 10

15. Minimum number of swaps needed to convert the array 89, 19, 14, 40, 17, 12, 10, 2, 5, 7, 11, 6, 9, 70 into a max heap
(A) 2 (B) 3 (C) 1 (D) 0

Previous Years' Questions

1. In a binary tree with n nodes, every node has an odd number of descendants. Every node is considered to be its own descendant. What is the number of nodes in the tree that have exactly one child? **[2010]**
(A) 0 (B) 1
(C) $(n-1)/2$ (D) $n-1$

2. The following C function takes a singly-linked 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.

```
typedef 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;
}
```

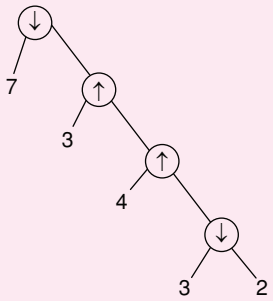
Choose the correct alternative to replace the blank line. **[2010]**
(A) `q = NULL; p->next = head; head = p;`
(B) `q->next = NULL; head = p; p->next = head;`
(C) `head = p; p->next = q; q->next = NULL;`
(D) `q->next = NULL; p->next = head; head = p;`

3. Consider two binary operators '↑' and '↓' with the

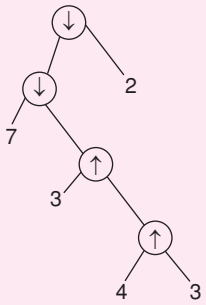
precedence of operator ↓ being lower than that of the operator ↑. Operator ↑ is right associative while operator ↓ is left associative. Which one of the following represents the parse tree for expression (7 ↓ 3 ↑ 4 ↑ 3 ↓ 2)?

[2011]

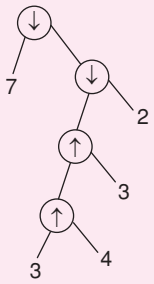
(A)



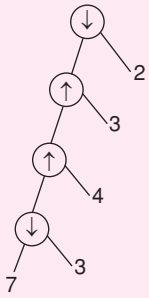
(B)



(C)



(D)



4. The height of a tree is defined as the number of edges on the longest path in the tree. The function shown in the pseudocode below is invoked as "height(root)" to compute the height of a binary tree rooted at the tree pointer "root".

```
int height(treeptr n)
{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 2
          }
    }
}
```

The appropriate expressions for the two boxes B_1 and B_2 are **[2012]**

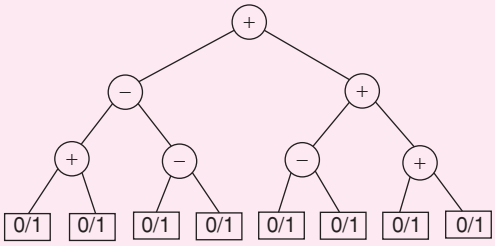
(A) B_1: (1 + height ($n \rightarrow$ right))
B_2: (1 + max (h_1, h_2))

(B) B_1: (height ($n \rightarrow$ right))
B_2: (1+ max(h_1, h_2))

(C) B_1: height ($n \rightarrow$ right)
B_2: max(h_1, h_2)

(D) B_1: (1 + height ($n \rightarrow$ right))
B_2: max(h_1, h_2)

5. 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 expression represented by the tree is ————. **[2014]**



6. Consider the pseudocode given below. The function **Dosomething()** takes as argument a pointer to the root of an arbitrary tree represented by the *leftMostChild-rightSibling* representation. Each node of the tree is of type **treenode.** **[2014]**

```
type def struct treeNode* treeptr;
struct treeNode
{
treeptr leftMostChild, rightSibling;
};
int Dosomething (treeptr tree)
```

```
{
int value = 0;
if (tree ! = NULL){
if (tree - > leftMostChild = = NULL)
value = 1;
else
value = Dosomething (tree - > leftMostChild);
value = value +Dosomething (tree - > right
Sibling);
}
return (value);
}
```

When the pointer to the root of a tree is passed as the argument to **DoSomething**, the value returned by the function corresponds to the

(A) Number of internal nodes in the tree
(B) Height of the tree
(C) Number of nodes without a right sibling in the tree
(D) Number of leaf nodes in the tree

7. The height of a tree is the length of the longest root-to-leaf path in it. The maximum and minimum number of nodes in a binary tree of height 5 are **[2015]**

(A) 63 and 6, respectively
(B) 64 and 5, respectively
(C) 32 and 6, respectively
(D) 31 and 5, respectively

8. Which of the following is/are correct inorder traversal sequence(s) of binary search tree(s)? **[2015]**

I. 3, 5, 7, 8, 15, 19, 25
II. 5, 8, 9, 12, 10, 15, 25
III. 2, 7, 10, 8, 14, 16, 20
IV. 4, 6, 7, 9, 18, 20, 25

(A) I and IV only (B) II and III only
(C) II and IV only (D) II only

9. Consider a max heap, represented by the array: 40, 30, 20, 10, 15, 16, 17, 8, 4 **[2015]**

Array Index	1	2	3	4	5	6	7	8	9
Value	40	30	20	10	15	16	17	8	4

Now consider that a value 35 is inserted into this heap. After insertion, the new heap is

(A) 40, 30, 20, 10, 15, 16, 17, 8, 4, 35
(B) 40, 35, 20, 10, 30, 16, 17, 8, 4, 15
(C) 40, 30, 20, 10, 35, 16, 17, 8, 4, 15
(D) 40, 35, 20, 10, 15, 16, 17, 8, 4, 30

10. A binary tree T has 20 leaves. The number of nodes in T having two children is _____ **[2015]**

11. Consider a binary tree T that has 200 leaf nodes. Then, the number of nodes in T that have exactly two children are _____. **[2015]**

12. While inserting the elements 71, 65, 84, 69, 67, 83 in an empty binary search tree (BST) in the sequence shown, the element in the lowest level is **[2015]**

(A) 65 (B) 67
(C) 69 (D) 83

13. Consider the following New–order strategy for traversing a binary tree: **[2016]**

- Visit the root;
- Visit the right subtree using New – order;
- Visit the left subtree using New – order;

The New – order traversal of the expression tree corresponding to the reverse polish expression

3 4 * 5 – 2 ^ 6 7 * 1 + – is given by:

(A) + – 1 6 7 * 2 ^ 5 – 3 4 *
(B) – + 1 * 6 7 ^ 2 – 5 * 3 4
(C) – + 1 * 7 6 ^ 2 – 5 * 4 3
(D) 1 7 6 * + 2 5 4 3 * – ∧ –

14. Let T be a binary search tree with 15 nodes. The minimum and maximum possible heights of T are: **[2017]**

Note: The height of a tree with a single node is 0.

(A) 4 and 15 respectively
(B) 3 and 14 respectively
(C) 4 and 14 respectively
(D) 3 and 15 respectively

15. The pre-order traversal of a binary search tree is given by 12,8,6,2,7,9,10,16,15,19,17,20. Then the post-order traversal of this tree is: **[2017]**

(A) 2, 6, 7, 8, 9, 10, 12, 15, 16, 17, 19, 20
(B) 2, 7, 6, 10, 9, 8, 15, 17, 20, 19, 16, 12
(C) 7, 2, 6, 8, 9,10, 20, 17, 19, 15, 16, 12
(D) 7, 6, 2, 10, 9, 8, 15, 16, 17, 20, 19, 12

16. The postorder traversal of a binary tree is 8, 9, 6, 7, 4, 5, 2, 3, 1. The inorder traversal of the same tree is 8, 6, 9, 4, 7, 2, 5, 1, 3. The height of a tree is the length of the longest path from the root to any leaf. The height of the binary tree above is _____. **[2018]**

17. The number of possible min-heaps containing each value from {1, 2, 3, 4, 5, 6, 7} exactly once is _____. **[2018]**

Answer Keys

Exercises

Practice Problems 1

1. B	**2.** B	**3.** A	**4.** C	**5.** D	**6.** C	**7.** C	**8.** B	**9.** B	**10.** B
11. B	**12.** D	**13.** A	**14.** C	**15.** A					

Practice Problems 2

1. D	**2.** B	**3.** B	**4.** A	**5.** C	**6.** B	**7.** D	**8.** A	**9.** A	**10.** B
11. B	**12.** A	**13.** C	**14.** A	**15.** B					

Previous Years' Questions

1. A	**2.** D	**3.** B	**4.** A	**5.** 6	**6.** D	**7.** A	**8.** A	**9.** B	**10.** 19
11. 199	**12.** B	**13.** C	**14.** B	**15.** B	**16.** 4	**17.** 80			

Test

Programming and Data Structure (Part 1)

Time: 60 min.

Directions for questions 1 to 30: Select the correct alternative from the given choices.

1. What is the behavior of following code?

```
auto int I;
int main( )
{
}
```

(A) Compiler error (B) No error
(C) Linker error (D) Runtime error

2. What is the output of following code:

```
#define scanf "%S is a string"
int main( )
{
printf(scanf, scanf);
}
```

(A) is a string is a string
(B) %S is a string is a string
(C) %S is a string %S is a string
(D) Syntax error

3.
```
void rec_fun (int n, int sum)
{
int k = 0, j = 0;
if(n = = 0) return;
k = n%10; j = n/10;
sum+ = k;
rec _fun(j, sum);
printf("%d\t", k);
}
main ( )
{
int a = 2048; sum = 0;
rec _fun(a, sum);
printf("%d", sum);
}
```

What does the above program print?

(A) 8 4 0 2 14
(B) 8 4 0 2 0
(C) 2 0 4 8 18
(D) 2 0 4 8 0

4.
```
int fun (int * p, int n)
{
if (n < = 0) return 0;
else
if(*P % 2 = = 0)
return *p + fun(p + 1, n - 1);
else
return *p - fun(p + 1, n -1);
}
main( )
{
int arr[ ] = {56, 48, 55, 10, 49, 14};
printf("% d", fun(arr, 6));
}
```

Which of the following is the output of above function?

(A) 110 (B) –122
(C) 114 (D) 108

5. Consider the function:

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

The value returned by $f(3)$ is

(A) 6 (B) 7
(C) 8 (D) 9

6. Which of the following code will change a lower case letter to an upper case?

(A) char $C_2 = (C_1 >= 'A' \,\&\, C_1 < 'Z')?$
$'a' + 'C_1' - 'A' : C_1);$

(B) char $C_2 = (C_1 >= 'a' \,\&\, C_1 <= 'z')?$
$'A' - 'a' + 'C_1') : C_1;$

(C) char $C_2 = (C_1 >= 'a' \,\&\, \& \, C_1 <= 'z')?$
$'A' + 'C_1' - 'a' : C_1;$

(D) char $C_2 = (C_1 >= 'A' \,\&\, \& \, C_1 <= 'Z')?$
$'A' - 'C_1' + 'a' : C_1;$

7. The following is a program to find the average length of several lines of text. What should be the lines of code corresponding to 'SecA' and 'SecB'.

```
main( )
{int n, count = 0, sum = 0;
float avg;
secA
```

```
{
count++;
sum + = n;
}
avg = (float) sum/count;
}
int linecount (void)
{
char line [80];
int count = 0;
while (secB)
{
count ++;
}
return (count);
}
```

(A) Sec A: while ($n > 0$)
Sec B: line [count]! = 0
(B) Sec A: while (linecount ( ) > 0)
Sec B: (line [count] = getch ( ) ! = '\n')
(C) Sec A: while ((n = linecount( ) > 0)
Sec B: (line [count] = getch ( ) ! = '\n')
(D) None of these

8. What is the meaning of following declaration? int($*f_1$) ( );
(A) f_1 is a function which returns a pointer to an integer number.
(B) f_1 is a pointer to a function which returns an integer number.
(C) f_1 is a function which takes an integer pointer.
(D) f_1 is a pointer to an integer number.

9. Which of the following represents the statement: "x is a pointer to a group of one dimensional 20 element arrays".
(A) int $*x$[20]; (B) int $*x$[10] [20];
(C) int $**x$[20]; (D) int ($*x$) [20];

10. What will be the output of the following code?

```
int number[ ] = {18, 20, 22, 24};
main( )
{
int *q;
q = number;
q + = 4;
printf("%d",*q);
}
```

(A) 24 (B) Compiler error
(C) syntax error (D) –24

11.

```
main( )
{
char fname[ ] = "TIME 4 EDUCATION";
time4(fname);
}
time4(char fname[5])
{
fname + = 7;
printf("%s", fname);
fname = 2;
printf("%s", fname);
}
```

What is the output of above code?
(A) 4 EDUCATION
(B) EDUCATION
(C) TIME 4 EDUCATION
(D) EDUCATION ME 4 EDUCATION

12. What will be output of following code?

```
int main( )
{
extern int x;
x = 13;
printf ("%d", x);
return 0;
}
```

(A) 13
(B) Vary from compiler
(C) Linker error
(D) Undefined symbol

13. What is the output of the following code?

```
main( )
{
int x = y = z = 100;
int i;
i = x > y < z;
printf("% d", i);
}
```

(A) 0 (B) 1
(C) error (D) No output

14. In the following code, how many times 'while' loop will be executed?

```
int count = 0;
while (count < 32767)
count++;
```

(A) 32767
(B) 32766
(C) infinite times
(D) varies from complier to compiler

15. #define SUM(*x*)*x* + *x* * *x*

#define DIF(*x*)*x* * *x* – *x*

```
int main( )
{
float y = SUM(5) /DIF(5);
printf("%f", y);
}
```

(A) 1.5 (B) 0
(C) 2 (D) 1

16. Which of the following is a correct description of void (* ptr[10]) ( ) ;

(A) ptr is an array of 10 pointers to functions returning type void.
(B) ptr is an array of 10 functions returning pointers of type void.
(C) ptr is an array of 10 functions returning void *.
(D) ptr is an array of data elements of type void.

17.
```
int main( )
{
int p = 2, g = 2;
printf(" %d%d", p < < g, p > > g);
}
```

Output of above code is

(A) 1 16 (B) 4 0
(C) 16 4 (D) 8 0

18. Which of the following is equivalent expression for *P* = *a* * 16 + b/8;

(A) $P = (a << 4) + (b >> 2)$
(B) $P = (a >> 4) - (b << 2)$
(C) $P = (a << 4) + (b >> 3)$
(D) $P = (a << 4) + (b << 3)$

19. What is the output of the below code?

```
main( )
{
int I = 0, *j = &I;
f1(j);
*j = *j + 10;
printf(" %d%d", I, j);
}
f1(int *k)
{
*k+ = 15;
}
```

(A) 20 55 (B) 25 25
(C) 45 55 (D) 35 35

20.
```
int rec_f2(int r)
{
if (r = = 1 || r= = 0)
return 1;
if (r % 2 = = 0)
return(rec_f2 (r/2)+ 2);
else return ((rec_f2(r – 1) + 3);
}
main( )
{
printf ("%d", rec_f2(7));
}
```

Which of the following is the output of above program?

(A) 10 (B) 11
(C) 13 (D) 0

21.
```
int guest(int a);
int host(int b);
main( )
{
int p = 50, q = 100, r ;
for (r = 0; r < 2; r ++)
{
q = guest(p) + host(p);
printf(" %d", q);
}
}
int guest (int a)
{
int y;
y = host(a);
return(y);
}
int host (int a)
{
static int y = 0;
y = y + 1;
return (a + y);
}
```

The output of above code will be

(A) 103 107 (B) 107 103
(C) 110 107 (D) 107 110

22. Choose the best matching between groups A and B:

	Group A		Group B
1	Volatile	P	Queue
2	Function pointer	Q	Auto
3	Default	R	Guest and host
4	FIFO	S	Switch–case

(A) 1 – S, 2 – Q, 3 – R, 4 – P
(B) 1 – Q, 2 – P, 3 – R, 4 – S

(C) 1 – Q, 2 – R, 3 – S, 4 – P
(D) 1 – R, 2 – Q, 3 – P, 4 – S

23. Which of the following expression represents the statement: "*P* is a function that accepts a pointer to a character array".
(A) int *p*(char *a[ ]); (B) int (*p) (char (*a)[ ]);
(C) int *p (char *a); (D) int *p*(char (*a) [ ]);

24.
```
void arr_fun(int [ ] [3]);
main( )
{
int x[3][3] = {{10, 20, 30}, {40, 50, 60},
{70, 80, 90}};
arr_fun(x);
printf("%d", x[2][1]);
}
void arr_fun (int y[ ][3])
{
++y;
y[1][1] = 9;
}
```
What is the output of the above code?
(A) 80 (B) 90
(C) 70 (D) None of these

25.
```
#define F – 1
#define T 1
#define N 0
main( )
{
if(N)
printf(" %s", "GOOD");
else
if(F)
printf(" %s", "MORNING");
else
printf(" %s", "GOOD NIGHT");
}
```
Output of above code will be
(A) GOOD
(B) MORNING
(C) GOOD MORNING
(D) GOOD NIGHT

26. An external variable
(i) is defined once and declared in other functions.
(ii) is globally accessible by all functions.
(iii) cannot be static
(iv) is defined after main( )
(A) (i) and (ii) (B) (i), (ii) and (iii)
(C) (ii), (iii) and (iv) (D) (i), (ii), (iii), (iv)

Common Data for Questions 27 and 28:

Consider the following code:
```
int f(int P)
{
if(P < = 0) return 1;
if(P% 10 = = 0)
return f(P – 2); //X
else
return f(P – 3); //Y
}
main( )
{
printf(" %d", f(30));
}
```

27. What will be the output of above code?
(A) 10 (B) 50
(C) 100 (D) 1

28. What will be the output if the lines labeled *X* and *Y* are changed as follows:
X: return 3 + *f*(*P*/2);
Y: retrun 2 + *f*(*P*/3);
(A) 4 (B) 6
(C) 8 (D) 10

Common data for Questions 29 and 30:

```
1. float fn1(float n, int a)
2. {
3. float P, S;
   int I;
4. for(S = x, P = 1, I = 1; P = P * x * x;
5. I < a; I ++)
   S = S + P/(l);
6. return (S);
7. }
8. int f(int x)
9. {
10. int f = 1;
11. for (int i = 1; i < = n; i++)
12. f = f * i;
13. return (f);
14. }
```

29. Output of above code for fn1(2.1, 20) is

(A) $x+\frac{x^2}{2!}+\frac{x^4}{4!}+...a+2.1$

(B) $\frac{x-x^3}{3!}+\cdot a+2.1$

(C) $1+\frac{x}{1!}+\frac{x^2}{2!}+$

(D) $1+\frac{x}{1!}+\frac{x^3}{3!}+\frac{x^5}{5!}+\cdot a+2.1$

30. When the statement numbered 4, 5, 6, 7 are replaced by

```
{for (S = 1, P = -x, I = 1; I < a; i++)
P = P * x * x -1;
S = S + P/f(I);
}
```

What will be the approximation of $f(x)$?

(A) $1+x-x^2+x^3\ldots$

(B) $1-\frac{x}{1!}+\frac{x^3}{3!}-$

(C) $1-\frac{x}{1!}+\frac{x^2}{2!}+\frac{x^3}{3!}+\cdots$

(D) $1-\frac{x}{1!}+\frac{x^2}{2!}-\frac{x^3}{3!}+\cdots$

Answer Keys

1. A	**2.** B	**3.** A	**4.** C	**5.** A	**6.** C	**7.** C	**8.** B	**9.** D	**10.** B
11. D	**12.** C	**13.** B	**14.** C	**15.** D	**16.** A	**17.** D	**18.** C	**19.** B	**20.** B
21. A	**22.** C	**23.** B	**24.** A	**25.** B	**26.** B	**27.** D	**28.** D	**29.** A	**30.** D