Q.2 a. Mention any five data types in C. Give their respective size in bits and range.

Answer:

a. Data types in C

Type	Typical Size in Bits	Minimal Range
char	8	–127 to 127
unsigned char	8	0 to 255
signed char	8	–127 to 127
int	16 or 32	–32,767 to 32,767
unsigned int	16 or 32	0 to 65,535
signed int	16 or 32	same as int
short int	16	–32,767 to 32,767
unsigned short int	16	0 to 65,535
signed short int	16	same as short int
long int	32	–2,147,483,647 to 2,147,483,647
signed long int	32	same as long int
unsigned long int	32	0 to 4,294,967,295
float	32	Six digits of precision
double	64	Ten digits of precision
long double	80	Ten digits of precision

b. Write short notes on type casting.

Answer:

b. **Typecasting**

Type casting is used when you want to convert the value of one data type to another.

1

Type casting does not change the actual value of the variable, but the resultant value may be put in temporary storage.

Type casting is done using a cast operator that is also a unary operator.

The unary operators are associated from right to left.

c. Write a program to illustrate the usage of any four bitwise operators and give the corresponding output.

Answer:

c.
```
# include<stdio.h>
main()
{   char c1,c2,c3;
  printf("ENTER VAULES OF c1 and c2");
  scanf("%c,%c",&c1,&c2);
  c3 = c1 & c2;     printf("\n Bitwise AND i.e. c1 & c2 = %c",c3);
  c3 = c1 | c2;     printf("\n Bitwise OR i.e. c1 | c2 = %c",c3);
  c3 = c1 ^ c2;     printf("\n Bitwise XOR i.e. c1 ^ c2 = %c",c3);
  c3 = ~c1;         printf("\n ones complement of c1 = %c",c3);
  c3 = c1<<2;       printf("\n left shift by 2 bits c1 << 2 = %c",c3);
  c3 = c1>>2;       printf("\n right shift by 2 bits c1 >> 2 = %c",c3);  }
```

d. Give the tabular format to indicate precedence of operators in C language.

Answer:

d. **Object-based programming** is a style of programming that primarily supports encapsulation and

Highest	( ) [ ] -> .
	! ~ ++ -- (type) * & sizeof
	* / %
	+ -
	<< >>
	< <= > >=
	== !=
	&
	^
	|
	&&
	||
Highest	
	?:
	= += -= *= /= etc.
Lowest	,

2x1

1 M

1 M

Q.3 a. Compare while, do-while and for loop statements in C programming language.

Answer:

3.

a. **While stmt**

Syntax of while loop is given as:

while(condition) statement;

where statement is either an empty statement, a single statement, or a block of statements. The condition may be any expression, and true is any nonzero value. The loop iterates while the condition is true. When the condition becomes false, program control passes to the line of code immediately following the loop.

Do-while Stmt

Unlike for and while loops, which test the loop condition at the top of the loop, the do-while loop checks its condition at the bottom of the loop. This means that a do-while loop always executes at least once. The general form of the do-while loop is

```
do {
statement;
} while(condition);
```

1 1/2 + 1

Although the curly braces are not necessary when only one statement is present, they are usually used to avoid confusion (to you, not the compiler) with the while. The do-while loop iterates until condition becomes false.

For Stmt

The general design of the for loop is reflected in some form or another in all procedural programming languages. The general form of the for statement is

for(initialization; condition; increment) statement;

The for loop allows many variations, but its most common form works like this. The initialization is an assignment statement that is used to set the loop control variable. The condition is a relational expression that determines when the loop exits. The increment defines how the loop control variable changes each time the loop is repeated. We must separate these three major sections by semicolons. The for loop continues to execute as long as the condition is true. Once the condition becomes false, program execution resumes on the statement following the for.

b.Mention the role of the following for printf() and scanf() statements:

(i) Type identifiers (ii) Field width
(iii) Precision (iv) Flags
(v) Escape Sequence

Answer:

b.

5x1

Type identifiers

d, i Signed integers
o Unsigned integers displayed in octal form.
u Unsigned integers in decimal form.
x Unsigned integers in hexadecimal form, and the hexadecimal characters a, b, c, d, e, and f printed in lowercase.

Field width

Field-width indicates the least number of columns that will be allocated to the output. For example, if you write %4d to i and the value of i is 10, then 4 columns are allocated for i and 2 blank are added on left side of value of i. So the output is bb10. Here, b indicates blank.

Precision

Precision indicates the minimum number of digits printed for type integers d, i, o, u, x, and X. For example,

```
printf("%10.4d\n", 35)
```

Flags

Flag characters are used to give directives for the output. You can use multiple flag characters in any order.

The flag characters are as follows: Indicates that output is left justified.

```
printf("%-10.4d\n", 25)
```

Escape Sequence

\b Backspace - Moves the cursor to the last column of the previous line.

\f Form feed - Moves the cursor to start of next page.

c. Explain the role of address and pointers in C language. Give an example of each for illustration.

Answer:

c. **Address**

3

For every variable declared in a program there is some memory allocation. Memory is specified in arrays of bytes, the size of which depending on the type of variable. For the integer type, 2 bytes are allocated, for floats, 4 bytes are allocated, etc. For every variable there are two attributes: address and value, described as follows:

```
#include <stdio.h>
main ()
{
  int i, j, k;     //A
     i = 10;    //B
  j = 20;          //C
  k = i + j; //D
  printf ("Value of k is %d\n", k);
}
```

Pointers

3

A pointer is a variable whose value is also an address. As described earlier, each variable has two attributes: address and value. A variable can take any value specified by its data type. For example, if the variable i is of the integer type, it can take any value permitted in the range specified by the integer data type. A pointer to an integer is a variable that can store the address of that integer.

```
#include <stdio.h>
main ()
{
  int i;        //A
  int * ia;     //B
  i = 10;       //C
  ia = &i;  //D

   printf (" The address of i is %8u \n", ia);         //E
   printf (" The value at that location is %d\n", i);   //F
   printf (" The value at that location is %d\n", *ia); //G
 *ia = 50;                          //H
   printf ("The value of i is %d\n", i);             //I
}
```

Q.4 a. Explain recursion with the help of an example.

Answer:

a. In C, a function can call itself. A function is said to be recursive if a statement in the body of the function calls itself. Recursion is the process of defining something in terms of itself, and is sometimes called circular definition. A simple example of a recursive function is factr( ), which computes the factorial of an integer. The factorial of a number n is the product of all the whole numbers between 1 and n. For example, 3 factorial is 1 x 2 x 3, or 6. Both factr( ) and its iterative equivalent are shown here:

```
/* recursive */
int factr(int n) {
int answer;
if(n==1) return(1);answer = factr(n-1)*n; /* recursive call */
return(answer); }
```

b. Explain the following with respect to functions and give an example for illustration: (i) call-by-value (ii) call-by-reference

Answer:

(A) In call by value method copies the value of an argument into the formal parameter of the subroutine. In this case, changes made to the parameter have no effect on the argument.

```
#include <stdio.h>
int sqr(int x);
int main(void) {
int t=10;
printf("%d %d", sqr(t), t);
return 0; }
int sqr(int x) {
x = x*x;
return(x); }
```

3M

(B) Call by reference is the second way of passing arguments to a subroutine. In this method, the address of an argument is copied into the parameter. Inside the subroutine, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

```
void swap(int *x, int *y) {
int temp;
temp = *x; /* save the value at address x */
*x = *y; /* put y into x */
*y = temp; /* put x into y */ }
int main(void) {
int i, j;
i = 10; j = 20;
printf("%d %d", i, j);
swap(&i, &j); /* pass the addresses of i and j */
printf("%d %d", i, j);
return 0; }
```

3M

c. How can array elements be accessed using pointers? Give an illustration.

Answer:

c. Two methods of accessing array elements: pointer arithmetic and array indexing. Although the standard array-indexing notation is sometimes easier to understand, pointer arithmetic can be faster. Since speed is often a consideration in programming.

```
/* Access s as a pointer. */
void putstr(char *s) {
while(*s) putchar(*s++); }
```

2 +2 =

Q.5 a. Using array of strings, write a program to display strings January to December.

b. List any four file operations.

c. Consider a structure Student with data members - char name[20] and float tmarks. Write a program to read and display the values of data members: name and tmarks using pointer to student structure.

Answer:

a.
```
#include <stdio.h>
int main() {
const int MONTHS = 12; //number of strings in array
const int MAX = 10; //maximum size of each string
//array of strings
char star[MONTHS][MAX] = { "January", "Februaryy", "March"......
                "Decembery"};
for(int j=0; j<MONTHS; j++) //display every string
cout << star[j] << endl;
return 0; }
```

b.

fopen( )	Opens a file.
fclose( )	Closes a file.
putc( )	Writes a character to a file.
fputc( )	Same as putc( ).
getc( )	Reads a character from a file.
fgetc( )	Same as getc( ).
fgets( )	Reads a string from a file.
fputs( )	Writes a string to a file.
fseek( )	Seeks to a specified byte in a file.
ftell( )	Returns the current file position.

c.
```
struct student      \\ A
{
   char name[30];   \\ B
   float marks;     \\ C
};           \\ D

main ( )
{
   struct student *student1;      \\ E
   struct student student2; \\ F
   char s1[30];
   float f;
   student1 = &student2;    \\ G
   scanf ("%s", name);         \\ H
   scanf (" %f", & f);      \\ I
   *student1.name = s1;        \\ J student1-> name = f;
   *student2.marks = f;        \\ K student1-> marks = s1;

   printf (" Name is %s \n", *student1.name);    \\ L
   printf (" Marks are %f \n", *student2.marks); \\ M
}
```

Q.6 a. Give the worst case and average case complexities for the following:
(i) Quick sort (ii) Merge sort
(iii) Heap sort

b. Write a C program to merge two sorted lists.

c. Define Binary Search Tree. Give an example for illustration.

Answer:

6.

a.

Sorting Algorithm	Worst case	Average case
Quick sort	O(n^2)	O(n logn)
Merge sort	O(n logn)	O(n logn)
Heap sort	O(n logn)	O(n logn)

2.

b.

```
// merge two sorted list
#include<stdio.h>
#include<conio.h>
void main() {
  void read(int *,int);    void dis(int *,int);    void sort(int *,int);
  void merge(int *,int *,int *,int);
  int a[5],b[5],c[10];
  clrscr();
  printf("Enter the elements of first list \n");   read(a,5);     /*read the list*/
  printf("The elements of first list are \n");     dis(a,5);  /*Display the first list*/
  printf("Enter the elements of second list \n");    read(b,5);  /*read the list*/
  printf("The elements of second list are \n");   dis(b,5);  /*Display the second list*/
  sort(a,5);      printf("The sorted list a is:\n");
  dis(a,5);   sort(b,5);   printf("The sorted list b is:\n");
  dis(b,5);

  merge(a,b,c,5);
  printf("The elements of merged list are \n");
  dis(c,10); /*Display the merged list*/   getch(); }
void read(int c[],int i) {
  int j;
  for(j=0;j<i;j++)
    scanf("%d",&c[j]);
  fflush(stdin); }

void dis(int d[],int i) {
  int j;
  for(j=0;j<i;j++)
    printf("%d ",d[j]);   printf("\n"); }
void sort(int arr[] ,int k) {
  int temp;
  int i,j;
  for(i=0;i<k;i++)    {
    for(j=0;j<k-i-1;j++)      {
      if(arr[j]<arr[j+1])
      {
        temp=arr[j];         arr[j]=arr[j+1];          arr[j+1]=temp;        }    }  }}
void merge(int a[],int b[],int c[],int k) {
  int ptra=0,ptrb=0,ptrc=0;
  while(ptra<k && ptrb<k)    {
    if(a[ptra] < b[ptrb])      {
        c[ptrc]=a[ptra];
      ptra++;
    }
```

3

```
    Else      {
        c[ptrc]=b[ptrb];
      ptrb++;
    }
    ptrc++;
  }
  while(ptra<k)   {
    c[ptrc]=a[ptra];
    ptra++;ptrc++;
  }
  while(ptrb<k)   {
    c[ptrc]=b[ptrb];
    ptrb++;  ptrc++;
  }}
```

c.

A *binary search tree* is a binary tree in which the nodes are labeled with elements of a set. The important property of a binary search tree is that all elements stored in the left subtree of any node x are all less than the element stored at x, and all elements stored in the right subtree of x are greater than the element stored at x. This condition, called the *binary search tree property*, holds for every node of a binary search tree, including the root.

Example

Q.7 a. Mention the applications of stacks and queues.

Answer:

a. **Stack**

One of the applications of the stack is in expression evaluation. A complex assignment statement such as a = b + c*d/e–f may be interpreted in many different ways. Therefore, to give a unique meaning, the precedence and associativity rules are used. But still it is difficult to evaluate an expression by computer in its present form, called the infix notation. In infix notation, the binary operator comes in between the operands. A unary operator comes before the operand. To get it evaluated, it is first converted to the postfix form, where the operator comes after the operands. For example, the postfix form for the expression a*(b–c)/d is abc–*d/. A good thing about postfix expressions is that they do not require any precedence rules or parentheses for unique definition. So, evaluation of a postfix expression is possible using a stack-based algorithm.

Queues

One application of the queue data structure is in the implementation of priority queues required to be maintained by the scheduler of an operating system. It is a queue in which each element has a priority value and the elements are required to be inserted in the queue in decreasing order of priority. This requires a change in the function that is used for insertion of an element into the queue. No change is required in the delete function.

b. Write a C program to illustrate the following operations in doubly linked list:
   (i) Insert a new value after the specified value
   (ii) Delete a new value after the specified value

Answer:

(A) Insert a new value after the specified value

```
/* a function which inserts a newly created node after the specified
   node in a doubly linked list */

struct node * newinsert ( struct dnode *p, int node_no, int value )     {
    struct dnode *temp, * temp1;
    int i;
    if ( node_no <= 0 || node_no > nodecount (p))       {
    printf("Error! the specified node does not exist\n");
    exit(0);
    }
    if ( node_no == 0)      {
    temp = ( struct dnode * )malloc ( sizeof ( struct dnode ));
    if ( temp == NULL )      {
    printf( " Cannot allocate \n");
    exit (0);
    }
    temp -> data = value;      temp -> right = p;      temp->left = NULL
    p = temp ;
    }
    Else     {
    temp = p ;     i = 1;     while ( i < node_no )      {
    i = i+1;
    temp = temp-> right ;
    }
    temp1 = ( struct dnode * )malloc ( sizeof(struct dnode));
    if ( temp == NULL )      {
    printf("Cannot allocate \n");
    exit(0);
    }
    temp1 -> data = value ;
    temp1 -> right = temp -> right;
    temp1 -> left = temp;
    temp1->right->left = temp1;
    temp1->left->right = temp1
    }
    return (p);
}
```

(B)

```
/* a function which delets a newly created node after the specified
   node in a doubly linked list */
  struct dnode * delete( struct dnode *p, int node_no, int *val)    {
  struct dnode *temp ,*prev=NULL;
  int i;
  if ( node_no <= 0 || node_no > nodecount (p))     {
  printf("Error! the specified node does not exist\n");
  exit(0);
  }
  if ( node_no == 0)    {
   temp = p;
   p = temp->right;
   p->left = NULL;

    *val = temp->data;
    return(p);
   }
   Else     {
   temp = p ;
    i = 1;
   while ( i < node_no )     {
   i = i+1;
    prev = temp;
   temp = temp-> right ;
   }
    prev->right = temp->right;
    if(temp->right != NULL)
     temp->right->left = prev;
    *val = temp->data;
    free(temp);
   }
     return (p);
   }
```

c. Give the polynomial representation of linked list.

Answer:

c. One of the problems that a linked list can deal with is manipulation of symbolic polynomials. By symbolic, we mean that a polynomial is viewed as a list of coefficients and exponents. For example, the polynomial $3x^2+2x+4$,

can be viewed as list of the following pairs (3,2),(2,1),(4,0)
Therefore, we can use a linked list in which each node will have three fields, as shown below:

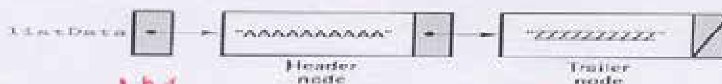
A polynomial $10x^4 + 5x^2 + 1$ can be represented as shown here:

d. Give an example to illustrate empty linked list with header and trail nodes.

Answer:

d.

Empty Linked list with Header and trailer:



Q.8 a. Write a program to search for a target key in a binary search tree.

Answer:

a.

```
/* A function to serch for a given data value in a binary search tree*/
struct tnode *search( struct tnode *p,int key)    {
    struct tnode *temp;
     temp = p;
    while( temp != NULL)        {
       if(temp->data == key)
            return(temp);
       else
       if(temp->data > key)
          temp = temp->lchild;
       else
          temp = temp->rchild;
          }
    return(NULL); }
```

b. Write sequence of steps for the following tree traversals:
(i) Preorder (ii) Inorder
(iii) Postorder

Answer:

b. Preorder
1. Process the root R
2. Traverse the left subtree of R in preorder
3. Traverse the right subtree of R in preorder

Inorder
1. Traverse the left subtree of R in inorder
2. Process the root R
3. Traverse the right subtree of R in ineorder

Postorder
1. Traverse the left subtree of R in postorder
2. Traverse the right subtree of R in postorder
3. Process the root R

c. Give the Big O comparisons for binary search tree and linked list for the following operations:
(i) FindElement() (ii) MakeEmpty()

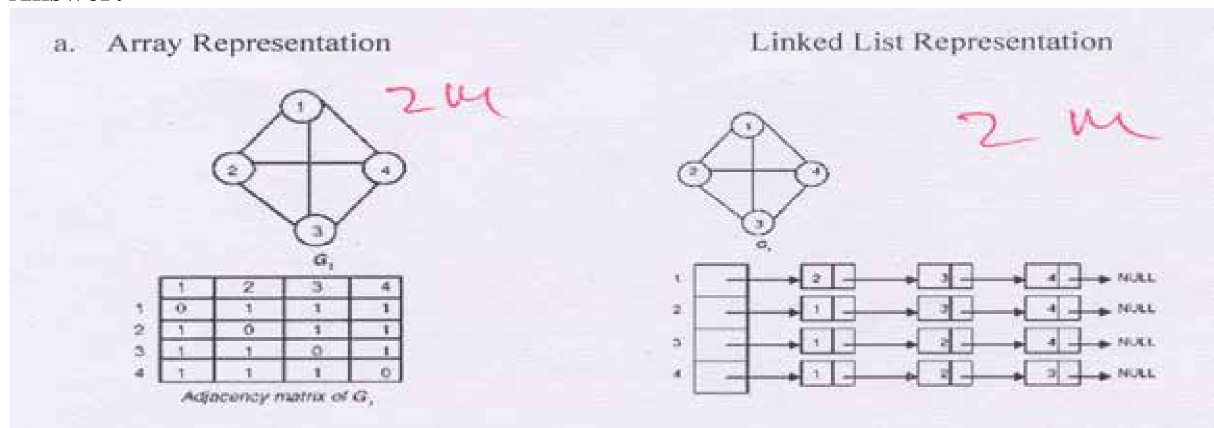
Answer:

c.

	FindElement	MakeEmpty
Binary Search tree	O(logn)	O(n)
Linked List	O(n)	O(n)

Q.9 a. Give an example to illustrate array and linked list representation of graphs.

Answer:



	1	2	3	4
1	0	1	1	1
2	1	0	1	1
3	1	1	0	1
4	1	1	1	0

b. Write the algorithm for depth first search (DFS) and give its analysis.

Answer:

b. **Algorithm DFS**

1. Input the vertices and edges of the graph G = (V, E).
2. Input the source vertex and assign it to the variable S.
3. Push the source vertex to the stack.
4. Repeat the steps 5 and 6 until the stack is empty.
5. Pop the top element of the stack and display it.
6. Push the vertices which is neighbor to just popped element, if it is not in the queue and displayed (ie; not visited).
7. Exit.

Analysis

In case G is represented by its adjacency lists then the vertices w adjacent to v can be determined by following a chain of links. Since the algorithm DFS would examine each node in the adjacency lists at most once and there are 2e list nodes, the time to complete the search is O(e). If G is represented by its adjacency matrix, then the time to determine all vertices adjacent to v is O(n). Since at most n vertices are visited, the total time is O(n2).

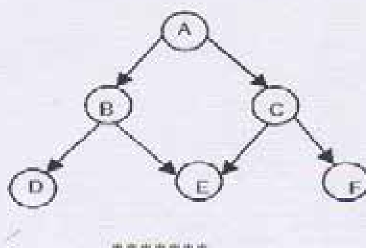
c. Explain direct acyclic graph with an example.

Answer:

c. Directed Acyclic Graphs

A directed acyclic graph, or dag for short, is a directed graph with no cycles. Measured in terms of the relationships they can represent, dags are more general than trees but less general than arbitrary directed graphs. DAGs are useful in representing the syntactic structure of arithmetic expressions with common sub-expressions. For example, consider the following expression:

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



****

2 M

2x2 =

Text Book

C & Data Structures, P.S. Deshpande and O.G. Kakde, Dreamtech Press, 2005