Q.2 a. Given the value of C1 = 4 and C2 = 6. What will be the value of C3 for each of the following expression?

(i) C3 = C1 & C2 **(ii) C1 << 2**

(iii) C1 ^ C2 **(iv) C1 | C2** **(4)**

Answer:

(i) 4 (ii) 16

(iii) 2 (iv) 6

b. Write a C language program to find whether a given number is palindrome or not. (6)

Answer:

```
#include <stdio.h>
   int main()
   {
    int n, reverse=0, rem,temp;
    printf("Enter an integer: ");
    scanf("%d", &n);
    temp=n;
    while(temp!=0)
    {
      rem=temp%10;
      reverse=reverse*10+rem;
      temp/=10;
    }
   /* Checking if number entered by user and it's reverse number is equal. */
    if(reverse==n)
       printf("%d is a palindrome.",n);
    else
       printf("%d is not a palindrome.",n);
    return 0;
   }
```

c. What is data type? Explain any four data types used in C language. (6)

Answer:

C has a concept of 'data types' which are used to define a variable before its use. The definition of a variable will assign storage for the variable and define the type of data that will be held in the location.

The value of a variable can be changed any time.

C has the following basic built-in datatypes.

- int
- float
- double
- char

Q.3 a. Differentiate between relational and logical operators used in C. (4)

Answer:

Logical operators don't Compare values they combine Boolean values and produce a Boolean result. Examples of logical operators are && (and), ||, (or), ! (not). If you have two Boolean values and you combined them with the && operator the result will be (TRUE) only if both values were (TRUE). If you combined them with the || operator the result will be TRUE if any of them or both of them were TRUE. Relational operators compare two values and produce a Boolean result. Most of the time we use logical operators to combine the results of two or more comparison expressions that use relational operators. For instance we may say:

If(a > b && a <= c) { do something}

This means that if both expressions a>b AND a <= c are TRUE the result will be true and the body of the if statement will be executed.

b. Compare and contrast the following:

(i) *While* and *For* loop

(ii) *Break* and *Continue* statement

Use suitable examples. (8)

Answer:

Introduction

The while loop is used when you want to repeat the execution of a certain statement or a set of statements (compound statement).

Program/Example

The general format for a while loop is

```
while (condition)
simple or compound statement (body of the loop)
```

For example,

```
i = 0;
while (i<5)
{
    printf(" the value of i is %d\n", i);
    i = i + 1;
}
```

Explanation

1. Before entering into the loop, the while condition is evaluated. If it is true then only the loop body is executed.

2. Before making an iteration, the `while` condition is checked. If it is true then the loop body is executed.
3. It is the responsibility of the programmer to ensure that the condition is false after certain iterations; otherwise, the loop will make infinite iterations and it will not terminate.
4. The programmer should be aware of the final value of the looping variable. For example, in this case, the final value of the looping variable is 5.
5. While writing the loop body, you have to be careful to decide whether the loop variable is updated at the start of the body or at the end of the body.

THE do-while LOOP

Introduction

The `do-while` loop is used when you want to execute the loop body at least once. The `do-while` loop executes the loop body and then traces the condition.

Program/Example

The general format for a `do-while` loop is

```
do
    simple or compound statement
    while (condition)
```

For example,

```
i = 0;
do
{
    printf(" the value of i is %d\n", i);
    i = i + 1;
}
while (i<5)
```

Explanation

1. The loop body is executed at least once.
2. The condition is checked after executing the loop body once.

3. If the condition is false then the loop is terminated.
4. In this example, the last value of i is printed as 5.

THE for LOOP

Introduction

The for loop is used only when the number of iterations is predetermined, for example, 10 iterations or 100 iterations.

Program/Example

The general format for the for loop is

```
for (initializing; continuation condition; update)
simple or compound statement
```

For example,

```
for (i = 0; i < 5; i++)
{
    printf("value of i");
}
```

Explanation

1. The for loop has four components; three are given in parentheses and one in the loop body.
2. All three components between the parentheses are optional.
3. The initialization part is executed first and only once.
4. The condition is evaluated before the loop body is executed. If the condition is false then the loop body is not executed.
5. The update part is executed only after the loop body is executed and is generally used for updating the loop variables.
6. The absence of a condition is taken as true.
7. It is the responsibility of the programmer to make sure the condition is false after certain iterations.

THE for LOOP WITH A COMMA OPERATOR

Introduction

You may want to control the loop variables in the same for loop. You can use one for loop with a comma operator in such situations.

Program/Example

```
for (i = 0, j = 10; i < 3 && j > 8; i++, j--)
printf (" the value of i and j %d %d\n",i, j);
```

Explanation

1. First i is initialized to 0, and j is initialized to 10.
2. The conditions i<3 and j>8 are evaluated and the result is printed only if both conditions are true.
3. After executing the loop body, i is incremented by 1 and j is decremented by 1.
4. The comma operator also returns a value. It returns the value of the rightmost operand. The value of (i = 0, j = 10) is 10.

THE break STATEMENT

Introduction

Just like the switch statement, break is used to break any type of loop. Breaking a loop means terminating it. A break terminates the loop in which the loop body is written.

Program/Example

For example,

```
i = 0;
while (1)
{
```

```
    i = i + 1;
    printf(" the value of i is %d\n");
    if (i>5) break;
}
```

Explanation

1. The `while (1)` here means the `while` condition is always true.
2. When `i` reaches 6, the `if` condition becomes true and `break` is executed, which terminates the loop.

THE continue STATEMENT

Introduction

The break statement breaks the entire loop, but a `continue` statement breaks the current iteration. After a `continue` statement, the control returns to top of the loop, that is, to the test conditions. `Switch` doesn't have a `continue` statement.

Program/Example

Suppose you want to print numbers 1 to 10 except 4 and 7. You can write:

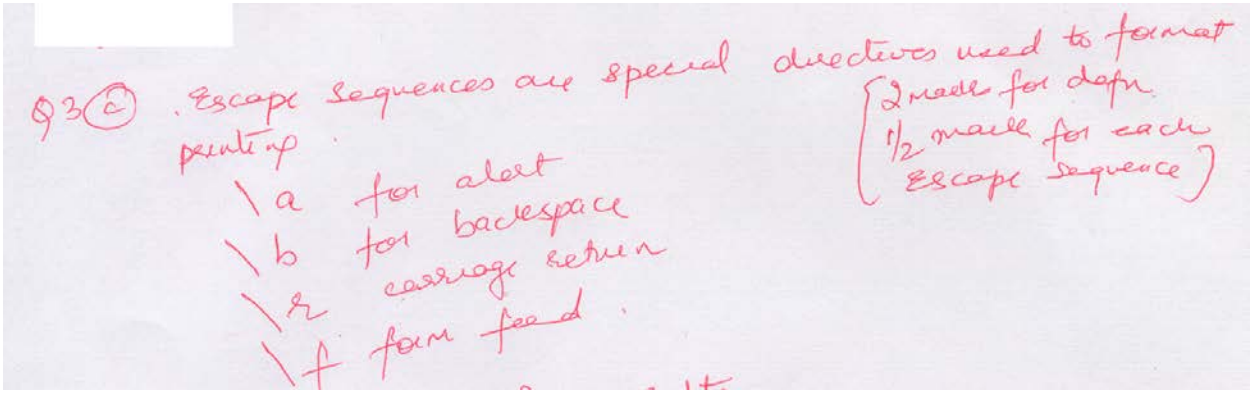
```
for(i = 0, i < 11, i++)
{
   if ((i == 4) || (i == 7)) continue;
   printf(" the value of i is %d\n", i);
}
```

Explanation

1. If `i` is 1 then the `if` condition is not satisfied and `continue` is not executed. The value of `i` is printed as 1.
2. When `i` is 4 then the `if` condition is satisfied and `continue` is executed.
3. After executing the `continue` statement, the next statement, (`printf`), is not executed; instead, the updated part of the `for` statement (`i++`) is executed.

c. What are escape sequences? What is effect of following Escape sequences? \a, \b, \r, \f (4)

Answer:



Q.4 a. What are the differences between malloc() and calloc()? (4)

Answer:

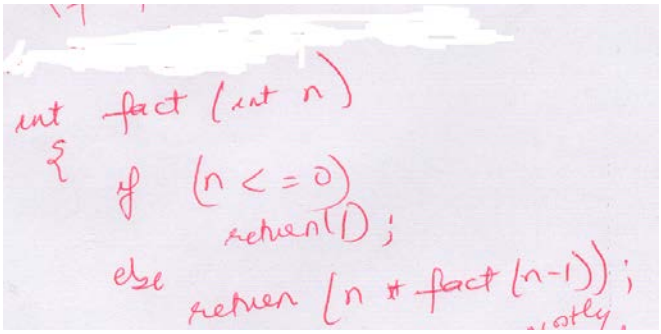
There are 2 differences:

First, is in the number of arguments. malloc() takes a single argument(memory required in bytes), while calloc() needs 2 arguments(number of variables to allocate memory, size in bytes of a single variable).

Secondly, malloc() does not initialize the memory allocated, while calloc() initializes the allocated memory to ZERO.

b. Write a recursive function to calculate factorial of a number. (4)

Answer:



c. Write a C language program to read two matrices and add them. (8)

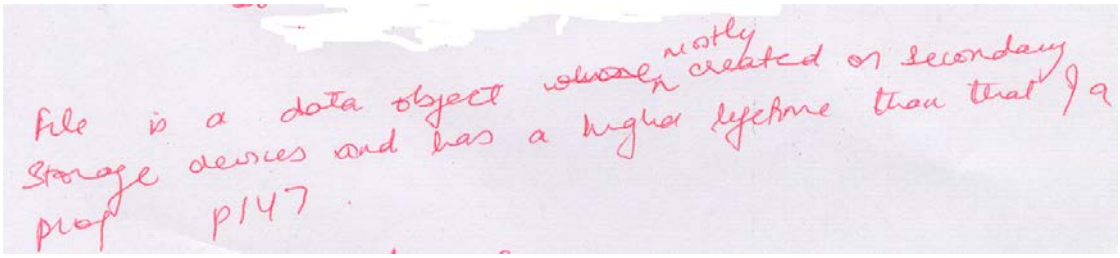
Answer:

```
#include<stdio.h>
#include<conio.h>
main()
{
    /*declaration of three array with same size */
    int a[10][10],b[10][10],c[10][10],i,j,m,n;
    printf("Enter the size of A and B metrix\n");
    scanf("%d%d",&m,&n);
    /* reading system */
    printf("Enter the elements of A metrix\n");
    for(i=0;i<m;i++)
            for(j=0;j<n;j++)
                    scanf("%d",&a[i][j]);
```

```
            printf("Enter the elements of B metrix\n");
    for(i=0;i<m;i++)
            for(j=0;j<n;j++)
                    scanf("%d",&b[i][j]);
           /* Addition Logic */
    for(i=0;i<m;i++)
            for(j=0;j<n;j++)
                 c[i][j] = a[i][j]+b[i][j];
            clrscr();
        /*display entire matrix */
            printf("\nA matrix is :\n\n");
    for(i=0;i<m;i++)
    {
             for(j=0;j<n;j++)
                    printf("%4d", a[i][j]);
             printf("\n");
    }
    printf("\nB matrix is :\n\n");

    for(i=0;i<m;i++)
    {
             for(j=0;j<n;j++)
            printf("%4d", b[i][j]);
             printf("\n");
    }

   printf("\nC matrix is :\n\n");
   for(i=0;i<m;i++)
   {
        for(j=0;j<n;j++)
        printf("%4d", c[i][j]);
        printf("\n");
   }

   getch();
}
```

Q.5 a. What is a file? Identify & explain the various types of operations that can be performed on sequential files. (6)

Answer:



THE CONCEPT OF FILES

Introduction

A *file* is a data object whose lifetime may be greater than the lifetime of a program responsible for creating it, because it is created on secondary storage devices. It is used to store persistent data values and information. The files are used mainly for input and output of data to an external operating environment. The components of the file are called as records (this term has nothing to do with record data structure).

Types of Files

A file may be a *sequential file*, a *direct-access file*, or an *indexed sequential file*. A sequential file can be thought of as a linear sequence of components of the same type with no fixed maximum bound. The major operations on the sequential files are:

Open operation: When a file is to be used, it is first required to be opened. The open operation requires two operands: the name of the file and the access mode telling whether the file is to be opened for reading or writing. If the access mode is "read," then the file must exist. If the access mode is "write," then if the file already exists, that file is emptied and the file position pointer is set to the start of the file. If the file does not exist then the operating system is requested to create a new empty file with a given name. The open operation requests the information about the locations and other properties of the file from the operating system. The operating system allocates the storage for this information and for buffers, and sets the file-position pointer to the first component of the file. The runtime library of C provides an `fopen(name,mode)` function for it. This function returns a pointer to the internal structure called FILE (you get the definition of this structure in `stdio.h`). This pointer is called a *file descriptor*; it is used by the C program to refer to the file for reading or writing purposes.

Read operation: This operation transfers the current file component to the designated program variable. The runtime library of C provides a function `fgetc(fp)`, where `fp` is a file descriptor, for `fscanf()`. `fscanf()` is similar to `scanf()` except that one extra parameter, `fp`, is required to be passed as the first parameter. The second and third parameters are the same as the first and second parameters of `scanf()`.

Write operation: This operation transfers the contents of the designated program variable to the new component created at the current position. The runtime library of C provides a function `fputc(c,fp)`, where `fp` is a file descriptor, and `c` is a character to be written in the file `fprintf()`. `fprintf()` is similar to `printf()` except that one extra parameter, `fp`, is required to be passed as the first parameter. The second and third parameters are the same as the first and second parameters of `printf()`.

Close operation: This operation notifies the operating system that the file can be detached from the program and that it can deallocate the internal storage used for the file. The file generally gets closed implicitly when the program terminates without explicit action by the programmer. But when the access mode is required to be changed it is required to be closed explicitly and reopened in the new mode. The runtime library of C provides an `fclose(fp)` function for it.

b. What is a pre-processor directive in C programming language? (4)

Answer:

The compiler reads C code, and (ultimately) turns that into Assembly. But sometimes you want to do more than C code can express. The trick, then, is to use the preprocessor. It looks at the source code before the compiler sees it, and based on certain directives, alters it in some way.If you are using gcc as your compiler, you can get it to output the code after the preprocessor has analyzed it by using the -E flag. For example, if we do:

```
#define ANSWER_TO_LIFE 42
int main( ) {
printf( "The answer to life, the universe, and everything: %d\n" , ANSWER_TO_LIFE );
}
```

Then we can run that as

```
gcc -E preprocessor_example.c
```

and it outputs

```
int main( ) {
printf( "The answer to life, the universe, and everything: %d\n" , 42 );
}
```

This should help you to test out different examples for yourself. Of course, when you include lots of libraries that will add a lot of other stuff into the code

c. Write a program to store information of 10 students using structure. (6)

Answer:

```
Ans: #include <stdio.h>
struct student{
   char name[50];
   int roll;
   float marks;
};
int main(){
   struct student s[10];
   int i;
   printf("Enter information of students:\n");
   for(i=0;i<10;++i)
   {
      s[i].roll=i+1;
      printf("\nFor roll number %d\n",s[i].roll);
```

```
        printf("Enter name: ");
        scanf("%s",s[i].name);
        printf("Enter marks: ");
        scanf("%f",&s[i].marks);
        printf("\n");
    }
    printf("Displaying information of students:\n\n");
    for(i=0;i<10;++i)
    {
     printf("\nInformation for roll number %d:\n",i+1);
     printf("Name: ");
     puts(s[i].name);
     printf("Marks: %.1f",s[i].marks);
    }
    return 0;
}
```

PART (B)

Answer at least TWO Questions from this part. Each question carries 16 marks.

Q.6 a. What data structures is used to perform recursion and why? (4)

Answer:

Stack. Because of its LIFO (Last In First Out) property it remembers its 'caller' so knows whom to return when the function has to return. Recursion makes use of system stack for storing the return addresses of the function calls.

Every recursive function has its equivalent iterative (non-recursive) function. Even when such equivalent iterative procedures are written, explicit stack is to be used.

b. Differentiate between the data structures, a queue and a stack. (4)

Answer:

A queue is typically FIFO (priority queues don't quite follow that) while a stack is LIFO. Elements get inserted at one end of a queue and retrieved from the other, while the insertion and removal operations for a stack are done at the same end.

c. Write the algorithm for sorting a list of numbers using bubble sort. (8)

Answer:

BUBBLE SORT

Introduction

Bubble sorting is a simple sorting technique in which we arrange the elements of the list by forming pairs of adjacent elements. That means we form the pair of the i^{th} and $(i+1)^{th}$ element. If the order is ascending, we interchange the elements of the pair if the first element of the pair is greater than the second element. That means for every pair (list[i],list[i+1]) for i :=1 to (n–1) if list[i] > list[i+1], we need to interchange list[i] and list[i+1]. Carrying this out once will move the element with the highest value to the last or n^{th} position. Therefore, we repeat this process the next time with the elements from the first to $(n–1)th$ positions. This will bring the highest value from among the remaining $(n–1)$ values to the $(n–1)^{th}$ position. We repeat the process with the remaining $(n–2)$ values and so on. Finally, we arrange the elements in ascending order. This requires to perform $(n–1)$ passes. In the first pass we have $(n–1)$ pairs, in the second pass we have $(n–2)$ pairs, and in the last (or $(n–1)^{th}$) pass, we have only one pair. Therefore, the number of probes or comparisons that are required to be carried out is

$$(n–1) + (n–2) + (n–3) + \ldots + 1$$
$$= n(n–1)/2,$$

and the order of the algorithm is $O(n^2)$.

Program

```
#include <stdio.h>
#define MAX 10
void swap(int *x,int *y)
{
    int temp;
    temp = *x;
    *x = *y;
    *y = temp;
}
void bsort(int list[], int n)
```

```
{
    int i,j;
    for(i=0;i<(n-1);i++)
        for(j=0;j<(n-(i+1));j++)
            if(list[j] > list[j+1])
                swap(&list[j],&list[j+1]);
}

void readlist(int list[],int n)
{
    int i;
    printf("Enter the elements\n");
    for(i=0;i<n;i++)
        scanf("%d",&list[i]);
}

void printlist(int list[],int n)
{
    int i;
    printf("The elements of the list are: \n");
    for(i=0;i<n;i++)
        printf("%d\t",list[i]);
}

void main()
{
    int list[MAX], n;
    printf("Enter the number of elements in the list max = 10\n");
    scanf("%d",&n);
    readlist(list,n);
    printf("The list before sorting is:\n");
    printlist(list,n);
    bsort(list,n);
    printf("The list after sorting is:\n");
    printlist(list,n);
}
```

Example

Input

```
Enter the number of elements in the list, max = 10
5
Enter the elements
23
5
4
9
1
```

Output

```
The list before sorting is:
The elements of the list are:
23   5   4   9   1
The list after sorting is:
The elements of the list are:
1    4    5    9    23
```

Q.7 a. Write an algorithm to find the Smallest Element in the Array. (8)

Answer:

Smallest element(a, n, k, loc)

Here a is linear array of size n. This sub algorithm finds the location loc of smallest element among a[k-1],a[k+1],a[k+2]…a[n-1]. A temporary variable "small" is used to hold the current smallest element and j is used as the loop control variable.

Begin

set small=a[k-1] set loc=k-1

for j=k to (n-1) by 1
do if(a[j]<small) then
set small = a[j]

```
set
loc=j endif

endfor
end
```

b. For a graph, define the following:
(i) Degree of a vertex (ii) Simple path
(iii) Completely connected graph (iv) Maximum number of edges (8)

Answer:

Degree of vertex: The number of edges incident onto the vertex. For example, in graph G_1, the degree of vertex 1 is 3, because 3 edges are incident onto it. For a directed graph, we need to define indegree and outdegree. *Indegree* of a vertex vi is the number of edges incident onto vi, with vi as the head. *Outdegree* of vertex vi is the number of edges incident onto vi, with vi as the tail. For graph G_2, the indegree of vertex 2 is 1, whereas the outdegree of vertex 2 is 2.

Simple path: A simple path is a path given by a sequence of vertices in which all vertices are distinct except the first and the last vertices. If the first and the last vertices are same, the path will be a cycle.

Completely connected graph: A graph G is completely connected if, for every pair of distinct vertices (v_i, v_j), there exists an edge. A completely connected graph is shown in Figure 22.6.

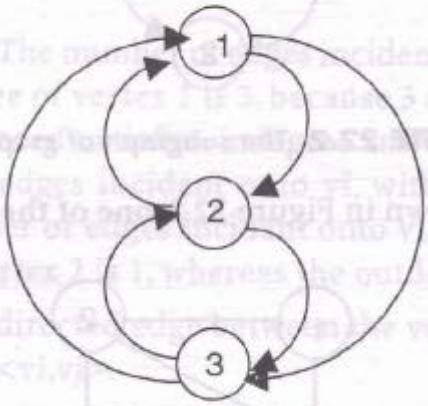


FIGURE 22.6 A completely connected graph.

Maximum number of edges: The maximum number of edges in an undirected graph with n vertices is $n(n-1)/2$. In a directed graph, it is $n(n-1)$.

Q.8 a. Write a program in C to allocate memory dynamically for strings, and store their addresses in array of pointers to strings. (8)

Answer:

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

```
#include <string.h>

void main( )
{
 char *name[5] ;
 char str[20] ;
 int i ;

 clrscr( ) ;

 for ( i = 0 ; i < 5 ; i++ )
 {
 printf ( "Enter a String: " ) ;  gets ( str ) ;
 name[i] = ( char *) malloc ( strlen ( str ) + 1 ) ;
 strcpy ( name[i], str ) ;
 }

 printf ( "\nThe strings are:" ) ;

 for ( i = 0 ; i < 5 ; i++ )
 printf ( "\n%s", name[i] ) ;

 for ( i = 0 ; i < 5 ; i++ )
 free ( name[i] ) ;

 getch( ) ;
}
```

b. Write C function to

(i) Create a linked list

(ii) Delete an element from a linked list **(8)**

Answer:

Here is a program for building and printing the elements of the linked list:

```
# include <stdio.h>
# include <stdlib.h>
struct node
{
int data;
struct node *link;
};
struct node *insert(struct node *p , int n)
{
struct node *temp;
 /* if the existing list is empty then insert a new node as the
starting node */
  if(p==NULL)
    {
       p=(struct node *)malloc(sizeof(struct node)); /* creates new node
data value passes
 as  parameter */
       if(p==NULL)
       {
     printf("Error\n");
           exit(0);
       }
       p-> data = n;
       p-> link = p; /* makes the pointer pointing to itself because it
is a circular list*/
     }
     else
     {
       temp = p;
  /* traverses the existing list to get the pointer to the last node of
```

```
it */
   while (temp-> link != p)
      temp = temp-> link;
       temp-> link =  (struct node *)malloc(sizeof(struct node)); /*
creates new node using
        data value passes as
          parameter  and puts its
         address in the link field
         of last node of the
         existing list*/
       if(temp -> link == NULL)
       {
     printf("Error\n");
          exit(0);
       }
       temp = temp-> link;
       temp-> data = n;
       temp-> link = p;
      }
      return (p);
  }
  void printlist ( struct node *p  )
  {
   struct node *temp;
    temp = p;
   printf("The data values in the list are\n");
      if(p!= NULL)
      {
         do
                {
                printf("%d\t",temp->data);
                temp=temp->link;
                } while (temp!= p);
       }
       else
          printf("The list is empty\n");
   }

  void main()
  {
       int n;
       int x;
       struct node *start = NULL ;
```

```
    printf("Enter the nodes to be created \n");
    scanf("%d",&n);
    while ( n-- > 0 )
    {
  printf( "Enter the data values to be placed in a node\n");
      scanf("%d",&x);
      start = insert ( start , x );
    }
    printf("The created list is\n");
    printlist ( start );
}
```

Program

```
# include <stdio.h>
# include <stdlib.h>
struct node *delet ( struct node *, int );
int length ( struct node * );
struct node
{
    int data;
    struct node *link;
};
struct node *insert(struct node *p , int n)
{
    struct node *temp;
   if(p==NULL)
   {
      p=(struct node *)malloc(sizeof(struct node));
      if(p==NULL)
      {
            printf("Error\n");
        exit(0);
      }
      p-> data = n;
      p-> link = NULL;
    }
    else
    {
      temp = p;
```

```
    while (temp-> link != NULL)
     temp = temp-> link;
     temp-> link  =  (struct node *)malloc(sizeof(struct node));
    if(temp -> link == NULL)
    {
            printf("Error\n");
      exit(0);
    }
     temp = temp-> link;
    temp-> data = n;
    temp-> link= NULL;
   }
    return (p);
}

void printlist ( struct node *p  )
{
   printf("The data values in the list are\n");
   while (p!= NULL)
    {
       printf("%d\t",p-> data);
      p = p-> link;
    }
}
void main()
{
    int n;
   int x;
   struct node *start = NULL;
   printf("Enter the nodes to be created \n");
   scanf("%d",&n);
   while ( n-- > 0 )
   {
       printf( "Enter the data values to be placed in a node\n");
      scanf("%d",&x);
      start = insert ( start, x );
   }
    printf(" The list before deletion is\n");
   printlist ( start );
   printf(" \n Enter the node no \n");
   scanf ( " %d",&n);
   start = delet (start , n );
```

```
    printf(" The list after deletion is\n");
    printlist ( start );
}

  /* a function to delete the specified node*/
struct node *delet ( struct node *p , int node_no )
{
    struct node *prev , *curr ;
    int i;

    if (p == NULL )
    {
        printf("There is no node to be deleted \n");
    }
    else
    {
        if ( node_no > length (p))
        {
                printf("Error\n");
        }
        else
        {
                prev = NULL;
                curr = p;
                i = 1 ;
                while ( i < node_no )
                {
                        prev = curr;
                        curr = curr-> link;
                        i = i+1;
                }
                if ( prev == NULL )
                {
                        p = curr -> link;
                        free ( curr );
                }
                else
                {
                        prev -> link  = curr -> link ;
                        free ( curr );
                }
        }
```

```
    }
    return(p);
}
/* a function to compute the length of a linked list */
int length ( struct node *p )
{
    int count = 0 ;
    while ( p != NULL )
    {
        count++;
        p = p->link;
    }
    return ( count ) ;
}
```

Q.9 a. Write an algorithm in C to search for an element in a list of elements using Binary Search. (8)

Answer:

BINARY SEARCH

Introduction

The prerequisite for using binary search is that the list must be a sorted one. We compare the element to be searched with the element placed approximately in the middle of the list.

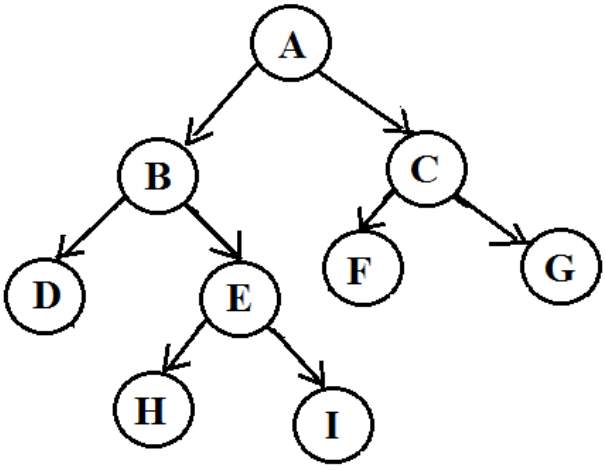
If a match is found, the search terminates successfully. Otherwise, we continue the search for the key in a similar manner either in the upper half or the lower half. If the elements of the list are arranged in ascending order, and the key is less than the element in the middle of the list, the search is continued in the lower half. If the elements of the list are arranged in descending order, and the key is greater than the element in the middle of the list, the search is continued in the upper half of the list. The procedure for the binary search is given in the following program.

Program

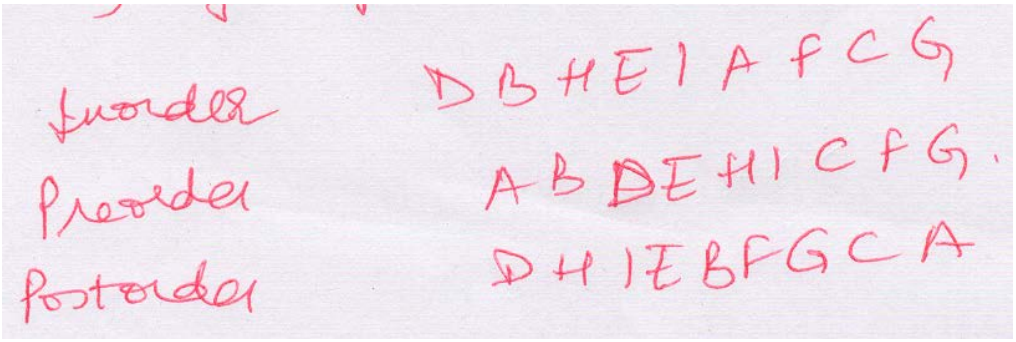
```
#include <stdio.h>
#define MAX 10

void bsearch(int list[],int n,int element)
{
    int l,u,m, flag = 0;
    l = 0;
    u = n-1;
    while(l <= u)
    {
        m = (l+u)/2;
       if( list[m] == element)
        {
                printf(" The element whose value is %d is present at
position %d in list\n",
                        element,m);
                flag =1;
                break;
        }
       else
                if(list[m] < element)
                        l = m+1;
                else
                        u = m-1;
    }

    if( flag == 0)
    printf("The element whose value is %d is not present in the list\n",
        element);
}
void readlist(int list[],int n)
{
    int i;
    printf("Enter the elements\n");
    for(i=0;i<n;i++)
        scanf("%d",&list[i]);
}
```

```
void printlist(int list[],int n)
{
    int i;
    printf("The elements of the list are: \n");
    for(i=0;i<n;i++)
        printf("%d\t",list[i]);
}
void main()
{
    int list[MAX], n, element;
    printf("Enter the number of elements in the list max = 10\n");
    scanf("%d",&n);
    readlist(list,n);
    printf("\nThe list before sorting is:\n");
    printlist(list,n);
    printf("\nEnter the element to be searched\n");
    scanf("%d",&element);
    bsearch(list,n,element);
}
```

b. Traverse the following tree in
(i) Inorder
(ii) Preorder
(iii) Postorder
and give the output. **(8)**



Answer:



TEXT BOOK

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