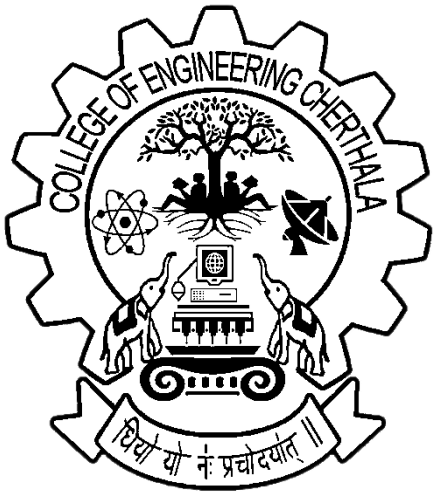
COLLEGE OF ENGINEERING CHERTHALA

LAB RECORD

20MCA135 – DATA STRUCTURES LAB



CERTIFICATE

This is certified to be bonafide works of Mr./Ms. ……..……., In the class ……...., Reg. No. ……., of College of Engineering Cherthala, during the academic year 2022-23.

Teacher In Charge **External Examiner**

Internal Examiner

INDEX

Program No.1

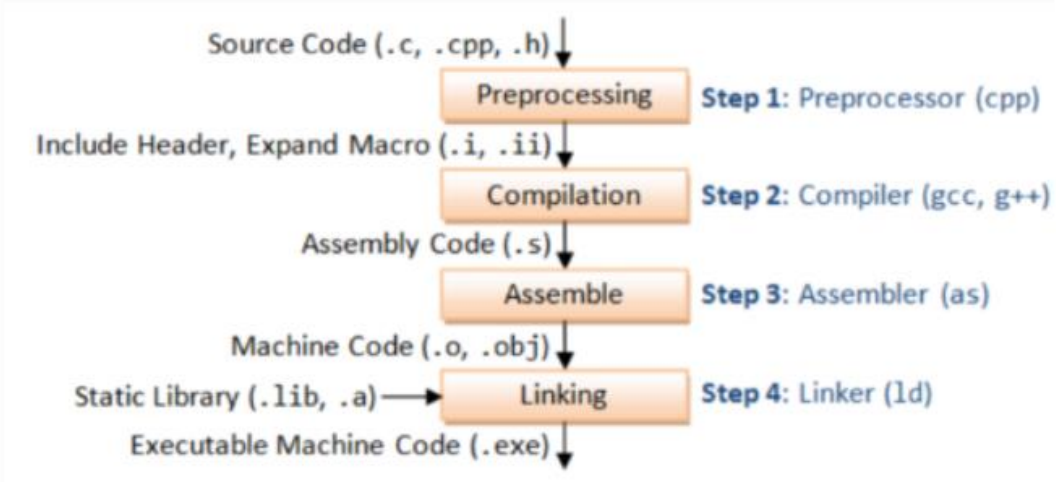
ADVANCED USE OF GCC

Aim: Advanced use of GCC.

Creating an executable is a multistage process divided into two components: compilation and linking. Even if a program "compiles fine" it might not actually work because of errors during the linking phase.

Compilation refers to the processing of source code files (.c, .cc, or .cpp) and the creation of an 'object' file(.o, .obj files). This step doesn't create anything the user can actually run. Instead, the compiler merely produces the machine language instructions that correspond to the source code file that was compiled. You need to turn them into executables your operating system can use. That's where the linker comes in.

Linking refers to the creation of a single executable file from multiple object files. Compiler doesn't look at the contents of more than one file at a time. The linker, on the other hand, may look at multiple files and try to find references for the functions that weren't mentioned.



GCC Compiler options:

-o option: Specify the Output Executable Name

In its most basic form, gcc compiler can be used as :

gcc main.c

The above command executes the complete compilation process and outputs an executable with name a.out.

Use option -o, as shown below, to specify the output file name for the executable.

gcc main.c -o main

The command above would produce an output file with name ‘main’.

-Wall option : Enable all warnings set through -Wall option

This option enables all the warnings in GCC.

```
#include<stdio.h>;
int main(void)
{
int i;
printf(“\n The Geek Stuff [%d]\n”, i);
return 0;
}
```

If the above code is compiled, the following warning related to uninitialized variable i is produced :

```
In function ‘main’:
main.c:5:4: warning: ‘i’ is used uninitialized [-Wuninitialized]
5 | printf(“\n The Geek Stuff [%d]\n”, i);
```

-E option: Produce only the preprocessor output with -E option

The output of preprocessing stage can be produced using the -E option.

$ gcc -E main.c > main.i

The gcc command produces the output on stdout so you can redirect the output in any file. In our case(above), the file main.i would contain the preprocessed output.

-S option: Produce only the assembly code using -S option

The assembly level output can be produced using the -S option.

gcc -S main.c > main.s

In this case, the file main.s would contain the assembly output.

-C option: Produce only the compiled code using the -C option

To produce only the compiled code (without any linking), use the -C option.

gcc -C main.c

The command above would produce a file main.o that would contain machine level code or the compiled code.

-save-temps option: Produce all the intermediate files using -save-temps function

The option -save-temps can do all the work done above. Through this option, output at all the stages of compilation is stored in the current directory. Please note that this option produces the executable also.

$ gcc -save-temps main.c

$ ls

a.out main.c main.i main.o main.s

So we see that all the intermediate files as well as the final executable was produced in the output.

-

-l option: Link with shared (system) libraries using -l option

The option -l can be used to link with shared libraries. For example:

gcc -Wall main.c -o main -lCPPfile

The gcc command mentioned above links the code main.c with the shared library libCPPfile.so as to produce the final executable 'main'.

For example, I recently installed FFTW on my computer. It added libfftw3.a to my /usr/local/lib directory. To link with this library I type:

gcc main.cc -lfftw3

Result: Familiarised with GCC options.

Program No:2

ADVANCED USE OF GPROF

Aim: Familiarization of Gprof options.

Profiling is an important aspect of software programming. Through profiling one can determine the parts in program code that are time consuming and need to be re-written. This helps make your program execution faster which is always desired.

In very large projects, profiling can save your day by not only determining the parts in your program which are slower in execution than expected but also can help you find many other statistics through which many potential bugs can be spotted and sorted out.

// test_gprof.c

```
#include<stdio.h>

int main(void)

{

 printf("\n Inside main()\n");

 int i;

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

printf("hello");


 return 0;

}
```

Step-1 : Profiling enabled while compilation

In this first step, we need to make sure that the profiling is enabled when the compilation of the code is done. This is made possible by adding the '-pg' option in the compilation step.

So, lets compile our code with '-pg' option :

$ gcc -Wall -pg test_ gprof.c test_gprof_new.c -o test_gprof

Step-2 : Execute the code

In the second step, the binary file produced as a result of step-1 (above) is executed so that profiling information can be generated.

$ ls

test_gprof test_gprof.c test_gprof_new.c

$ ./test_gprof

Inside main()

Inside func1

Inside new_func1()

Inside func2

$ ls

gmon.out test_gprof test_gprof.c test_gprof_new.c

$

So we see that when the binary was executed, a new file ‘gmon.out’ is generated in the current working directory.

Step-3 : Run the gprof tool

In this step, the gprof tool is run with the executable name and the above generated ‘gmon.out’ as argument. This produces an analysis file which contains all the desired profiling information.

$ gprof test_gprof gmon.out > analysis.txt

Note that one can explicitly specify the output file (like in example above) or the information is produced on stdout.

$ ls

analysis.txt gmon.out test_gprof test_gprof.c test_gprof_new.c

So we see that a file named ‘analysis.txt’ was generated, which is broadly divided into two parts :

1. Flat profile

2. Call graph

Comprehending the profiling information

Flat profile :

Each sample counts as 0.01 seconds.

% cumulative self self total

time seconds seconds calls s/call s/call name

33.86 15.52 15.52 1 15.52 15.52 func2

33.82 31.02 15.50 1 15.50 15.50 new_func1

33.29 46.27 15.26 1 15.26 30.75 func1

0.07 46.30 0.03 main

Flat profile (explanation follows) :

% the percentage of the total running time of the time program used by this function.

cumulative a running sum of the number of seconds accounted seconds for by this function and those listed above it.

self the number of seconds accounted for by this seconds function alone. This is the major sort for this listing.

calls the number of times this function was invoked, if this function is profiled, else blank.

self the average number of milliseconds spent in this ms/call function per call, if this function is profiled, else blank.

total the average number of milliseconds spent in this ms/call function and its descendents per call, if this function is profiled, else blank.

name the name of the function. This is the minor sort for this listing. The index shows the location of the function in the gprof listing. If the index is in parenthesis it shows where it would appear in the gprof listing if it were to be printed.

Call graph :

granularity: each sample hit covers 2 byte(s) for 0.02% of 46.30 seconds

index % time self children called name
[1] 100.0 0.03 46.27 main [1]

15.26 15.50 1/1 func1 [2]

15.52 0.00 1/1 func2 [3]

15.26 15.50 1/1 main [1]

[2] 66.4 15.26 15.50 1 func1 [2]

15.50 0.00 1/1 new_func1 [4]

15.52 0.00 1/1 main [1]

[3] 33.5 15.52 0.00 1 func2 [3]

15.50 0.00 1/1 func1 [2]

[4] 33.5 15.50 0.00 1 new_func1 [4]

Call graph (explanation follows) :

This table describes the call tree of the program, and was sorted by the total amount of time spent in each function and its children.

Each entry in this table consists of several lines. The line with the index number at the left hand margin lists the current function.

The lines above it list the functions that called this function,and the lines below it list the functions this one called.

This line lists:

index A unique number given to each element of the table. Index numbers are sorted numerically.

The index number is printed next to every function name so it is easier to look up where the function in the table.

% time This is the percentage of the `total' time that was spent in this function and its children. Note that due to different viewpoints, functions excluded by options, etc, these numbers will NOT add up to 100%.

self This is the total amount of time spent in this function.

children This is the total amount of time propagated into this function by its children.

called This is the number of times the function was called.

If the function called itself recursively, the number only includes non-recursive calls, and is followed by a `+' and the number of recursive calls.

name The name of the current function. The index number is printed after it. If the function is a member of a cycle, the cycle number is printed between the function's name and the index number.

For the function's parents, the fields have the following meanings:

self This is the amount of time that was propagated directly from the function into this parent.

children This is the amount of time that was propagated from the function's children into this parent.

called This is the number of times this parent called the function `/' the total number of times the function was called. Recursive calls to the function are not included in the number after the `/'.

name This is the name of the parent. The parent's index number is printed after it. If the parent is a member of a cycle, the cycle number is printed between the name and the index number.

If the parents of the function cannot be determined, the word `' is printed in the `name' field, and all the other fields are blank.

For the function's children, the fields have the following meanings:

self This is the amount of time that was propagated directly from the child into the function.

children This is the amount of time that was propagated from the child's children to the function.

called This is the number of times the function called this child `/' the total number of times the child was called.

Recursive calls by the child are not listed in the number after the `/'. name This is the name of the child. The child's index number is printed after it. If the child is a

member of a cycle, the cycle number is printed between the name and the index number.

If there are any cycles (circles) in the call graph, there is an entry for the cycle-as-a-whole. This entry shows who called the cycle (as parents) and the members of the cycle (as children.)

The `+' recursive calls entry shows the number of function calls that were internal to the cycle, and the calls entry for each member shows, for that member, how many times it was called from other members of the cycle.

Index by function name

[2] func1 [1] main

[3] func2 [4] new_func1

Customize gprof output using flags

There are various flags available to customize the output of the gprof tool. Some of them are discussed below:

1. Suppress the printing of statically(private) declared functions using -a

If there are some static functions whose profiling information you do not require then this can be achieved using -a option :

$ gprof -a test_gprof gmon.out > analysis.txt

2. Suppress verbose blurbs using -b

As you would have already seen that gprof produces output with lot of verbose information so in case this information is not required then this can be achieved using the -b flag.

$ gprof -b test_gprof gmon.out > analysis.txt

3. Print only flat profile using -p

In case only flat profile is required then :

$ gprof -p -b test_gprof gmon.out > analysis.txt

4. Print information related to specific function in flat profile

This can be achieved by providing the function name along with the -p option:

$ gprof -pfunc1 -b test_gprof gmon.out > analysis.txt

5. Suppress flat profile in output using -P

If flat profile is not required then it can be suppressed using the -P option :

$ gprof -P -b test_gprof gmon.out > analysis.txt

6. Print only call graph information using -q

gprof -q -b test_gprof gmon.out > analysis.txt

7. Print only specific function information in call graph.

This is possible by passing the function name along with the -q option.

$ gprof -qfunc1 -b test_gprof gmon.out > analysis.txt

8. Suppress call graph using -Q

If the call graph information is not required in the analysis output then -Q option can be used.

$ gprof -Q -b test_gprof gmon.out > analysis.txt

Result: Familiarised with Gprof options.

Program No:3

MERGE TWO SORTED ARRAYS AND STORE IN A THIRD ARRAY

Aim :- Merge two sorted arrays and store in a third array.

Algorithm :-

1. Start the program.
2. Input the length of both the arrays.
3. Input the arrays elements from user.
4. Copy the elements of the first array to the merged array when initialising it.
5. Copy the elements of the second array to the merged array while initialising the second array.
6. Sort the merged array now.
7. Display the merged array.
8. The program ends here

Program :-

```
#include<stdio.h>
int main(){
        int a[100],b[100],c[200],i,i1,j,j1,k,n,m,t,p;
        printf("Enter the first array limit :");
        scanf("%d",&n);
        printf("\nEnter the first array numbers\n");
        for(i=0;i<n;i++){
                scanf("%d",&a[i]);
        }
        for(i=0;i<n;i++){
                for(i1=i+1;i1<n;i1++){
                        if(a[i]>a[i1]){
```

```
                    t=a[i];
                    a[i]=a[i1];
                    a[i1]=t;
}
            }
        }
        for(i=0;i<n;i++){
            printf("%d\t",a[i]);
        }
        printf("\nEnter the second array limit :");
        scanf("%d",&m);
        printf("\nEnter the second array numbers\n");
        for(j=0;j<m;j++){
            scanf("%d",&b[j]);
        }
        for(j=0;j<m;j++){
            for(j1=j+1;j1<m;j1++){
                if(b[j]>b[j1]){
                    p=b[j];
                    b[j]=b[j1];
                    b[j1]=p;
                }
            }
        }
        for(j=0;j<m;j++){
            printf("%d\t",b[j]);
```

```
        }
        i=j=k=0;
        while(i<n&&j<m)
                if(a[i]>=b[j])
                        c[k++]=b[j++];
                else
                        c[k++]=a[i++];
        while(i<n)
                c[k++]=a[i++];
        while(j<m)
                c[k++]=b[j++];


        printf("\nAfter merging\n");
        for(k=0;k<n+m;k++){
                printf("%d\t",c[k]);
        }
}
```

Output :-

Enter the first array limit :3

Enter the first array numbers

5

4

6

4 5 6

Enter the second array limit :2

Enter the second array numbers

2

1

1 2

After merging

1 2 4 5 6

Result :-Program executed successfully and output verified.

Program No:4

STACK USING ARRAY

Aim :-To implement a stack using array

Algorithm :-

1. Define a fixed size for the stack and create an array of that size.
2. Initialize a variable to keep track of the top of the stack. Set it to -1 to indicate that the stack is initially empty.
3. Create a function to push an element onto the stack:
    a. Check if the stack is already full (i.e., if the top of the stack is equal to the size of the array minus 1).
    b. If the stack is not full, increment the top variable and add the new element to the array at the index of the new top value.
4. Create a function to pop an element off the stack:
    a. Check if the stack is empty (i.e., if the top of the stack is equal to -1).
    b. If the stack is not empty, remove the top element by returning the value at the index of the current top value and decrement the top variable.
5. Create a function to check if the stack is empty:
    a. Check if the top of the stack is equal to -1.
    b. If it is, return true. Otherwise, return false.
6. Create a function to check if the stack is full:
    a. Check if the top of the stack is equal to the size of the array minus 1.
    b. If it is, return true. Otherwise, return false.
7. Create a function to peek at the top element of the stack:
    a. Check if the stack is empty (using the function from step 5).
    b. If it is not empty, return the value at the index of the current top value.

Program :-

```
#include<stdio.h>

int S[100],element,size,top=-1,choice,i;

void push();

void pop();

void display();

int main()
```

```
{
        int choice=1;
        printf("\nSTACK WITH ARRAY");
        printf("\nEnter the size of the stack(max 100) :");
        scanf("%d",&size);
        while(choice<4&&choice!=0)
        {
                printf("\n\nPress 1 to PUSH element");
                printf("\n\nPress 2 to POP element");
                printf("\n\nPress 3 to DISPLAY elements");
                printf("\n\nEnter your choice :");
                scanf("%d",&choice);
                switch(choice)
                {
                        case 1:
                                push();
                                break;
                        case 2:
                                pop();
                                break;
                        case 3:
                                display();
                                break;
                }
        }
}
```

```
void push()
{
	if(top==size-1)
	{
		printf("\nStack is full!!");
	}
	else
	{
		printf("\nEnter the element to push :");
		scanf("%d",&element);
		top++;
		S[top]=element;
		printf("\nInserted :%d",element);
	}
}
void pop()
{
	if(top==-1){
		printf("\nStack is empty!!");
	}
	else
	{
		printf("Deleted :%d",S[top]);
		top--;
	}
```

```
}
void display()
{
        if(top>=0)
        {
                printf("\nStack elements are given below");
                for(i=top;i>=0;i--)
                {
                        printf("\n%d",S[i]);
                }
        }
        else
        {
                printf("\nStack is empty!!");
        }
}
```

Output :-

```
STACK WITH ARRAY
Enter the size of the stack(max 100) :5
Press 1 to PUSH element
Press 2 to POP element
Press 3 to DISPLAY elements
Enter your choice :1
Enter the element to push :10
Inserted :10
```

Enter your choice :1

Enter the element to push :20

Inserted :20

Enter your choice :3

Stack elements are given below

20

10

Enter your choice :2

Deleted :20

Enter your choice :3

Stack elements are given below

10

Enter your choice :2

Deleted :10

Enter your choice :3

Stack is empty!!

Result :-Program executed successfully and output verified.

Program No:5

QUEUE USING ARRAY

Aim :- To implement queue using array

Algorithm :-

1. Create an array of a fixed size to store the queue elements.
2. Initialize the front and rear indices of the queue to -1.
3. Define a function to check if the queue is empty, which returns true if the front index is equal to -1.
4. Define a function to check if the queue is full, which returns true if the rear index is equal to the size of the array minus one.
5. Define a function to add an element to the back of the queue, which performs the following steps:
    a. Check if the queue is full. If so, return an error message.
    b. Increment the rear index.
    c. Insert the new element at the rear index of the array.
    d. If this is the first element in the queue, set the front index to 0.
6. Define a function to remove an element from the front of the queue, which performs the following steps:
    a. Check if the queue is empty. If so, return an error message.
    b. Store the element at the front index in a temporary variable.
    c. Increment the front index.
    d. If the front index is now equal to the rear index, the queue is empty, so set both indices to -1.
    e. Return the stored element.
7. Define a function to peek at the element at the front of the queue without removing it, which performs the following steps:
    a. Check if the queue is empty. If so, return an error message.
    b. Return the element at the front index of the array.
8. Define a function to display the contents of the queue, which performs the following steps:
    a. Check if the queue is empty. If so, return an error message.
    b. Iterate over the array from the front index to the rear index, printing each element.
9. Implement any additional functions as needed for your specific use case, such as a function to get the size of the queue.

Program :-

```
#include<stdio.h>
#include<stdlib.h>
int Q[100],size,element,front=-1,rear=-1,i,choice=1;
void enqueue();
void dequeue();
void display();
void search();
int main()
{
        printf("\nQUEUE");
        printf("\n\nEnter the size of the queue(max 100 :");
        scanf("%d",&size);
printf("\n1.Insert \n2.Delete \n3.Display \n4.Search");
        while(choice<5&&choice!=0)
        {

                printf("\nEnter your choice :");
                scanf("%d",&choice);
                switch(choice)
                {
                        case 1:
                                enqueue();
                                break;
                        case 2:
                                dequeue();
```

```
                    break;
                case 3:
                    display();
                    break;
                case 4:
                    search();
                    break;
                default:
                    printf("\nEnter valid choice");
            }
        }
}
void enqueue()
{
        if(rear==size-1)
        {
            printf("\nQueue is full!!");
        }
        else if(front==-1&&rear==-1)
        {
            front=0;
            rear=0;
            printf("\nEnter the element to insert :");
            scanf("%d",&element);
            Q[rear]=element;
            printf("\nInserted %d",element);
```

```
        }
        else
        {
                rear++;
                printf("\nEnter the element to insert :");
                scanf("%d",&element);
                Q[rear]=element;
                printf("\nInserted %d",element);

        }
}
void dequeue()
{
        if(front==-1&&rear==-1)
        {
                printf("\nQueue is empty!!");
        }
        else if(front==rear)
        {
                element=Q[front];
                printf("\nDeleted %d",element);
                front=-1;
                rear=-1;
        }
        else
        {
```

```
            element=Q[front];
            printf("\nDeleted %d",Q[front]);
            front++;
        }
}
void display()
{
        if(front==-1&&rear==-1)
        {
            printf("\nQueue is empty!!");
        }
        else
        {
            printf("\nFront :%d\n",front);
            for(i=front;i<=rear;i++)
            {
                printf("%d\t",Q[i]);
            }
            printf("\nRear :%d",rear);
        }
}
void search()
{
        if(front!=-1&&rear!=-1)
        {
            printf("\nEnter the elment to search :");
```

```
            scanf("%d",&element);
            for(i=front;i<=rear;i++)
            {
                if(element==Q[i])
                printf("\nElement %d found in location %d",element,i);
            }
        }
        else
        {
            printf("\nQueue is empty!!");
        }
}
```

Output :-

```
QUEUE
Enter the size of the queue(max 100 :5
1.Insert
2.Delete
3.Display
4.Search
Enter your choice :1
Enter the element to insert :13
Inserted 13
Enter your choice :1
Enter the element to insert :26
Inserted 26
Enter your choice :3
```

```
Front :0
13      26
Rear :1
Enter your choice :2
Deleted 13
Enter your choice :3
Front :1
26
Rear :1
Enter your choice :2
Deleted 26
Enter your choice :3
Queue is empty!!
```

Result :-Program executed successfully and output verified.

Program No:6

CIRCULAR QUEUE USING ARRAY

Aim :- To implement circular queue using array

Algorithm :-

1. Check if the queue is full (Rear + 1 % Maxsize = Front)

2. If the queue is full, there will be an Overflow error

3. Check if the queue is empty, and set both Front and Rear to 0

4. If Rear = Maxsize - 1 &Front != 0 (rear pointer is at the end of the queue and front is not at 0th index), then set Rear = 0

5. Otherwise, set Rear = (Rear + 1) % Maxsize

6. Insert the element into the queue (Queue[Rear] = x)

7. Exit

8. Check if the queue is empty (Front = -1 & Rear = -1)

9. If the queue is empty, Underflow error

10.Set Element = Queue[Front]

11.If there is only one element in a queue, set both Front and Rear to -1 (IF Front = Rear, set Front = Rear = -1)

12.And if Front = Maxsize -1 set Front = 0

13.Otherwise, set Front = Front + 1

14.Exit

15.Search an element from queue,do:

16. Check if the element is in queue,if it is in,then return the element

17.Else print “not found”

18.Exit

Program :-

```
#include<stdio.h>
#define size 5
int Q[size];
int front=-1,rear=-1;
int isfull()
{
        if((front==rear+1)||(front==0&&rear==size-1))return 1;
        return 0;
}
int isempty()
{
        if(front==-1)return 1;
        return 0;
}
void enqueue(int element)
{
        if(isfull())
        printf("\nQueue is full\n");

        else
        {
                if(front==-1)
                front=0;
                rear=(rear+1)%size;
                Q[rear]=element;
```

```
            printf("\n Inserted :%d",element);
        }
}
int dequeue()
{
        int element;
        if(isempty())
        {
                printf("\nQueue is Empty!!\n");
                return(-1);
        }
        else
        {
                element=Q[front];
                if(front==rear)
                {
                        front=-1;
                        rear=-1;
                }
                else
                {
                        front=(front+1)%size;
                }
                printf("\nDeleted element :%d\n",element);
                return(element);
        }
```

```
}
void display()
{
	int i;
	if(isempty())
	printf("\nEmpty Queue\n");
	else
	{
		printf("\nFront :%d",front);
		printf("\nQ :");
		for(i=front;i!=rear;i=(i+1)%size)
		{
			printf("%d\t",Q[i]);
		}
		printf("%d",Q[i]);
		printf("\nRear :%d\n",rear);
	}
}
void search()
{
	int a,j;
	printf("\nEnter the element to search :");
	scanf("%d",&a);
	if(front==-1)
	{
		printf("\nQueue is empty!!");
```

```
            return ;
        }
        else
        {
            for(j=0;j<size;j++)
            {
                if(a==Q[j])
                {
                    printf("\nElement %d found in %d",a,j);
                    return ;
                }
            }
                if(a!=Q[j])
                {
                    printf("\nElement not found");
                    return ;
                }
        }
}
int main()
{
        int choice=1,element;
        while(choice<5&&choice!=0){
            printf("\nPress 1 to insert element");
            printf("\nPress 2 to delete element");
            printf("\nPress 3 to display");
```

```
            printf("\nPress 4 to search");
            printf("\nEnter your choice :");
            scanf("%d",&choice);
            switch(choice)
            {
                case 1:
                printf("\nEnter the element to insert :");
                scanf("%d",&element);
                enqueue(element);
                break;
                case 2:
                dequeue();
                break;
                case 3:
                display();
                break;
case 4:
                search();
                break;
                default:
                printf("Enter valid option");
            }
        }
        return 0;
}
```

Output :-

Press 1 to insert element

Press 2 to delete element

Press 3 to display

Press 4 to search

Enter your choice :1

Enter the element to insert :10

Inserted :10

Press 1 to insert element

Press 2 to delete element

Press 3 to display

Press 4 to search

Enter your choice :1

Enter the element to insert :20

Inserted :20

Press 1 to insert element

Press 2 to delete element

Press 3 to display

Press 4 to search

Enter your choice :3

Front :0

Q :10 20

Rear :1

Press 1 to insert element

Press 2 to delete element

Press 3 to display

Press 4 to search

Enter your choice :2

Deleted element :10

Press 1 to insert element

Press 2 to delete element

Press 3 to display

Press 4 to search

Enter your choice :3

Front :1

Q :20

Rear :1

Press 1 to insert element

Press 2 to delete element

Press 3 to display

Press 4 to search

Enter your choice :4

Enter the element to search :20

Element 20 found in 1

Press 1 to insert element

Press 2 to delete element

Press 3 to display

Press 4 to search

Enter your choice :5

Enter valid option

Result :-Program executed successfully and output verified.

Program No:7

SINGLY LINKED STACK

Aim :- To implement singly linked stack

Algorithm :-

1. Include all the header files which are used in the program. And declare all the user defined functions.

2. Define a 'Node' structure with two members data and next.

3. Define a Node pointer 'top' and set it to NULL.

4. Implement the main method by displaying Menu with list of operations and make suitable function calls in the main method.

5. Create a newNode with given value.

6. Check whether stack is Empty (top == NULL)

7. If it is Empty, then set newNode → next = NULL.

8. If it is Not Empty, then set newNode → next = top.

9. Finally, set top = newNode.

10. Check whether stack is Empty (top == NULL).

11. If it is Empty, then display "Stack is Empty!!! Deletion is not possible!!!" and terminate the function

12. If it is Not Empty, then define a Node pointer 'temp' and set it to 'top'.

13. Then set 'top = top → next'.

14. Finally, delete 'temp'. (free(temp)).

15. Check whether stack is Empty (top == NULL).

16. If it is Empty, then display 'Stack is Empty!!!' and terminate the function.

17. If it is Not Empty, then define a Node pointer 'temp' and initialize with top.

18. Display 'temp → data --->' and move it to the next node. Repeat the same until temp reaches to the first node in the stack. (temp → next != NULL).

19. Finally! Display 'temp → data ---> NULL'

Program ;-

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
        int data;
        struct node *link;
}*top,*top1,*temp;

void push(int element);
void pop();
void display();

void main()
{
        int num,choice;
        printf("\nStack using linked list\n");
        printf("1-PUSH\n");
        printf("2-POP\n");
        printf("3-DISPLAY\n");
        printf("4-EXIT\n");
        while(1)
        {
                printf("\n\nEnter the choice :");
                scanf("%d",&choice);
```

```
            switch(choice)
            {
                case 1:
                    printf("\nEnter the element :");
                    scanf("%d",&num);
                    push(num);
                    break;
                case 2:
                    pop();
                    break;
                case 3:
                    display();
                    break;
                case 4:
                    exit(0);
                default:
                    printf("\nEnter valid choice!!\n");
                    break;
            }
        }
}
void push(int element)
{
        if(top==NULL)
        {
            top=(struct node *)malloc(1*sizeof(struct node));
```

```
            top->link=NULL;
            top->data=element;
        }
        else
        {
            temp=(struct node *)malloc(1*sizeof(struct node));
            temp->link=top;
            temp->data=element;
            top=temp;
        }
}
void pop()
{
        top1=top;
        if(top1==NULL)
        {
            printf("\nError :empty\n");
            return;

        }
        else
        {
            top1=top1->link;
            printf("\nPopped element is %d\n",top->data);
            free(top);
            top=top1;
```

```
        }
}
void display()
{
        top1=top;
        if(top1==NULL)
        {
                printf("\nEmpty stack!!\n");
                return;
        }
        while(top1!=NULL)
        {
                printf("%d--->",top1->data);
                top1=top1->link;
        }
        printf("NULL\n");
}
```

Output :-

```
Stack using linked list
1-PUSH
2-POP
3-DISPLAY
4-EXIT
Enter the choice :1
Enter the element :12
```

Enter the choice :5

Enter valid choice!!

Enter the choice :1

Enter the element :23

Enter the choice :3

23--->12--->NULL

Enter the choice :2

Popped element is 23

Enter the choice :3

12--->NULL

Result :-Program executed successfully and output verified.

Program No:8

DISJOINT SET

Aim :- To find a graph contain cycle or not

Algorithm :-

1. Declare a structure to represent each element in the set, which should contain at least two fields: an integer value to represent the element's value, and an integer value to represent the element's parent.

2. Initialize each element's parent value to itself, indicating that each element is initially in a disjoint set of size 1.

3. Define two functions: find and union.

4. The find function takes an element as input and returns the root of the set that the element belongs to. To find the root of a set, recursively follow the parent pointers until the root is reached (i.e., an element whose parent is itself). To optimize performance, implement path compression by updating each element's parent to the root during the recursion, so that future calls to find can complete more quickly.

5. The union function takes two elements and combines the sets they belong to. To do this, first find the roots of the sets that the elements belong to using the find function. If the roots are the same, the elements are already in the same set and no further action is needed. Otherwise, set the parent of one root to be the other root, effectively merging the sets. To optimize performance, implement union by rank or union by size, which involve merging the smaller set into the larger one to minimize the height of the resulting tree.

6. Use the find and union functions to perform set operations such as checking if two elements are in the same set, or merging sets.

Program :-

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

```
#include<conio.h>
#include<stdlib.h>
struct node{
  struct node *rep;
  struct node *next;
  int data;
}*heads[50],*tails[50];
static int countRoot=0;
void makeSet(int x){
        struct node *new=(struct node *)malloc(sizeof(struct node));
        new->rep=new;
        new->next=NULL;
        new->data=x;
        heads[countRoot]=new;
        tails[countRoot++]=new;
}
struct node* find(int a){
        int i;
        struct node *tmp=(struct node *)malloc(sizeof(struct node));
        for(i=0;i<countRoot;i++){
                tmp=heads[i];
                while(tmp!=NULL){
                if(tmp->data==a)
                return tmp->rep;
                tmp=tmp->next;
                }
```

```
        }
        return NULL;
}
void unionSets(int a,int b){
        int i,pos,flag=0,j;
        struct node *tail2=(struct node *)malloc(sizeof(struct node));
        struct node *rep1=find(a);
        struct node *rep2=find(b);
        if(rep1==NULL||rep2==NULL){
                printf("\nElement not present in the DS\n");
                return;
        }
        if(rep1!=rep2){
                for(j=0;j<countRoot;j++){
                        if(heads[j]==rep2){
                                pos=j;
                                flag=1;
                                countRoot-=1;
                                tail2=tails[j];
                                for(i=pos;i<countRoot;i++){
                                        heads[i]=heads[i+1];
                                        tails[i]=tails[i+1];
                                }
                        }
                        if(flag==1)
                                break;
```

```
                }
                for(j=0;j<countRoot;j++){
                        if(heads[j]==rep1){
                                tails[j]->next=rep2;
                                tails[j]=tail2;
                                break;
                        }
                }
                while(rep2!=NULL){
                rep2->rep=rep1;
                rep2=rep2->next;
                }
        }
}
int search(int x){
        int i;
        struct node *tmp=(struct node *)malloc(sizeof(struct node));
        for(i=0;i<countRoot;i++){
                tmp=heads[i];
                if(heads[i]->data==x)
                        return 1;
                while(tmp!=NULL){
                        if(tmp->data==x)
                                return 1;
                        tmp=tmp->next;
                }
```

```
	}
	return 0;
}
void main(){
int choice,x,i,j,y,flag=0;
	do{
		printf("\n||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||\n");
		printf("\n.......MENU.......\n\n1.Make Set\n2.Display set representatives\n3.Union\n4.Find Set\n5.Exit\n");

		printf("Enter your choice :  ");
		scanf("%d",&choice);
		printf("\n||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||\n");
		switch(choice){
		case 1:
			printf("\nEnter new element : ");
			scanf("%d",&x);
			if(search(x)==1)
				printf("\nElement already present in the disjoint set DS\n");
			else
				makeSet(x);
			break;
		case 2:
			printf("\n");
			for(i=0;i<countRoot;i++)
				printf("%d ",heads[i]->data);
```

```
            printf("\n");
            break;
        case 3:
            printf("\nEnter first element : ");
            scanf("%d",&x);
            printf("\nEnter second element : ");
            scanf("%d",&y);
            unionSets(x,y);
            break;
        case 4:
            printf("\nEnter the element");
            scanf("%d",&x);
            struct node *rep=(struct node *)malloc(sizeof(struct node));
            rep=find(x);
            if(rep==NULL)
            printf("\nElement not present in the DS\n");
            else
            printf("\nThe representative of %d is %d\n",x,rep->data);
            break;
        case 5:
            exit(0);
        default:
            printf("\nWrong choice\n");
            break;
        }
}
```

```
        while(1);
};
```

Output :-

```
||||||||||||||||||||||||||||||||||||||||||||||||||||||||
.......MENU.......
1.Make Set
2.Display set representatives
3.Union
4.Find Set
5.Exit
Enter your choice :  1
||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Enter new element : 15
||||||||||||||||||||||||||||||||||||||||||||||||||||||||
.......MENU.......
1.Make Set
2.Display set representatives
3.Union
4.Find Set
5.Exit
Enter your choice :  1
||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Enter new element : 32
||||||||||||||||||||||||||||||||||||||||||||||||||||||||
.......MENU.......
1.Make Set
```

2.Display set representatives

3.Union

4.Find Set

5.Exit

Enter your choice : 2

||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||

15 32

||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||

.....MENU.....

1.Make Set

2.Display set representatives

3.Union

4.Find Set

5.Exit

Enter your choice : 3

||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||

Enter first element : 15

Enter second element : 32

||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||

.....MENU.....

……

Result :- Program executed successfully and output verified.

Program No:9

B-TREE

Aim :- To implement traversal in a bst

Algorithm :-

1. Define the structure for a B-tree node. A B-tree node contains a fixed number of keys and pointers to its children.

2. Implement a function to create a new B-tree node. This function should allocate memory for the new node and initialize its key and child pointer arrays.

3. Implement a function to insert a new key into the B-tree. This function should start at the root node and traverse down the tree until it finds the appropriate leaf node for the new key. If the leaf node is full, it should split the node into two and promote the median key to the parent node.

4. Implement a function to search for a key in the B-tree. This function should start at the root node and traverse down the tree, comparing each key in the node with the search key until it finds a match or reaches a leaf node.

5. Implement a function to delete a key from the B-tree. This function should start at the root node and traverse down the tree until it finds the appropriate leaf node for the key to be deleted. If the leaf node has enough keys to spare, it can simply delete the key. Otherwise, it may have to borrow a key from a neighboring node or merge with a neighboring node.

6. Implement a function to print the keys in the B-tree in sorted order. This function should start at the leftmost leaf node and traverse the tree in-order, printing each key it encounters.

7. Implement a function to free the memory used by the B-tree. This function should recursively free the memory for each node in the tree.

Program :-

```
#include<stdio.h>
        #include "stdlib.h"

        #define M 5

        typedef struct _node {
            int    n;
            int    keys[M - 1];
            struct _node *p[M];
        } node;
        node *root = NULL;

        typedef enum KeyStatus {
            Duplicate,
            SearchFailure,
            Success,
            InsertIt,
            LessKeys,
        } KeyStatus;

        void insert(int key);
        void display(node *root, int);
        void DelNode(int x);
        void search(int x);
        KeyStatus ins(node *r, int x, int* y, node** u);
```

```
int searchPos(int x, int *key_arr, int n);
KeyStatus del(node *r, int x);
void eatline(void);
void inorder(node *ptr);
int totalKeys(node *ptr);
void printTotal(node *ptr);
int getMin(node *ptr);
int getMax(node *ptr);
void getMinMax(node *ptr);
int max(int first, int second, int third);
int maxLevel(node *ptr);
void printMaxLevel(node *ptr);


int main() {
        int key;
        int choice;
        printf("Creation of B tree for M=%d\n", M);

                printf("1.Insert\n");
                printf("2.Delete\n");
                printf("3.Search\n");
                printf("4.Display\n");
                printf("5.Quit\n");
                printf("6.Enumerate\n");
                printf("7.Total Keys\n");
```

```
        printf("8.Min and Max Keys\n");
        printf("9.Max Level\n");
    while (1) {
        printf("Enter your choice : ");
        scanf("%d", &choice); eatline();

        switch (choice) {
        case 1:
            printf("Enter the key : ");
            scanf("%d", &key); eatline();
            insert(key);
            break;
        case 2:
            printf("Enter the key : ");
            scanf("%d", &key); eatline();
            DelNode(key);
            break;
        case 3:
            printf("Enter the key : ");
            scanf("%d", &key); eatline();
            search(key);
            break;
        case 4:
            printf("Btree is :\n");
            display(root, 0);
            break;
```

```
            case 5:
                exit(1);
            case 6:
                printf("Btree in sorted order is:\n");
                inorder(root); putchar('\n');
                break;
            case 7:
                printf("The total number of keys in this tree is:\n");
                printTotal(root);
                break;
            case 8:
                getMinMax(root);
                break;
            case 9:
                printf("The maximum level in this tree is:\n");
                printMaxLevel(root);
                break;
            default:
                printf("Wrong choice\n");
                break;
            }/*End of switch*/
        }/*End of while*/
        return 0;
}/*End of main()*/

void insert(int key) {
```

```
        node *newnode;
        int upKey;
        KeyStatus value;
        value = ins(root, key, &upKey, &newnode);
        if (value == Duplicate)
                printf("Key already available\n");
        if (value == InsertIt) {
                node *uproot = root;
                root = (node*)malloc(sizeof(node));
                root->n = 1;
                root->keys[0] = upKey;
                root->p[0] = uproot;
                root->p[1] = newnode;
        }/*End of if */
}/*End of insert()*/

KeyStatus ins(node *ptr, int key, int *upKey, node **newnode) {
        node *newPtr, *lastPtr;
        int pos, i, n, splitPos;
        int newKey, lastKey;
        KeyStatus value;
        if (ptr == NULL) {
                *newnode = NULL;
                *upKey = key;
                return InsertIt;
        }
```

```
n = ptr->n;
pos = searchPos(key, ptr->keys, n);
if (pos < n && key == ptr->keys[pos])
        return Duplicate;
value = ins(ptr->p[pos], key, &newKey, &newPtr);
if (value != InsertIt)
        return value;

if (n < M - 1) {
        pos = searchPos(newKey, ptr->keys, n);

        for (i = n; i>pos; i--) {
                ptr->keys[i] = ptr->keys[i - 1];
                ptr->p[i + 1] = ptr->p[i];
        }
        /*Key is inserted at exact location*/
        ptr->keys[pos] = newKey;
        ptr->p[pos + 1] = newPtr;
        ++ptr->n;
        return Success;
}/*End of if */

if (pos == M - 1) {
        lastKey = newKey;
        lastPtr = newPtr;
}
```

```
        else {
                lastKey = ptr->keys[M - 2];
                lastPtr = ptr->p[M - 1];
                for (i = M - 2; i>pos; i--) {
                        ptr->keys[i] = ptr->keys[i - 1];
                        ptr->p[i + 1] = ptr->p[i];
                }
                ptr->keys[pos] = newKey;
                ptr->p[pos + 1] = newPtr;
        }
        splitPos = (M - 1) / 2;
        (*upKey) = ptr->keys[splitPos];

        (*newnode) = (node*)malloc(sizeof(node));/*Right node after split*/
        ptr->n = splitPos; /*No. of keys for left splitted node*/
        (*newnode)->n = M - 1 - splitPos;/*No. of keys for right splitted node*/
        for (i = 0; i < (*newnode)->n; i++) {
                (*newnode)->p[i] = ptr->p[i + splitPos + 1];
                if (i < (*newnode)->n - 1)
                        (*newnode)->keys[i] = ptr->keys[i + splitPos + 1];
                else
                        (*newnode)->keys[i] = lastKey;
        }
        (*newnode)->p[(*newnode)->n] = lastPtr;
        return InsertIt;
}/*End of ins()*/
```

```
void display(node *ptr, int blanks) {
    if (ptr) {
        int i;
        for (i = 1; i <= blanks; i++)
            printf(" ");
        for (i = 0; i < ptr->n; i++)
            printf("%d ", ptr->keys[i]);
        printf("\n");
        for (i = 0; i <= ptr->n; i++)
            display(ptr->p[i], blanks + 10);
    }/*End of if*/
}/*End of display()*/

void search(int key) {
    int pos, i, n;
    node *ptr = root;
    printf("Search path:\n");
    while (ptr) {
        n = ptr->n;
        for (i = 0; i < ptr->n; i++)
            printf(" %d", ptr->keys[i]);
        printf("\n");
        pos = searchPos(key, ptr->keys, n);
        if (pos < n && key == ptr->keys[pos]) {
            printf("Key %d found in position %d of last dispalyed node\n",
key, i);
```

```
                return;
            }
            ptr = ptr->p[pos];
        }
        printf("Key %d is not available\n", key);
}/*End of search()*/

int searchPos(int key, int *key_arr, int n) {
        int pos = 0;
        while (pos < n && key > key_arr[pos])
            pos++;
        return pos;
}/*End of searchPos()*/

void DelNode(int key) {
        node *uproot;
        KeyStatus value;
        value = del(root, key);
        switch (value) {
        case SearchFailure:
            printf("Key %d is not available\n", key);
            break;
        case LessKeys:
            uproot = root;
            root = root->p[0];
            free(uproot);
```

```
            break;
        default:
            return;
        }/*End of switch*/
}/*End of delnode()*/

KeyStatus del(node *ptr, int key) {
        int pos, i, pivot, n, min;
        int *key_arr;
        KeyStatus value;
        node **p, *lptr, *rptr;

        if (ptr == NULL)
            return SearchFailure;
        /*Assigns values of node*/
        n = ptr->n;
        key_arr = ptr->keys;
        p = ptr->p;
        min = (M - 1) / 2;


        pos = searchPos(key, key_arr, n);
        // p is a leaf
        if (p[0] == NULL) {
            if (pos == n || key < key_arr[pos])
                return SearchFailure;
```

```
        /*Shift keys and pointers left*/
        for (i = pos + 1; i < n; i++)
        {
                key_arr[i - 1] = key_arr[i];
                p[i] = p[i + 1];
        }
        return --ptr->n >= (ptr == root ? 1 : min) ? Success : LessKeys;
}/*End of if */

 //if found key but p is not a leaf
if (pos < n && key == key_arr[pos]) {
        node *qp = p[pos], *qp1;
        int nkey;
        while (1) {
                nkey = qp->n;
                qp1 = qp->p[nkey];
                if (qp1 == NULL)
                        break;
                qp = qp1;
        }/*End of while*/
        key_arr[pos] = qp->keys[nkey - 1];
        qp->keys[nkey - 1] = key;
}/*End of if */
value = del(p[pos], key);
if (value != LessKeys)
        return value;
```

```
if (pos > 0 && p[pos - 1]->n > min) {
        pivot = pos - 1;
        lptr = p[pivot];
        rptr = p[pos];

        rptr->p[rptr->n + 1] = rptr->p[rptr->n];
        for (i = rptr->n; i>0; i--) {
                rptr->keys[i] = rptr->keys[i - 1];
                rptr->p[i] = rptr->p[i - 1];
        }
        rptr->n++;
        rptr->keys[0] = key_arr[pivot];
        rptr->p[0] = lptr->p[lptr->n];
        key_arr[pivot] = lptr->keys[--lptr->n];
        return Success;
}/*End of if */
 //if (posn > min)
if (pos < n && p[pos + 1]->n > min) {
        pivot = pos;
        lptr = p[pivot];
        rptr = p[pivot + 1];
        lptr->keys[lptr->n] = key_arr[pivot];
        lptr->p[lptr->n + 1] = rptr->p[0];
        key_arr[pivot] = rptr->keys[0];
        lptr->n++;
```

```
        rptr->n--;
        for (i = 0; i < rptr->n; i++) {
                rptr->keys[i] = rptr->keys[i + 1];
                rptr->p[i] = rptr->p[i + 1];
        }/*End of for*/
        rptr->p[rptr->n] = rptr->p[rptr->n + 1];
        return Success;
}/*End of if */

if (pos == n)
        pivot = pos - 1;
else
        pivot = pos;

lptr = p[pivot];
rptr = p[pivot + 1];
/*merge right node with left node*/
lptr->keys[lptr->n] = key_arr[pivot];
lptr->p[lptr->n + 1] = rptr->p[0];
for (i = 0; i < rptr->n; i++) {
        lptr->keys[lptr->n + 1 + i] = rptr->keys[i];
        lptr->p[lptr->n + 2 + i] = rptr->p[i + 1];
}
lptr->n = lptr->n + rptr->n + 1;
free(rptr); /*Remove right node*/
for (i = pos + 1; i < n; i++) {
```

```
            key_arr[i - 1] = key_arr[i];
            p[i] = p[i + 1];
        }
        return --ptr->n >= (ptr == root ? 1 : min) ? Success : LessKeys;
}

void eatline(void) {
        char c;
        while ((c = getchar()) != '\n');
}


void inorder(node *ptr) {
        if (ptr) {
                if (ptr->n >= 1) {
                        inorder(ptr->p[0]);
                        printf("%d ", ptr->keys[0]);
                        inorder(ptr->p[1]);
                        if (ptr->n == 2) {
                                printf("%d ", ptr->keys[1]);
                                inorder(ptr->p[2]);
                        }
                }
        }
}
```

```
int totalKeys(node *ptr) {
	if (ptr) {
		int count = 1;
		if (ptr->n >= 1) {
			count += totalKeys(ptr->p[0]);
			count += totalKeys(ptr->p[1]);
			if (ptr->n == 2) count += totalKeys(ptr->p[2]) + 1;
		}
		return count;
	}
	return 0;
}


void printTotal(node *ptr) {
	printf("%d\n", totalKeys(ptr));
}

int getMin(node *ptr) {
	if (ptr) {
		int min;
		if (ptr->p[0] != NULL) min = getMin(ptr->p[0]);
		else min = ptr->keys[0];
		return min;
	}
```

```
        return 0;
}


int getMax(node *ptr) {
        if (ptr) {
                int max;
                if (ptr->n == 1) {
                        if (ptr->p[1] != NULL) max = getMax(ptr->p[1]);
                        else max = ptr->keys[0];
                }
                if (ptr->n == 2) {
                        if (ptr->p[2] != NULL) max = getMax(ptr->p[2]);
                        else max = ptr->keys[1];
                }
                return max;
        }
        return 0;
}
void getMinMax(node *ptr) {
        printf("%d %d\n", getMin(ptr), getMax(ptr));
}

int max(int first, int second, int third) {
        int max = first;
        if (second > max) max = second;
```

```
        if (third > max) max = third;
        return max;
    }

    int maxLevel(node *ptr) {
        if (ptr) {
            int l = 0, mr = 0, r = 0, max_depth;
            if (ptr->p[0] != NULL) l = maxLevel(ptr->p[0]);
            if (ptr->p[1] != NULL) mr = maxLevel(ptr->p[1]);
            if (ptr->n == 2) {
                if (ptr->p[2] != NULL) r = maxLevel(ptr->p[2]);
            }
            max_depth = max(l, mr, r) + 1;
            return max_depth;
        }
        return 0;
    }
    void printMaxLevel(node *ptr) {
        int max = maxLevel(ptr) - 1;
        if (max == -1) printf("tree is empty\n");
        else printf("%d\n", max);
    }
```

Output :-

Creation of B tree for M=5

1.Insert

2.Delete

3.Search

4.Display

5.Quit

6.Enumerate

7.Total Keys

8.Min and Max Keys

9.Max Level

Enter your choice : 1

Enter the key : 10

Enter your choice : 1

Enter the key : 3

Enter your choice : 1

Enter the key : 6

Enter your choice : 1

Enter the key : 13

Enter your choice : 1

Enter the key : 18

Enter your choice : 1

Enter the key : 1

Enter your choice : 1

Enter the key : 2

Enter your choice : 1

Enter the key : 4

Enter your choice : 1

Enter the key : 5

Enter your choice : 1

Enter the key : 7

Enter your choice : 1

Enter the key : 8

Enter your choice : 1

Enter the key : 9

Enter your choice : 1

Enter the key : 11

Enter your choice : 1

Enter the key : 12

Enter your choice : 1

Enter the key : 14

Enter your choice : 1

Enter the key : 16

Enter your choice : 1

Enter the key : 19

Enter your choice : 1

Enter the key : 20

Enter your choice : 1

Enter the key : 21

Enter your choice : 1

Enter the key : 24

Enter your choice : 4

Btree is :

10

3 6

1 2

4 5

7 8 9

13 18

11 12

14 16

19 20 21 24

Enter your choice : 2

Enter the key : 8

Enter your choice : 2

Enter the key : 18

Enter your choice : 2

Enter the key : 16

Enter your choice : 2

Enter the key : 4

Enter your choice : 4

Btree is :

3 10 13 20

1 2

5 6 7 9

11 12

14 19

21 24

Result :- Program executed successfully and output verified.

Program No:10

BFS GRAPH TRAVERSAL

Aim :- To create a program to implement BFS traversal

Algorithm :-

1. Create a queue data structure to store the nodes that will be visited.
2. Create a boolean array to mark the visited nodes.
3. Start with a source node and mark it as visited.
4. Enqueue the source node into the queue.
5. While the queue is not empty, do the following:
    a. Dequeue the front node from the queue.
    b. For each of the adjacent nodes of the dequeued node, do the following:
    i. If the adjacent node has not been visited, mark it as visited and enqueue it into the queue.
6. Repeat step 5 until the queue is empty.

Program :-

```
#include<stdio.h>
#include<stdlib.h>
#define MAX 100
#define initial 1
#define waiting 2
```

```
#define visited 3

int n;
int adj[MAX][MAX];
int state[MAX];
void create_graph();
void BF_Traversal();
void BFS(int v);
int queue[MAX], front = -1,rear = -1;
void insert_queue(int vertex);
int delete_queue();
int isEmpty_queue();

int main()
{
        create_graph();
        BF_Traversal();
        return 0;
}

void BF_Traversal()
{
        int v;
        for(v=0; v<n; v++)
        state[v] = initial;
        printf("Enter Start Vertex for BFS: \n");
```

```
        scanf("%d", &v);
        BFS(v);
}

void BFS(int v)
{
        int i;
        printf("BFS traversal");
        insert_queue(v);
        state[v] = waiting;
        while(!isEmpty_queue())
        {
                v = delete_queue();
                printf("%d ",v);
                state[v] = visited;
                for(i=0;i<n;i++)
                {
                        if(adj[v][i] == 1 && state[i] == initial)
                        {
                                insert_queue(i);
                                state[i] = waiting;
                        }
                }
        }
        printf("\n");
}
```

```
void insert_queue(int vertex)
{
        if(rear == MAX-1)
        printf("Queue Overflow\n");
        else
        {
                if(front == -1)
                front = 0;
                rear = rear+1;
                queue[rear] = vertex ;
        }
}


int isEmpty_queue()
{
        if(front == -1 || front > rear)
        return 1;
        else
        return 0;
}


int delete_queue()
{
```

```
        int delete_item;
        if(front == -1 || front > rear)
        {
                printf("Queue Underflow\n");
                exit(1);
        }
        delete_item = queue[front];
        front = front+1;
        return delete_item;
}
void create_graph()
{
        int count,max_edge,origin,destin;
        printf("Enter number of vertices : ");
        scanf("%d",&n);
        max_edge = n*(n-1);
        for(count=1;count<=max_edge;count++)
        {
                printf("Enter edge %d( -1 -1 to quit ) : ",count);
                scanf("%d %d",&origin,&destin);
                if((origin == -1) && (destin == -1))
                break;
                if(origin>=n || destin>=n || origin<0 || destin<0)
                {
                        printf("Invalid edge!\n");
                        count--;
```

```
                }
                else
                {
                        adj[origin][destin] = 1;
                }
        }
}
```

Output :-

```
Enter number of vertices : 5
Enter edge 1( -1 -1 to quit ) : 0 2
Enter edge 2( -1 -1 to quit ) : 2 0
Enter edge 3( -1 -1 to quit ) : 0 3
Enter edge 4( -1 -1 to quit ) : 3 0
Enter edge 5( -1 -1 to quit ) : 0 1
Enter edge 6( -1 -1 to quit ) : 1 0
Enter edge 7( -1 -1 to quit ) : 2 4
Enter edge 8( -1 -1 to quit ) : 4 2
Enter edge 9( -1 -1 to quit ) : 1 2
Enter edge 10( -1 -1 to quit ) : 2 1
Enter edge 11( -1 -1 to quit ) : -1 -1
Enter Start Vertex for BFS:
0
BFS traversal
0 1 2 3 4
```

Result :- Program executed successfully and output verified.

Program No:11

DFS GRAPH TRAVERSAL

Aim :- To create a program to implement DFS traversal

Algorithm :-

1. Create a stack to store the vertices to be visited.
2. Initialize all vertices as not visited.
3. Pick a starting vertex and mark it as visited and push it to the stack.
4. While the stack is not empty, do the following:
    a. Pop a vertex from the stack and print it.
    b. Get all adjacent vertices of the popped vertex.
    c. For each adjacent vertex, if it has not been visited, mark it as visited and push it to the stack.
5. Repeat step 4 until the stack is empty.

Program :-

```
#include<stdio.h>
void DFS(int);
int G[10][10],visited[10],n;
void main()
{
int i,j;
printf("Enter number of vertices:");
scanf("%d",&n);
//read the adjecency matrix
printf("\nEnteradjecency matrix of the graph:");
```

```
for(i=0;i<n;i++)
for(j=0;j<n;j++)
scanf("%d",&G[i][j]);
//visited is initialized to zero
printf("DFS traversal");
for(i=0;i<n;i++)
visited[i]=0;
DFS(0);
}
void DFS(int i)
{
int j;
printf("\n%d",i);
visited[i]=1;
for(j=0;j<n;j++)
if(!visited[j]&&G[i][j]==1)
DFS(j);
}
```

Output :-

Enter number of vertices:5

Enter adjecency matrix of the graph:

0 1 1 1 0

1 0 1 0 0

1 1 0 0 1

1 0 0 0 0

0 0 1 0 0

DFS traversal

0

1

2

4

3

Result :- Program executed successfully and output verified.

Program No:12

KRUSKALS ALGORITHM

Aim :- To find minimum cost spanning tree using Kruskal's algorithm

Algorithm :-

1. Sort the edges of the graph by weight in non-decreasing order.
2. Create a forest of n disjoint trees, where n is the number of vertices in the graph.
3. Iterate through the edges in order of increasing weight, and for each edge:
    a. If the endpoints of the edge belong to different trees in the forest, add the edge to the MST and merge the two trees.
    b. If the endpoints of the edge belong to the same tree, discard the edge to avoid creating a cycle.
4. Continue until all vertices are in the same tree, or until n-1 edges have been added to the MST.

Program :-

```
#include<stdio.h>

#include<stdlib.h>

int i,j,k,a,b,u,v,n,ne=1;

int min,mincost=0,cost[9][9],parent[9];

int find(int);

int uni(int,int);

void main()

{

        printf("\nImplementation of Kruskal's algorithm\n");

        printf("\nEnter the no. of vertices:");

        scanf("%d",&n);

        printf("\nEnter the cost adjacency matrix:\n");

        for(i=1;i<=n;i++)

        {
```

```
        for(j=1;j<=n;j++)
        {
                scanf("%d",&cost[i][j]);
                if(cost[i][j]==0)
                        cost[i][j]=999;
        }
}
printf("The edges of Minimum Cost Spanning Tree are\n");
while(ne < n)
{
        for(i=1,min=999;i<=n;i++)
        {
                for(j=1;j <= n;j++)
                {
                        if(cost[i][j] < min)
                        {
                                min=cost[i][j];
                                a=u=i;
                                b=v=j;
                        }
                }
        }
        u=find(u);
        v=find(v);
        if(uni(u,v))
        {
```

```
                printf("%d edge (%d,%d) =%d\n",ne++,a,b,min);
                mincost +=min;
            }
            cost[a][b]=cost[b][a]=999;
        }
        printf("\nMinimum cost = %d\n",mincost);
}
int find(int i)
{
        while(parent[i])
        i=parent[i];
        return i;
}
int uni(int i,int j)
{
        if(i!=j)
        {
            parent[j]=i;
            return 1;
        }
        return 0;
}
```

Output :-

Implementation of Kruskal's algorithm

Enter the no. of vertices:5

Enter the cost adjacency matrix:

0 2 1 0 0

2 0 0 4 5

1 0 0 0 2

0 4 0 0 0

0 0 2 0 0

The edges of Minimum Cost Spanning Tree are

1 edge (1,3) =1

2 edge (1,2) =2

3 edge (3,5) =2

4 edge (2,4) =4

Minimum cost = 9

Result :- Program executed successfully and output verified.

Program No:13

PRIMS ALGORITHM

Aim :- To find minimum cost spanning tree using Prim's algorithm

Algorithm :-

1. Start with a random vertex, say vertex V, from the given graph.
2. Initialize an empty set called the MST (minimum spanning tree) that will eventually contain the edges of the MST.
3. Initialize a priority queue called PQ that will contain pairs of vertices and their respective distances. The distance between vertices is the weight of the edge connecting them.
4. For every vertex adjacent to V, add the pair (V, adjacent vertex) to the PQ with its respective distance.
5. While the PQ is not empty, do the following:
    a. Remove the vertex-pair with the smallest distance from the PQ.
    b. If both vertices of the removed pair are already in the MST, continue to the next iteration.
    c. Otherwise, add the edge connecting the two vertices to the MST and add all the pairs of vertices adjacent to the newly added vertex to the PQ with their respective distances.
6. Continue iterating until all vertices are in the MST.
7. Return the MST.

Program :-

```
#include<stdio.h>

int a,b,u,v,n,i,j,ne=1;

int visited[10]={0},min,mincost=0,cost[10][10];

void main()

{

    printf("\nEnter the number of nodes:");

    scanf("%d",&n);

    printf("\nEnter the adjacency matrix:\n");
```

```
        for(i=1;i<=n;i++)
        for(j=1;j<=n;j++)
        {
                scanf("%d",&cost[i][j]);
                if(cost[i][j]==0)
                cost[i][j]=999;
        }
        visited[1]=1;
        printf("\n");
        while(ne < n)
        {
                for(i=1,min=999;i<=n;i++)
                for(j=1;j<=n;j++)
                if(cost[i][j]< min)
                if(visited[i]!=0)
                {
                        min=cost[i][j];
                        a=u=i;

b=v=j;
                }
                if(visited[u]==0 || visited[v]==0)
                {
                        printf("\n Edge %d:(%d %d) cost:%d",ne++,a,b,min);
                        mincost+=min;
                        visited[b]=1;
```

```
            }
            cost[a][b]=cost[b][a]=999;
        }
        printf("\n Minimun cost=%d",mincost);
}
```

Output :-

Enter the number of nodes:5

Enter the adjacency matrix:

0	2	1	0	0
2	0	0	4	5
1	0	0	0	2
0	4	0	0	0
0	0	2	0	0

Edge 1:(1 3) cost:1

Edge 2:(1 2) cost:2

Edge 3:(3 5) cost:2

Edge 4:(2 4) cost:4

Minimun cost=9

Result :- Program executed successfully and output verified.