D.Y.Patil University, Ambi, Pune

Laboratory Manual

Subject: Advanced Data Structures Lab Code: UDSPC409/UITUPC407

Batch: 2022

Semester: IV

Sr. No	Title of the experiments	Unit. No	Approx. Hrs. Required
1	Write a program in C to implement hash Table.	**1**	**4 hrs**
2	Write a Program in C to implement Hash Table using Open Addressing.	**1**	**4 hrs**
3	Write a Program in C for Prim's greedy Algorithm to find the minimum spanning tree from a graph.	**2**	**4 hrs.**
4	Write a Program in C for Dijkstra's algorithm to find the shortest path between any two vertices of a graph.	**2**	**4 hrs.**
5	Write a Program in C to convert given binary tree into threaded binary tree.	**3**	**4 hrs**
6	Write a Program in C for depth first and breadth first traversal for a binary tree.	**3**	**4 hrs**
7	Write a Program to find Optimal Binary Search Tree using Dynamic Method in C	**4**	**4 hrs**
8	Write a Program in C to find the Maximum Depth or Height of given Binary Tree.	**4**	**4 hrs**
9	Write a C Program To Perform Insertion, Deletion and Traversal In B-Tree.	**5**	**4 hrs**
	Write a C Program for searching a key on a B-tree.	**5**	**4 hrs**
10	Write a Program in C for Random access file to read or write any data in our disk file without reading or writing every piece of data before it.	**6**	**4 hrs**
11	Case Study- Design a mini project to implement a Smart text editor.	**6**	**4 hrs**

Experiment No. 01: Write a program in C to implement hash Table.

What is hashing?

Hashing is the process of transforming any given key or a string of characters into another value. This is usually represented by a shorter, fixed-length value or key that represents and makes it easier to find or employ the original string.

The most popular use for hashing is the implementation of hash tables. A hash table stores key and value pairs in a list that is accessible through its index. Because key and value pairs are unlimited, the hash function will map the keys to the table size. A hash value then becomes the index for a specific element.

A hash function generates new values according to a mathematical hashing algorithm, known as a hash value or simply a hash. To prevent the conversion of hash back into the original key, a good hash always uses a one-way hashing algorithm.

Hash Function Method:

Division Method:

This is the most simple and easiest method to generate a hash value. The hash function divides the value k by M and then uses the remainder obtained.

Formula:

$h(K) = k \bmod M$

Here,

k is the key value, and

M is the size of the hash table.

It is best suited that M is a prime number as that can make sure the keys are more uniformly distributed. The hash function is dependent upon the remainder of a division.

Example:

$k = 12345$

$M = 95$

$h(12345) = 12345 \bmod 95$

$= 90$

Problem Description

A hash table is a data structure used to implement an associative array, a structure that can map keys to values. A hash table uses a hash function to compute an index into an array of buckets or slots. This program will implement a hash table by putting each element in a particular index of hash table array.

Problem Solution

1. Create an array of structure, data (i.e a hash table).

2. Take a key to be stored in hash table as input.

3. Corresponding to the key, an index will be generated.

4. In case of absence of data in that index of array, create one and insert the data item (key and value) into it and increment the size of hash table.

5. In case the data already exists, the new data cannot be inserted if the already present data does not match given key.

6. To display all the elements of hash table, data at each index is extracted and elements (key and value) are read and printed.

7. To remove a key from hash table, we will first calculate its index and extract its data, delete the key in case it matches the given key.

8. Exit

Program/Source Code

```
#include<stdio.h>

#define size 7

int array[size];

void init()

{

   int i;

   for(i = 0; i < size; i++)

      array[i] = -1;

}

 void insert(int val)

{

   int key = val % size;

     if(array[key] == -1)

   {

      array[key] = val;

      printf("%d inserted at array[%d]\n", val,key);

   }

   else

   {

      printf("Collision : array[%d] has element %d already!\n",key,array[key]);
```

```
        printf("Unable to insert %d\n",val);
    }
}

void del(int val)
{
    int key = val % size;
    if(array[key] == val)
        array[key] = -1;
    else
        printf("%d not present in the hash table\n",val);
}
  void search(int val)
{
    int key = val % size;
    if(array[key] == val)
        printf("Search Found\n");
    else
        printf("Search Not Found\n");
}
  void print()
{
    int i;
    for(i = 0; i < size; i++)
        printf("array[%d] = %d\n",i,array[i]);
}
  int main()
{
    init();
    insert(10);
```

```
    insert(4);
    insert(2);
    insert(3);

    printf("Hash table\n");
    print();
    printf("\n");
      printf("Deleting value 10..\n");
    del(10);
    printf("After the deletion hash table\n");
    print();
    printf("\n");
      printf("Deleting value 5..\n");
    del(5);
    printf("After the deletion hash table\n");
    print();
    printf("\n");
      printf("Searching value 4..\n");
    search(4);
    printf("Searching value 10..\n");
    search(10);
      return 0;
}
```

Output

```
10 inserted at array[3]
4 inserted at array[4]
2 inserted at array[2]
Collision : array[3] has element 10 already!
Unable to insert 3
Hash table
```

array[0] = -1

array[1] = -1

array[2] = 2

array[3] = 10

array[4] = 4

array[5] = -1

array[6] = -1

Deleting value 10..

After the deletion hash table

array[0] = -1

array[1] = -1

array[2] = 2

array[3] = -1

array[4] = 4

array[5] = -1

array[6] = -1

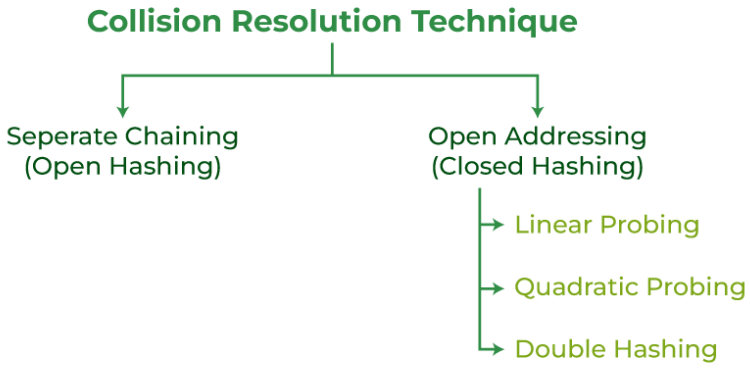
Experiment No. 02: Write a Program in C to implement Hash Table using Open Addressing.

What is collision?

The hashing process generates a small number for a big key, so there is a possibility that two keys could produce the same value. The situation where the newly inserted key maps to an already occupied, and it must be handled using some collision handling technology.

How to handle Collisions?

There are mainly two methods to handle collision:



Program:

```
#include <stdio.h>
#include<stdlib.h>
#define TABLE_SIZE 10
int h[TABLE_SIZE]={NULL};
void insert()
{
 int key,index,i,flag=0,hkey;
 printf("\nenter a value to insert into hash table\n");
 scanf("%d",&key);
 hkey=key%TABLE_SIZE;
 for(i=0;i<TABLE_SIZE;i++)
   {
    index=(hkey+i)%TABLE_SIZE;
    if(h[index] == NULL)
    {
```

```
        h[index]=key;
         break;
     }
    }
    if(i == TABLE_SIZE)
     printf("\nelement cannot be inserted\n");
}
void search()
{
 int key,index,i,flag=0,hkey;
 printf("\nenter search element\n");
 scanf("%d",&key);
 hkey=key%TABLE_SIZE;
 for(i=0;i<TABLE_SIZE; i++)
 {
    index=(hkey+i)%TABLE_SIZE;
    if(h[index]==key)
    {
      printf("value is found at index %d",index);
      break;
    }
  }
  if(i == TABLE_SIZE)
    printf("\n value is not found\n");
}
void display()
{
  int i;
  printf("\nelements in the hash table are \n");
  for(i=0;i< TABLE_SIZE; i++)
```

```
  printf("\nat index %d \t value =  %d",i,h[i]);
}
main()
{
   int opt,i;
   while(1)
   {
      printf("\nPress 1. Insert\t 2. Display \t3. Search \t4.Exit \n");
      scanf("%d",&opt);
      switch(opt)
      {
         case 1:
            insert();
            break;
         case 2:
            display();
            break;
         case 3:
            search();
            break;
         case 4:exit(0);
      }
   }
}
```

Output

```
Press 1. Insert     2. Display     3. Search     4.Exit
1
enter a value to insert into hash table
12
elements in the hash table are
```

at index 0 value = 0

at index 1 value = 0

at index 2 value = 12

at index 3 value = 13

at index 4 value = 22

at index 5 value = 0

at index 6 value = 0

at index 7 value = 0

at index 8 value = 0

at index 9 value = 0

Experiment No. 03: Write a Program in C for Prim's greedy Algorithm to find the minimum spanning tree from a graph.

Pim's Algorithm is used to manage Internet Radio Broadcasting to transfer information to the listeners efficiently. An efficient transfer is vital for Internet radio so that all the listeners that are tuned in receive the whole data they need to reconstruct the songs they're listening to.

Introduction to Prim's Algorithm

Prim's algorithm is used to find the Minimum Spanning Tree for a given graph. But, what is a Minimum Spanning Tree, or MST for short? A minimum spanning tree T(V', E') is a subset of graph G(V, E) with the same number of vertices as of graph G (V' = V) and edges equal to the number of vertices of graph G minus one (E' = |V| - 1). Prim's approach identifies the subset of edges that includes every vertex in the graph, and allows the sum of the edge weights to be minimized.

Prim's algorithm starts with a single node and works its way through several adjacent nodes, exploring all of the connected edges along the way. Edges with the minimum weights that do not cause cycles in the graph get selected for t inclusion in the MST structure. Hence, we can say that Prim's algorithm takes a locally optimum decision in order to find the globally optimal solution. That is why it is also known as a Greedy Algorithm.

How to Create MST Using Prim's Algorithm

Let's first look into the steps involved in Prim's Algorithm to generate a minimum spanning tree:

Step 1: Determine the arbitrary starting vertex.

Step 2: Keep repeating steps 3 and 4 until the fringe vertices (vertices not included in MST)remain.

Step 3: Select an edge connecting the tree vertex and fringe vertex having the minimum weight.

Step 4: Add the chosen edge to MST if it doesn't form any closed cycle.

Step 5: Exit

```
// A C program for Prim's Minimum

// Spanning Tree (MST) algorithm. The program is

// for adjacency matrix representation of the graph

#include <limits.h>

#include <stdbool.h>

#include <stdio.h>

// Number of vertices in the graph

#define V 5

// A utility function to find the vertex with
```

```
// minimum key value, from the set of vertices
// not yet included in MST
int minKey(int key[], bool mstSet[])
{
	// Initialize min value
	int min = INT_MAX, min_index;
	for (int v = 0; v < V; v++)
		if (mstSet[v] == false && key[v] < min)
			min = key[v], min_index = v;
	return min_index;
}
// A utility function to print the
// constructed MST stored in parent[]
int printMST(int parent[], int graph[V][V])
{
	printf("Edge \tWeight\n");
	for (int i = 1; i < V; i++)
		printf("%d - %d \t%d \n", parent[i], i,
			graph[i][parent[i]]);
}
// Function to construct and print MST for
// a graph represented using adjacency
// matrix representation
void primMST(int graph[V][V])
{
	// Array to store constructed MST
	int parent[V];
	// Key values used to pick minimum weight edge in cut
	int key[V];
	// To represent set of vertices included in MST
```

```
bool mstSet[V];
// Initialize all keys as INFINITE
for (int i = 0; i < V; i++)
        key[i] = INT_MAX, mstSet[i] = false;
// Always include first 1st vertex in MST.
// Make key 0 so that this vertex is picked as first
// vertex.
key[0] = 0;
// First node is always root of MST
parent[0] = -1;
// The MST will have V vertices
for (int count = 0; count < V - 1; count++) {

        // Pick the minimum key vertex from the
        // set of vertices not yet included in MST
        int u = minKey(key, mstSet);
        // Add the picked vertex to the MST Set
        mstSet[u] = true;
        // Update key value and parent index of
        // the adjacent vertices of the picked vertex.
        // Consider only those vertices which are not
        // yet included in MST
        for (int v = 0; v < V; v++)
                // graph[u][v] is non zero only for adjacent
                // vertices of m mstSet[v] is false for vertices
                // not yet included in MST Update the key only
                // if graph[u][v] is smaller than key[v]
                if (graph[u][v] && mstSet[v] == false
                        && graph[u][v] < key[v])
                        parent[v] = u, key[v] = graph[u][v];
```

```
        }
        // print the constructed MST
        printMST(parent, graph);
}
// Driver's code
int main()
{
        int graph[V][V] = { { 0, 2, 0, 6, 0 },
                                                { 2, 0, 3, 8, 5 },
                                                { 0, 3, 0, 0, 7 },
                                                { 6, 8, 0, 0, 9 },
                                                { 0, 5, 7, 9, 0 } };

        // Print the solution
        primMST(graph);
        return 0;
}
```

Output

```
Edge     Weight
0 - 1    2
1 - 2    3
0 - 3    6
1 - 4    5
```

Experiment No. 04: Write a Program in C for Dijkstra's algorithm to find the shortest path between any two vertices of a graph.

Dijkstra’s algorithm is very similar to Prim’s algorithm for minimum spanning tree.

Like Prim’s MST, generate a SPT (shortest path tree) with a given source as a root. Maintain two sets, one set contains vertices included in the shortest-path tree, other set includes vertices not yet included in the shortest-path tree. At every step of the algorithm, find a vertex that is in the other set (set not yet included) and has a minimum distance from the source.

Follow the steps below to solve the problem:

Create a set sptSet (shortest path tree set) that keeps track of vertices included in the shortest-path tree, i.e., whose minimum distance from the source is calculated and finalized. Initially, this set is empty.

Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign the distance value as 0 for the source vertex so that it is picked first.

While sptSet doesn’t include all vertices

Pick a vertex u which is not there in sptSet and has a minimum distance value.

Include u to sptSet.

Then update distance value of all adjacent vertices of u.

To update the distance values, iterate through all adjacent vertices.

For every adjacent vertex v, if the sum of the distance value of u (from source) and weight of edge u-v, is less than the distance value of v, then update the distance value of v.

Note: We use a boolean array sptSet[] to represent the set of vertices included in SPT. If a value sptSet[v] is true, then vertex v is included in SPT, otherwise not. Array dist[] is used to store the shortest distance values of all vertices.

```
// C program for Dijkstra's single source shortest path

// algorithm. The program is for adjacency matrix

// representation of the graph

#include <limits.h>

#include <stdbool.h>

#include <stdio.h>

// Number of vertices in the graph

#define V 9

// A utility function to find the vertex with minimum

// distance value, from the set of vertices not yet included

// in shortest path tree
```

```
int minDistance(int dist[], bool sptSet[])
{
	// Initialize min value
	int min = INT_MAX, min_index;
	for (int v = 0; v < V; v++)
		if (sptSet[v] == false && dist[v] <= min)
			min = dist[v], min_index = v;

	return min_index;
}
// A utility function to print the constructed distance
// array
void printSolution(int dist[])
{
	printf("Vertex \t\t Distance from Source\n");
	for (int i = 0; i < V; i++)
		printf("%d \t\t\t\t %d\n", i, dist[i]);
}
// Function that implements Dijkstra's single source
// shortest path algorithm for a graph represented using
// adjacency matrix representation
void dijkstra(int graph[V][V], int src)
{
	int dist[V]; // The output array. dist[i] will hold the
				// shortest
	// distance from src to i
	bool sptSet[V]; // sptSet[i] will be true if vertex i is
					// included in shortest
	// path tree or shortest distance from src to i is
	// finalized
```

```
        // Initialize all distances as INFINITE and stpSet[] as
        // false
        for (int i = 0; i < V; i++)
                dist[i] = INT_MAX, sptSet[i] = false;
        // Distance of source vertex from itself is always 0
        dist[src] = 0;
        // Find shortest path for all vertices
        for (int count = 0; count < V - 1; count++) {
                // Pick the minimum distance vertex from the set of
                // vertices not yet processed. u is always equal to
                // src in the first iteration.
                int u = minDistance(dist, sptSet);
                // Mark the picked vertex as processed
                sptSet[u] = true;
                // Update dist value of the adjacent vertices of the
                // picked vertex.
                for (int v = 0; v < V; v++)
                        // Update dist[v] only if is not in sptSet,
                        // there is an edge from u to v, and total
                        // weight of path from src to v through u is
                        // smaller than current value of dist[v]
                        if (!sptSet[v] && graph[u][v]
                                && dist[u] != INT_MAX
                                && dist[u] + graph[u][v] < dist[v])
                                dist[v] = dist[u] + graph[u][v];
        }
        // print the constructed distance array
        printSolution(dist);
}
// driver's code
```

```
int main()
{
    /* Let us create the example graph discussed above */
    int graph[V][V] = { { 0, 4, 0, 0, 0, 0, 0, 8, 0 },
                        { 4, 0, 8, 0, 0, 0, 0, 11, 0 },
                        { 0, 8, 0, 7, 0, 4, 0, 0, 2 },
                        { 0, 0, 7, 0, 9, 14, 0, 0, 0 },
                        { 0, 0, 0, 9, 0, 10, 0, 0, 0 },
                        { 0, 0, 4, 14, 10, 0, 2, 0, 0 },
                        { 0, 0, 0, 0, 0, 2, 0, 1, 6 },
                        { 8, 11, 0, 0, 0, 0, 1, 0, 7 },
                        { 0, 0, 2, 0, 0, 0, 6, 7, 0 } };
    // Function call
    dijkstra(graph, 0);
    return 0;
}
```

Output

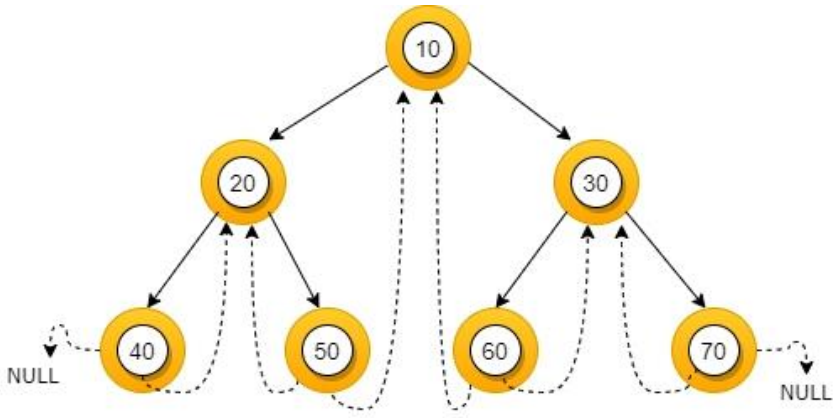
Vertex	Distance from Source
0	0
1	4
2	12
3	19
4	21
5	11
6	9
7	8
8	14

Experiment No. 05: Write a Program in C to convert given binary tree into threaded binary tree.

A threaded binary tree is just like a normal binary tree, but they have a specialty in which all the right child pointers that are NULL point to their in-order successor, and all the left child pointers that are NULL point to their in-order predecessor. It helps in facilitating faster tree traversal without requiring a stack or Recursion.

Structure of Node in Threaded Binary Tree

The structure of a node in a threaded binary tree is quite similar to that of a binary tree but with some modifications. For threaded binary trees, we need to use extra boolean variables in the node structure:



Threaded Binary Tree

```
#include <stdio.h>

#include <stdlib.h>

typedef enum {false,true} boolean;

struct node *in_succ(struct node *p);

struct node *in_pred(struct node *p);

struct node *insert(struct node *root, int ikey);

struct node *del(struct node *root, int dkey);

struct node *case_a(struct node *root, struct node *par,struct node *ptr);

struct node *case_b(struct node *root,struct node *par,struct node *ptr);

struct node *case_c(struct node *root, struct node *par,struct node *ptr);

void inorder( struct node *root);

void preorder( struct node *root);

struct node

{
```

```
        struct node *left;
        boolean lthread;
        int info;
        boolean rthread;
        struct node *right;
};
int main( )
{
clrscr();
    int choice,num;
        struct node *root=NULL;
        while(1)
        {
          printf("\Program of Threaded Tree in C\n");
              printf("1.Insert\n");
              printf("2.Delete\n");
              printf("3.Inorder Traversal\n");
              printf("4.Preorder Traversal\n");
              printf("5.Quit\n");
              printf("\nEnter your choice : ");
              scanf("%d",&choice);
              switch(choice)
              {
               case 1:
                    printf("\nEnter the number to be inserted : ");
                    scanf("%d",&num);
                    root = insert(root,num);
                    break;
               case 2:
                    printf("\nEnter the number to be deleted : ");
```

```
                scanf("%d",&num);
                root = del(root,num);
                break;
            case 3:
                inorder(root);
                break;
            case 4:
                preorder(root);
                break;
            case 5:
                 exit(1);
            default:
                printf("\nWrong choice\n");
        }
    }
        return 0;
}
struct node *insert(struct node *root, int ikey)
{
    struct node *tmp,*par,*ptr;
    int found=0;
    ptr = root;
    par = NULL;
    while( ptr!=NULL )
    {
        if( ikey == ptr->info)
        {
            found =1;
            break;
        }
```

```
    par = ptr;
    if(ikey < ptr->info)
    {
        if(ptr->lthread == false)
            ptr = ptr->left;
        else
            break;
    }
    else
    {
        if(ptr->rthread == false)
            ptr = ptr->right;
        else
            break;
    }
}
if(found)
    printf("\nDuplicate key");
else
{
    tmp=(struct node *)malloc(sizeof(struct node));
    tmp->info=ikey;
    tmp->lthread = true;
    tmp->rthread = true;
    if(par==NULL)
    {
        root=tmp;
        tmp->left=NULL;
        tmp->right=NULL;
    }
```

```
        else if( ikey < par->info )
        {
            tmp->left=par->left;
            tmp->right=par;
            par->lthread=false;
            par->left=tmp;
        }
        else
        {
            tmp->left=par;
            tmp->right=par->right;
            par->rthread=false;
            par->right=tmp;
        }
    }
    return root;
}
struct node *del(struct node *root, int dkey)
{
    struct node *par,*ptr;
    int found=0;
    ptr = root;
    par = NULL;
    while( ptr!=NULL)
    {
        if( dkey == ptr->info)
        {
            found =1;
            break;
        }
```

```
        par = ptr;
        if(dkey < ptr->info)
        {
            if(ptr->lthread == false)
                ptr = ptr->left;
            else
                break;
        }
        else
        {
            if(ptr->rthread == false)
                ptr = ptr->right;
            else
                break;
        }
    }
    if(found==0)
        printf("\ndkey not present in tree");
    else if(ptr->lthread==false && ptr->rthread==false)/*2 children*/
        root = case_c(root,par,ptr);
    else if(ptr->lthread==false )
    root = case_b(root, par,ptr);
    else if(ptr->rthread==false)
    root = case_b(root, par,ptr);
    else
        root = case_a(root,par,ptr);
    return root;
}
struct node *case_a(struct node *root, struct node *par,struct node *ptr )
{
```

```
    if(par==NULL)
        root=NULL;
    else if(ptr==par->left)
    {
        par->lthread=true;
        par->left=ptr->left;
    }
    else
    {
        par->rthread=true;
        par->right=ptr->right;
    }
    free(ptr);
    return root;
}
struct node *case_b(struct node *root,struct node *par,struct node *ptr)
{
    struct node *child,*s,*p;
    if(ptr->lthread==false)
        child=ptr->left;
    else
        child=ptr->right;
        if(par==NULL )
                root=child;
        else if( ptr==par->left)
                par->left=child;
        else
                par->right=child;
        s=in_succ(ptr);
        p=in_pred(ptr);
```

```
        if(ptr->lthread==false)
                p->right=s;
        else
        {
        if(ptr->rthread==false)
                s->left=p;
        }
        free(ptr);
        return root;
}
struct node *case_c(struct node *root, struct node *par,struct node *ptr)
{
        struct node *succ,*parsucc;
        parsucc = ptr;
        succ = ptr->right;
        while(succ->left!=NULL)
        {
                parsucc = succ;
                succ = succ->left;
        }
        ptr->info = succ->info;
        if(succ->lthread==true && succ->rthread==true)
                root = case_a(root, parsucc,succ);
        else
                root = case_b(root, parsucc,succ);
        return root;
}
struct node *in_succ(struct node *ptr)
{
        if(ptr->rthread==true)
```

```
        return ptr->right;
    else
    {
        ptr=ptr->right;
        while(ptr->lthread==false)
            ptr=ptr->left;
        return ptr;
    }
}
struct node *in_pred(struct node *ptr)
{
    if(ptr->lthread==true)
        return ptr->left;
    else
    {
        ptr=ptr->left;
        while(ptr->rthread==false)
            ptr=ptr->right;
        return ptr;
    }
}
void inorder( struct node *root)
{
    struct node *ptr;
    if(root == NULL )
    {
        printf("Tree is empty");
        return;
    }
    ptr=root;
```

```
        while(ptr->lthread==false)
                ptr=ptr->left;
        while( ptr!=NULL )
        {
                printf("%d ",ptr->info);
                ptr=in_succ(ptr);
        }
}
void preorder(struct node *root )
{
        struct node *ptr;
        if(root==NULL)
        {
                printf("Tree is empty");
                return;
        }
        ptr=root;
        while(ptr!=NULL)
        {
                printf("%d ",ptr->info);
                if(ptr->lthread==false)
                        ptr=ptr->left;
                else if(ptr->rthread==false)
                        ptr=ptr->right;
                else
                {
                        while(ptr!=NULL && ptr->rthread==true)
                                ptr=ptr->right;
                        if(ptr!=NULL)
                                ptr=ptr->right;
```

```
            }
        }
    }
```

Experiment No. 06: Write a Program in C for depth first and breadth first traversal for a binary tree.

The breadth-first search (BFS) algorithm is used to search a tree or graph data structure for a node that meets a set of criteria. It starts at the tree's root or graph and searches/visits all nodes at the current depth level before moving on to the nodes at the next depth level. Breadth-first search can be used to solve many problems in graph theory.

Breadth-First Traversal (or Search) for a graph is similar to the Breadth-First Traversal of a tree.

The only catch here is, that, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we divide the vertices into two categories:

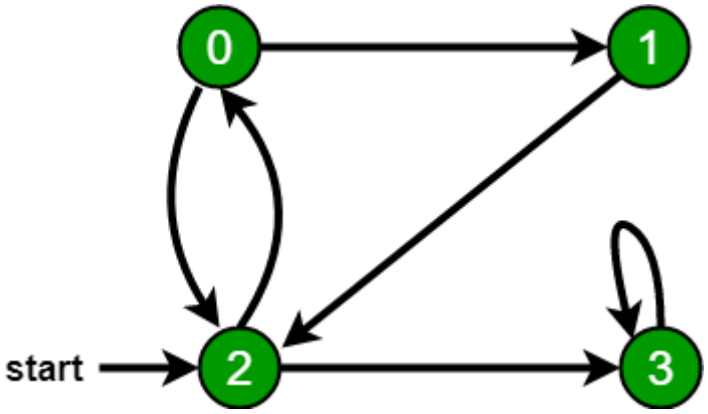
Visited and

Not visited.

A boolean visited array is used to mark the visited vertices. For simplicity, it is assumed that all vertices are reachable from the starting vertex. BFS uses a queue data structure for traversal.

Example:

In the following graph, we start traversal from vertex 2.



When we come to vertex 0, we look for all adjacent vertices of it.

2 is also an adjacent vertex of 0.

If we don't mark visited vertices, then 2 will be processed again and it will become a non-terminating process.

There can be multiple BFS traversals for a graph. Different BFS traversals for the above graph:

2, 3, 0, 1

2, 0, 3, 1

#include <stdbool.h>

#include <stdio.h>

```
#include <stdlib.h>
 #define MAX_VERTICES 50
 // This struct represents a directed graph using
// adjacency list representation
typedef struct Graph_t {
     // No. of vertices
    int V;
    bool adj[MAX_VERTICES][MAX_VERTICES];
} Graph;
 // Constructor
Graph* Graph_create(int V)
{
    Graph* g = malloc(sizeof(Graph));
    g->V = V;
     for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            g->adj[i][j] = false;
        }
    }
     return g;
}
 // Destructor
void Graph_destroy(Graph* g) { free(g); }
 // Function to add an edge to graph
void Graph_addEdge(Graph* g, int v, int w)
{
    // Add w to v's list.
    g->adj[v][w] = true;
}
 // Prints BFS traversal from a given source s
```

```
void Graph_BFS(Graph* g, int s)
{
    // Mark all the vertices as not visited
    bool visited[MAX_VERTICES];
    for (int i = 0; i < g->V; i++) {
        visited[i] = false;
    }
     // Create a queue for BFS
    int queue[MAX_VERTICES];
    int front = 0, rear = 0;
     // Mark the current node as visited and enqueue it
    visited[s] = true;
    queue[rear++] = s;
     while (front != rear) {
         // Dequeue a vertex from queue and print it
        s = queue[front++];
        printf("%d ", s);
         // Get all adjacent vertices of the dequeued
        // vertex s.
        // If an adjacent has not been visited,
        // then mark it visited and enqueue it
        for (int adjacent = 0; adjacent < g->V;
             adjacent++) {
            if (g->adj[s][adjacent] && !visited[adjacent]) {
                visited[adjacent] = true;
                queue[rear++] = adjacent;
            }
        }
    }
}
```

```
 // Driver code
int main()
{
    // Create a graph
    Graph* g = Graph_create(4);
    Graph_addEdge(g, 0, 1);
    Graph_addEdge(g, 0, 2);
    Graph_addEdge(g, 1, 2);
    Graph_addEdge(g, 2, 0);
    Graph_addEdge(g, 2, 3);
    Graph_addEdge(g, 3, 3);
     printf("Following is Breadth First Traversal "
          "(starting from vertex 2) \n");
    Graph_BFS(g, 2);
    Graph_destroy(g);
     return 0;
}
```

Output

```
Following is Breadth First Traversal (starting from vertex 2)
2 0 3 1
```

Experiment No. 07: Write a Program to find Optimal Binary Search Tree using Dynamic Method in C.

An Optimal Binary Search Tree (OBST), also known as a Weighted Binary Search Tree, is a binary search tree that minimizes the expected search cost. In a binary search tree, the search cost is the number of comparisons required to search for a given key.

In an OBST, each node is assigned a weight that represents the probability of the key being searched for. The sum of all the weights in the tree is 1.0. The expected search cost of a node is the sum of the product of its depth and weight, and the expected search cost of its children.

To construct an OBST, we start with a sorted list of keys and their probabilities. We then build a table that contains the expected search cost for all possible sub-trees of the original list. We can use dynamic programming to fill in this table efficiently. Finally, we use this table to construct the OBST.

```
// Dynamic Programming code for Optimal Binary Search

// Tree Problem

#include <stdio.h>

#include <limits.h>

// A utility function to get sum of array elements

// freq[i] to freq[j]

int sum(int freq[], int i, int j);

 /* A Dynamic Programming based function that calculates

  minimum cost of a Binary Search Tree. */

int optimalSearchTree(int keys[], int freq[], int n)

{

    /* Create an auxiliary 2D matrix to store results

      of subproblems */

    int cost[n][n];

     /* cost[i][j] = Optimal cost of binary search tree

       that can be  formed from keys[i] to keys[j].

       cost[0][n-1] will store the resultant cost */

     // For a single key, cost is equal to frequency of the key

    for (int i = 0; i < n; i++)

        cost[i][i] = freq[i];

     // Now we need to consider chains of length 2, 3, ... .
```

```
    // L is chain length.
    for (int L=2; L<=n; L++)
    {
        // i is row number in cost[][]
        for (int i=0; i<=n-L+1; i++)
        {
            // Get column number j from row number i and
            // chain length L
            int j = i+L-1;
            int off_set_sum = sum(freq, i, j);
            cost[i][j] = INT_MAX;

            // Try making all keys in interval keys[i..j] as root
            for (int r=i; r<=j; r++)
            {
              // c = cost when keys[r] becomes root of this subtree
              int c = ((r > i)? cost[i][r-1]:0) +
                      ((r < j)? cost[r+1][j]:0) +
                      off_set_sum;
              if (c < cost[i][j])
                 cost[i][j] = c;
            }
        }
    }
    return cost[0][n-1];
}
 // A utility function to get sum of array elements
// freq[i] to freq[j]
int sum(int freq[], int i, int j)
{
```

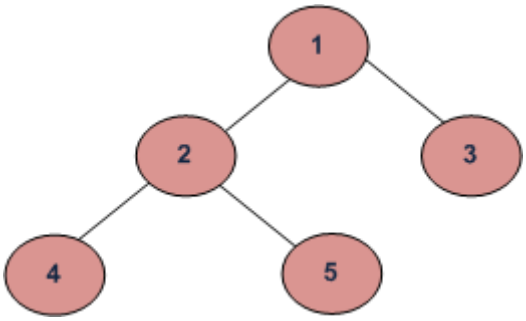
```
    int s = 0;
    for (int k = i; k <=j; k++)
       s += freq[k];
    return s;
}
 // Driver program to test above functions
int main()
{
    int keys[] = {10, 12, 20};
    int freq[] = {34, 8, 50};
    int n = sizeof(keys)/sizeof(keys[0]);
    printf("Cost of Optimal BST is %d ",
                optimalSearchTree(keys, freq, n));
    return 0;
}
```

Output

Cost of Optimal BST is 142

Experiment No. 08: Write a Program in C to find the Maximum Depth or Height of given Binary Tree.

Given a binary tree, the task is to find the height of the tree. Height of the tree is the number of edges in the tree from the root to the deepest node, Height of the empty tree is 0.



Recursively calculate height of left and right subtrees of a node and assign height to the node as max of the heights of two children plus 1.

```
#include <stdio.h>
#include <stdlib.h>
/* A binary tree node has data, pointer to left child
and a pointer to right child */
struct node {
        int data;
        struct node* left;
        struct node* right;
};
/* Compute the "maxDepth" of a tree -- the number of
        nodes along the longest path from the root node
        down to the farthest leaf node.*/
int maxDepth(struct node* node)
{
        if (node == NULL)
                return 0;
        else {
                /* compute the depth of each subtree */
                int lDepth = maxDepth(node->left);
                int rDepth = maxDepth(node->right);
```

```
                /* use the larger one */
                if (lDepth > rDepth)
                        return (lDepth + 1);
                else
                        return (rDepth + 1);
        }
}
/* Helper function that allocates a new node with the
given data and NULL left and right pointers. */
struct node* newNode(int data)
{
        struct node* node
                = (struct node*)malloc(sizeof(struct node));
        node->data = data;
        node->left = NULL;
        node->right = NULL;
        return (node);
}
int main()
{
        struct node* root = newNode(1);
        root->left = newNode(2);
        root->right = newNode(3);
        root->left->left = newNode(4);
        root->left->right = newNode(5);
        printf("Height of tree is %d", maxDepth(root));
        getchar();
        return 0;
}
```

Output Height of tree is 3

Experiment No. 09: Write a C Program To Perform Insertion, Deletion and Traversal In B-Tree.

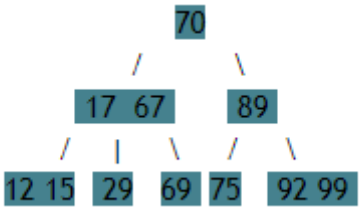
B- tree is a multiway search tree. A node in B-tree of order n can have at most n-1 values and n children.

All values that appear on the left sub-tree are smaller than left most value in the parent node.

All values that appear on the right sub-tree are greater than right most value in the parent node.

All values that appear on the middle sub-tree are greater than leftmost value in parent node and smaller than right most value in parent node.

Example for B-tree of Order 3:



Above is an example for B-Tree of order 3.

An intermediate node can have 2 or 3 children.

Any node can have at most 1 or 2 values.

```
#include <stdio.h>
 #include <stdlib.h>
 #define MAX 4
 #define MIN 2
 struct btreeNode {
      int val[MAX + 1], count;
      struct btreeNode *link[MAX + 1];
 };
 struct btreeNode *root;
 /* creating new node */
 struct btreeNode * createNode(int val, struct btreeNode *child) {
      struct btreeNode *newNode;
      newNode = (struct btreeNode *)malloc(sizeof(struct btreeNode));
      newNode->val[1] = val;
      newNode->count = 1;
```

```
    newNode->link[0] = root;
    newNode->link[1] = child;
    return newNode;
}
/* Places the value in appropriate position */
void addValToNode(int val, int pos, struct btreeNode *node,
                  struct btreeNode *child) {
    int j = node->count;
    while (j > pos) {
        node->val[j + 1] = node->val[j];
        node->link[j + 1] = node->link[j];
        j--;
    }
    node->val[j + 1] = val;
    node->link[j + 1] = child;
    node->count++;
}
/* split the node */
void splitNode (int val, int *pval, int pos, struct btreeNode *node,
  struct btreeNode *child, struct btreeNode **newNode) {
    int median, j;
    if (pos > MIN)
        median = MIN + 1;
    else
        median = MIN;
    *newNode = (struct btreeNode *)malloc(sizeof(struct btreeNode));
    j = median + 1;
    while (j <= MAX) {
        (*newNode)->val[j - median] = node->val[j];
        (*newNode)->link[j - median] = node->link[j];
```

```
            j++;
        }
        node->count = median;
        (*newNode)->count = MAX - median;
        if (pos <= MIN) {
            addValToNode(val, pos, node, child);
        } else {
            addValToNode(val, pos - median, *newNode, child);
        }
        *pval = node->val[node->count];
        (*newNode)->link[0] = node->link[node->count];
        node->count--;
}
/* sets the value val in the node */
int setValueInNode(int val, int *pval,
  struct btreeNode *node, struct btreeNode **child) {
        int pos;
        if (!node) {
            *pval = val;
            *child = NULL;
            return 1;
        }
        if (val < node->val[1]) {
            pos = 0;
        } else {
            for (pos = node->count;
                (val < node->val[pos] && pos > 1); pos--);
            if (val == node->val[pos]) {
                printf("Duplicates not allowed\n");
                return 0;
```

```
            }
        }
        if (setValueInNode(val, pval, node->link[pos], child)) {
            if (node->count < MAX) {
                addValToNode(*pval, pos, node, *child);
            } else {
                splitNode(*pval, pval, pos, node, *child, child);
                return 1;
            }
        }
        return 0;
}
/* insert val in B-Tree */
void insertion(int val) {
        int flag, i;
        struct btreeNode *child;
        flag = setValueInNode(val, &i, root, &child);
        if (flag)
            root = createNode(i, child);
}
/* copy successor for the value to be deleted */
void copySuccessor(struct btreeNode *myNode, int pos) {
        struct btreeNode *dummy;
        dummy = myNode->link[pos];
        for (;dummy->link[0] != NULL;)
            dummy = dummy->link[0];
        myNode->val[pos] = dummy->val[1];

}
/* removes the value from the given node and rearrange values */
```

```
void removeVal(struct btreeNode *myNode, int pos) {
    int i = pos + 1;
    while (i <= myNode->count) {
        myNode->val[i - 1] = myNode->val[i];
        myNode->link[i - 1] = myNode->link[i];
        i++;
    }
    myNode->count--;
}
/* shifts value from parent to right child */
void doRightShift(struct btreeNode *myNode, int pos) {
    struct btreeNode *x = myNode->link[pos];
    int j = x->count;
    while (j > 0) {
        x->val[j + 1] = x->val[j];
        x->link[j + 1] = x->link[j];
    }
    x->val[1] = myNode->val[pos];
    x->link[1] = x->link[0];
    x->count++;
    x = myNode->link[pos - 1];
    myNode->val[pos] = x->val[x->count];
    myNode->link[pos] = x->link[x->count];
    x->count--;
    return;
}
/* shifts value from parent to left child */
void doLeftShift(struct btreeNode *myNode, int pos) {
    int j = 1;
    struct btreeNode *x = myNode->link[pos - 1];
```

```
    x->count++;
    x->val[x->count] = myNode->val[pos];
    x->link[x->count] = myNode->link[pos]->link[0];
    x = myNode->link[pos];
    myNode->val[pos] = x->val[1];
    x->link[0] = x->link[1];
    x->count--;
    while (j <= x->count) {
        x->val[j] = x->val[j + 1];
        x->link[j] = x->link[j + 1];
        j++;
    }
    return;
}
/* merge nodes */
void mergeNodes(struct btreeNode *myNode, int pos) {
    int j = 1;
    struct btreeNode *x1 = myNode->link[pos], *x2 = myNode->link[pos - 1];
    x2->count++;
    x2->val[x2->count] = myNode->val[pos];
    x2->link[x2->count] = myNode->link[0];
    while (j <= x1->count) {
        x2->count++;
        x2->val[x2->count] = x1->val[j];
        x2->link[x2->count] = x1->link[j];
        j++;
    }
    j = pos;
    while (j < myNode->count) {
        myNode->val[j] = myNode->val[j + 1];
```

```
            myNode->link[j] = myNode->link[j + 1];
            j++;
        }
        myNode->count--;
        free(x1);
}
/* adjusts the given node */
void adjustNode(struct btreeNode *myNode, int pos) {
        if (!pos) {
            if (myNode->link[1]->count > MIN) {
                doLeftShift(myNode, 1);
            } else {
                mergeNodes(myNode, 1);
            }
        } else {
            if (myNode->count != pos) {
                if(myNode->link[pos - 1]->count > MIN) {
                    doRightShift(myNode, pos);
                } else {
                    if (myNode->link[pos + 1]->count > MIN) {
                        doLeftShift(myNode, pos + 1);
                    } else {
                        mergeNodes(myNode, pos);
                    }
                }
            } else {
                if (myNode->link[pos - 1]->count > MIN)
                    doRightShift(myNode, pos);
                else
                    mergeNodes(myNode, pos);
```

```
            }
        }
}
/* delete val from the node */
int delValFromNode(int val, struct btreeNode *myNode) {
    int pos, flag = 0;
    if (myNode) {
        if (val < myNode->val[1]) {
            pos = 0;
            flag = 0;
        } else {
            for (pos = myNode->count;
                (val < myNode->val[pos] && pos > 1); pos--);
             if (val == myNode->val[pos]) {
                flag = 1;
            } else {
                flag = 0;
            }
        }
        if (flag) {
            if (myNode->link[pos - 1]) {
                copySuccessor(myNode, pos);
                flag = delValFromNode(myNode->val[pos], myNode->link[pos]);
                if (flag == 0) {
                    printf("Given data is not present in B-Tree\n");
                }
            } else {
                removeVal(myNode, pos);
            }
        } else {
```

```
            flag = delValFromNode(val, myNode->link[pos]);
        }
        if (myNode->link[pos]) {
            if (myNode->link[pos]->count < MIN)
                adjustNode(myNode, pos);
        }
    }
    return flag;
}
/* delete val from B-tree */
void deletion(int val, struct btreeNode *myNode) {
    struct btreeNode *tmp;
    if (!delValFromNode(val, myNode)) {
        printf("Given value is not present in B-Tree\n");
        return;
    } else {
        if (myNode->count == 0) {
            tmp = myNode;
            myNode = myNode->link[0];
            free(tmp);
        }
    }
    root = myNode;
    return;
}
/* search val in B-Tree */
void searching(int val, int *pos, struct btreeNode *myNode) {
    if (!myNode) {
        return;
    }
```

```
    if (val < myNode->val[1]) {
        *pos = 0;
    } else {
        for (*pos = myNode->count;
            (val < myNode->val[*pos] && *pos > 1); (*pos)--);
        if (val == myNode->val[*pos]) {
            printf("Given data %d is present in B-Tree", val);
            return;
        }
    }
    searching(val, pos, myNode->link[*pos]);
    return;
}
/* B-Tree Traversal */
void traversal(struct btreeNode *myNode) {
    int i;
    if (myNode) {
        for (i = 0; i < myNode->count; i++) {
            traversal(myNode->link[i]);
            printf("%d ", myNode->val[i + 1]);
        }
        traversal(myNode->link[i]);
    }
}
int main() {
    int val, ch;
    while (1) {
        printf("1. Insertion\t2. Deletion\n");
        printf("3. Searching\t4. Traversal\n");
        printf("5. Exit\nEnter your choice:");
```

```
            scanf("%d", &ch);
            switch (ch) {
                case 1:
                    printf("Enter your input:");
                    scanf("%d", &val);
                    insertion(val);
                    break;
                case 2:
                    printf("Enter the element to delete:");
                    scanf("%d", &val);
                    deletion(val, root);
                    break;
                case 3:
                    printf("Enter the element to search:");
                    scanf("%d", &val);
                    searching(val, &ch, root);
                    break;
                case 4:
                    traversal(root);
                    break;
                case 5:
                    exit(0);
                default:
                    printf("U have entered wrong option!!\n");
                    break;
            }
            printf("\n");
        }
    }
```

1. Insertion 2. Deletion

3. Searching 4. Traversal

5. Exit

Enter your choice:1

Enter your input:70

1. Insertion 2. Deletion

3. Searching 4. Traversal

5. Exit

Experiment No. 10: Write a Program in C for Random access file to read or write any data in our disk file without reading or writing every piece of data before it.

In C, the data stored in a file can be accessed in the following ways:

Sequential Access

Random Access

If the file size is too huge, sequential access is not the best option for reading the record in the middle of the file. Random access to a file can be employed in this situation, enabling access to any record at any point in the file. We can imagine data in a random access file as songs on a compact disc or record; we can fast forward directly to any song we want without playing the other pieces. We can do so if we're playing the first song, the sixth song, the fourth song. This order has nothing to do with the songs' order initially recorded. Random file access sometimes takes more programming but rewards our effort with a more flexible file-access method. Thus, there are three functions which help in using the random access file in C:

fseek()

ftell()

rewind()

Now let us see the code in C:

```
#include<stdio.h>
int main()
{
   FILE *fp;
   fp=fopen("scaler.txt","r");
   if(!fp)
   {
      printf("Error: File cannot be opened\n") ;
      return 0;
   }
      //Since the file pointer points to the starting of the file, ftell() will return 0
   printf("Position pointer in the beginning : %ld\n",ftell(fp));
      char ch;
   while(fread(&ch,sizeof(ch),1,fp)==1)
   {
      //Here, we traverse the entire file and print its contents until we reach its end.
```

```
        printf("%c",ch);
    }
        printf("\nSize of file in bytes is : %ld\n",ftell(fp));
    fclose(fp);
    return 0;
}
```

Output:

```
Position pointer in the beginning : 0
Scaler is amazing
Size of file in bytes is : 17
```

Experiment No. 11: Case Study- Design a mini project to implement a Smart text editor.

ALGORITHM:

1.Display options new, open and exit and get choice.

2.If choice is 1 , call Create() function.

3.If choice is 2, call Display() function.

4.If choice is 3, call Append() function.

5.If choice is 4, call Delete() function.

6.If choice is 5, call Display() function.

7.Create()

7.1 Get the file name and open it in write mode.

7.2 Get the text from the user to write it.

8.Display()

8.1 Get the file name from user.

8.2 Check whether the file is present or not.

8.2 If present then display the contents of the file.

9.Append()

9.1 Get the file name from user.

9.2 Check whether the file is present or not.

9.3 If present then append the file by getting the text to add with the existing file.

10. Delete()

10.1 Get the file name from user.

10.2 Check whether the file is present or not.

10.3 If present then delete the existing file.

SOURCE CODE:

```
#include<stdio.h>
#include<conio.h>
#include<process.h>
int i,j,ec,fg,ec2;
char fn[20],e,c;
FILE *fp1,*fp2,*fp;
```

```
void Create();
void Append();
void Delete();
void Display();
void main()
{
 do {
  clrscr();
  printf("\n\t\t***** TEXT EDITOR *****");
  printf("\n\n\tMENU:\n\t-----\n ");
  printf("\n\t1.CREATE\n\t2.DISPLAY\n\t3.APPEND\n\t4.DELETE\n\t5.EXIT\n");
  printf("\n\tEnter your choice: ");
  scanf("%d",&ec);
  switch(ec)
  {
   case 1:
     Create();
     break;
   case 2:
     Display();
     break;
   case 3:
     Append();
     break;
   case 4:
     Delete();
     break;
   case 5:
     exit(0);
  }
```

```
 }while(1);
}
void Create()
{
 fp1=fopen("temp.txt","w");
 printf("\n\tEnter the text and press '.' to save\n\n\t");
 while(1)
 {
  c=getchar();
  fputc(c,fp1);
  if(c == '.')
  {
   fclose(fp1);
   printf("\n\tEnter then new filename: ");
   scanf("%s",fn);
   fp1=fopen("temp.txt","r");
   fp2=fopen(fn,"w");
   while(!feof(fp1))
   {
    c=getc(fp1);
    putc(c,fp2);
   }
   fclose(fp2);
   break;
  }}
}
void Display()
{
  printf("\n\tEnter the file name: ");
  scanf("%s",fn);
```

```
  fp1=fopen(fn,"r");
  if(fp1==NULL)
  {
   printf("\n\tFile not found!");
   goto end1;
  }
  while(!feof(fp1))
  {
   c=getc(fp1);
   printf("%c",c);
  }
end1:
  fclose(fp1);
  printf("\n\n\tPress any key to continue...");
  getch();
}
void Delete()
{
  printf("\n\tEnter the file name: ");
  scanf("%s",fn);
  fp1=fopen(fn,"r");
  if(fp1==NULL)
  {
   printf("\n\tFile not found!");
   goto end2;
  }
  fclose(fp1);
  if(remove(fn)==0)
  {
   printf("\n\n\tFile has been deleted successfully!");
```

```
   goto end2;
  }
  else
   printf("\n\tError!\n");
end2: printf("\n\n\tPress any key to continue...");
  getch();
}
void Append()
{
  printf("\n\tEnter the file name: ");
  scanf("%s",fn);
  fp1=fopen(fn,"r");
  if(fp1==NULL)
  {
   printf("\n\tFile not found!");
   goto end3;
  }
  while(!feof(fp1))
  {
   c=getc(fp1);
   printf("%c",c);
  }
  fclose(fp1);
  printf("\n\tType the text and press 'Ctrl+S' to append.\n");
  fp1=fopen(fn,"a");
  while(1)
  {
   c=getch();
   if(c==19)
    goto end3;
```

```
  if(c==13)
  {
   c='\n';
   printf("\n\t");
   fputc(c,fp1);
  }
  else
  {
   printf("%c",c);
   fputc(c,fp1);
  }
 }
end3: fclose(fp1);
 getch();
}
```

OUTPUT:

FILE CREATION: