

20MCA135 – DATA STRUCTURES LAB

Lab Report Submitted By

JENNY JOHNSON

Reg. No.: AJC22MCA-2053

In Partial Fulfilment for the Award of the Degree of

MASTER OF COMPUTER APPLICATIONS (2 Year) (MCA)

APJ ABDUL KALAM TECHNOLOGICAL UNIVERSITY



AMAL JYOTHI COLLEGE OF ENGINEERING

KANJIRAPPALLY

[Affiliated to APJ Abdul Kalam Technological University, Kerala. Approved by AICTE,
Accredited by NAAC with 'A' grade. Koovapally, Kanjirappally, Kottayam, Kerala – 686518]

2022-2023

DEPARTMENT OF COMPUTER APPLICATIONS

AMAL JYOTHI COLLEGE OF ENGINEERING

KANJIRAPPALLY



CERTIFICATE

This is to certify that the lab report, “**20MCA135 DATA STRUCTURES LAB**” is the bona fide work of **JENNY JOHNSON(AJC22MCA-2053)** in partial fulfilment of the requirements for the award of the Degree of Master of Computer Applications under APJ Abdul Kalam Technological University during the year **2022-23**.

Mrs. Meera Rose Mathew

Lab In- Charge

Rev. Fr. Dr. Rubin Thottupurathu Jose

Head of the Department

Internal Examiner

External Examiner

Course Code	Course Name	Syllabus Year	L-T-P-C
20MCA135	Data Structures Lab	2020	0-1-3-2

VISION

To promote an academic and research environment conducive for innovation centric technical education.

MISSION

MS1 -Provide foundations and advanced technical education in both theoretical and applied Computer Applications in-line with Industry demands.

MS2 -Create highly skilled computer professionals capable of designing and innovating real life solutions.

MS3 -Sustain an academic environment conducive to research and teaching focused to generate up-skilled professionals with ethical values.

MS4 -Promote entrepreneurial initiatives and innovations capable of bridging and contributing with sustainable, socially relevant technology solutions.

COURSE OUTCOME

CO	Outcome	Target
CO1	Use Basic Data Structures and its operations implementations.	61
CO2	Implement the Set and Disjoint Set Data Structures.	61
CO3	Understand the practical aspects of Advanced Tree Structures.	61
CO4	Realise Modern Heap Structures for effectively solving advanced Computational problems.	61
CO5	Implement Advanced Graph algorithms suitable for solving advanced computational problems.	61

COURSE END SURVEY

CO	Survey Question	Answer Format
CO1	To what extent you were able to use Basic Data Structures and its operations implementation	Excellent/Very Good/Good Satisfactory/Needs improvement
CO2	To what extent you were able to implement the Set and Disjoint Set Data Structures.	Excellent/Very Good/Good Satisfactory/Needs improvement
CO3	To what extent you were able to understand the practical aspects of Advanced Tree Structures.	Excellent/Very Good/Good Satisfactory/Needs improvement
CO4	To what extent you were able to understand Modern Heap Structures for effectively solving advanced Computational problems.	Excellent/Very Good/Good Satisfactory/Needs improvement
CO5	To what extent you were able to implement advanced graph algorithms suitable for solving advanced computational problems.	Excellent/Very Good/Good Satisfactory/Needs improvement

CONTENT

Sl. No.	Experiment	Date	CO	Page No.
1	Familiarization with gdb Advanced use of gcc:Important options-o,-c,-D, -l,-I,-g,-O,-save-temps,-pg Important commands-break,run,next,print,display,help Using gproof:Compile,Execute and Profile	29-10-2022	C01	1
2	Merge two sorted arrays and store in a third array.	01-11-2022	C01	16
3	Implementation of Singly Linked Stack.	08-11-2022	C01	21
4	Implementation of Circular Queue.	15-11-2022	C01	26
5	Implementation of Doubly Linked list.	22-11-2022	C01	32
6	Implementation of Binary search tree.	29-11-2022	C03	46
7	Implementation of Set data structure and Set operations.	06-12-2022	C02	56
8	Implementation of Binomial Heap.	13-12-2022	C04	60
9	Implementation of Depth First Search.	20-12-2022	C05	67
10	Implementation of Breadth First Search.	20-12-2022	C05	73
11	Implementation of Prim's Algorithm.	03-01-2023	C05	79
12	Implementation of Kruskal's Algorithm.	10-01-2023	C05	82

Experiment No.: 1

Aim

Familiarization with gdb Advanced use of gcc:Important options-o,-c,-D, -l,-l,-g,-O,-save-temps,-pg Important commands-break,run,next,print,display,help Using gprof:Compile,Execute and Profile.

CO1

Use Basic Data Structures and its operations implementations.

Procedure

Advanced use of GCC

The GNU Compiler Collection (GCC) is a collection of compilers and libraries for C, C++, Objective-C, Fortran, Ada, Go , and D programming languages. Many open-source projects, including the GNU tools and the Linux kernel, are compiled with GCC.

Installing GCC on Ubuntu

The default Ubuntu repositories contain a meta-package named build-essential that contains the GCC compiler and a lot of libraries and other utilities required for compiling software.

Perform the steps below to install the GCC Compiler Ubuntu 18.04:

Start by updating the packages list:

```
sudo apt-get update
```

Install the build-essential package by typing:

```
sudo apt-get install build-essential
```

The command installs a bunch of new packages including gcc, g++ and make.

You may also want to install the manual pages about using GNU/Linux for development:

```
sudo apt-get install manpages-dev
```

To validate that the GCC compiler is successfully installed, use the gcc -version command which prints the GCC version:

```
gcc -version Or gcc -v
```

The default version of GCC available in the Ubuntu 18.04 repositories is 7.4.0:

Compile and run a c++ program

Now go to that folder where you will create C/C++ programs. I am creating my programs in Desktop directory. Type these commands:

```
$ cd Desktop
```

```
$ sudo mkdir tst
```

```
$ cd tst
```

Open a file using any editor . Add this code in the file:

```
#include <iostream>
using namespace std;
int main() {
    cout << "Hello World!";
    return 0;
}
```

Save the file and exit.

Compile the program using any of the following command:

```
$ sudo g++ p1.cpp (p1 is the filename)
```

(or)

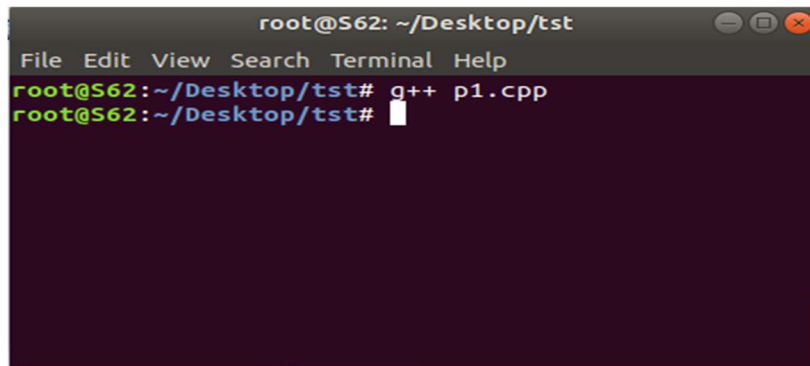
```
$ sudo g++ -o p1 p1.cpp
```

1. \$ sudo g++ p1.cpp

To compile your c++ code, use:

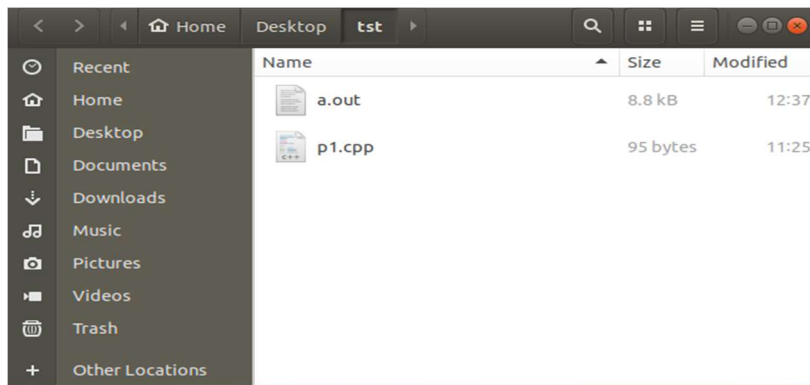
```
g++ p1.cpp
```

p1.cpp in the example is the name of the program to be compiled.



This will produce an executable in the same directory called a.out which you can run by typing this in your terminal:

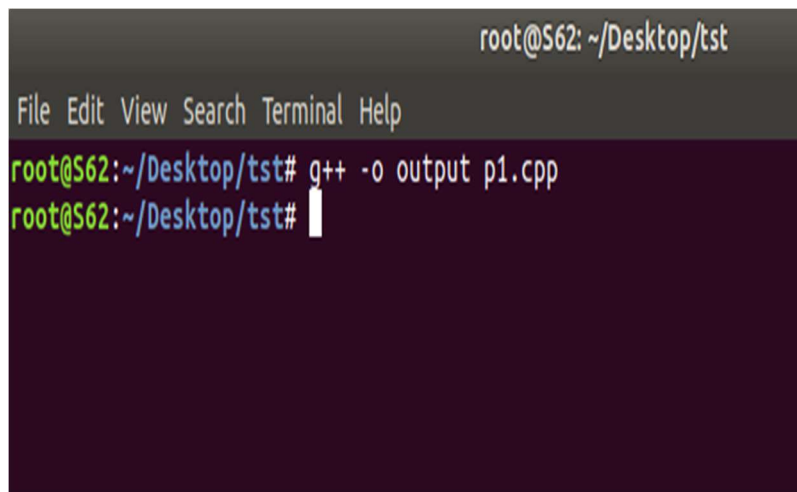
```
./a.out
```



\$ sudo g++ -o output p1.cpp

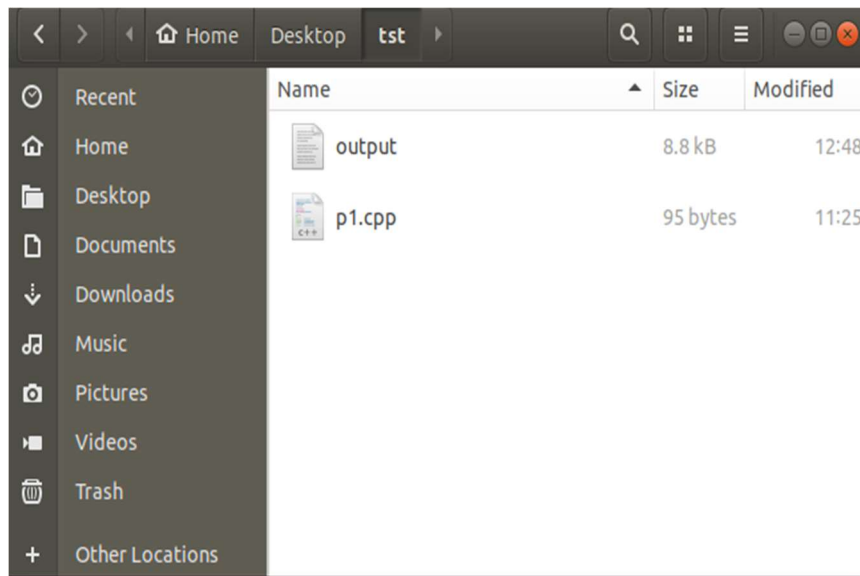
To specify the name of the compiled output file, so that it is not named a.out, use -o with your g++ command.

g++ -o output p1.cpp



This will compile p1.cpp to the binary file named output, and you can type ./output to run the compiled code.

./output.out

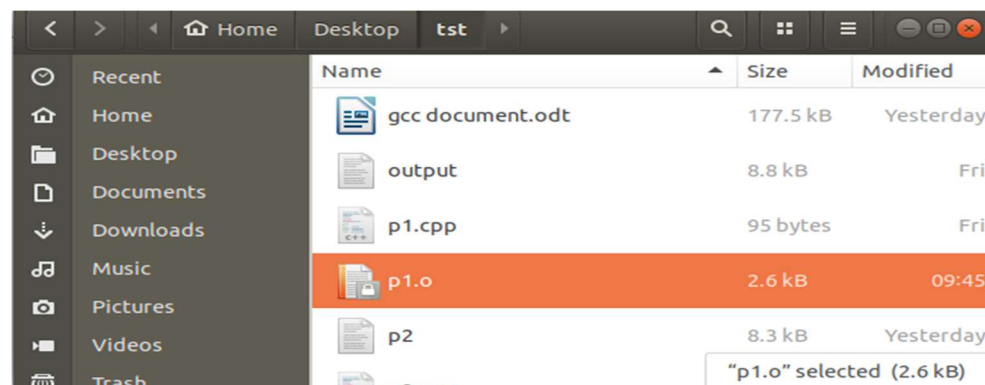


GCC: Important Options

→ **-c**

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

gcc -C p2.cpp



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

→ **-D**

The compiler option D can be used to define compile time macros in code.

Here is an example :

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

```
int main(void)
```

```
{
```

```
#ifdef MY_MACRO
```

```
    printf("\n Macro defined \n");
```



```

#endif
char c = -10;
// Print the string
printf("\n The Geek Stuff [%d]\n", c);
return 0;
}

```

The compiler option `-D` can be used to define the macro `MY_MACRO` from command line.

```
$ gcc -Wall -DMY_MACRO main.c -o main
```

```
$ ./main
```

Macro defined

The Geek Stuff [-10]

The print related to macro in the output confirms that the macro was defined.

```

root@S62: ~/Desktop/tst
File Edit View Search Terminal Help
root@S62:~/Desktop/tst# gcc p2.cpp
root@S62:~/Desktop/tst# ./a.out

The Geek Stuff [-10]
root@S62:~/Desktop/tst# gcc -Wall -DMY_MACRO p2.cpp -o p2
root@S62:~/Desktop/tst# ./p2

Macro defined

The Geek Stuff [-10]
root@S62:~/Desktop/tst#

```

→ `-l`

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` to produce the final executable 'main'.

→ `-g`

A program which goes into an infinite loop or "hangs" can be difficult to debug. On most systems a foreground process can be stopped by hitting Control-C, which sends it an interrupt signal (SIGINT). However, this does not help in debugging the problem--the SIGINT signal

terminates the process without producing a core dump. A more sophisticated approach is to *attach* to the running process with a debugger and inspect it interactively.

For example, here is a simple program with an infinite loop:

```
int
main (void)
{
    unsigned int i = 0;
    while (1) { i++; };
    return 0;
}
```

In order to attach to the program and debug it, the code should be compiled with the debugging option -g:

```
$ gcc -Wall -g loop.c
```

```
$ ./a.out
```

(program hangs)

Once the executable is running we need to find its process id (PID). This can be done from another session with the command ps x:

```
$ ps x
```

```
PID TTY STAT TIME COMMAND
```

```
... .. . . .
```

```
891 pts/1 R 0:11 ./a.out
```

```
s
```

→ -save-temps

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.

For example :

```
$ gcc -save-temps p2.cpp
```

```
$ ls
```

```
a.out p2.c p2.i p2.o p2.s
```

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

→ -pg

Generate extra code to write profile information suitable for the analysis program gprof. You must use this option when compiling the source files you want data about, and you must also use it when linking.

GDB Tutorial

Gdb is a debugger for C (and C++). It allows you to do things like run the program up to a certain point then stop and print out the values of certain variables at that point, or step through the program one line at a time and print out the values of each variable after executing each line. It uses a command line interface.

This is a brief description of some of the most commonly used features of gdb.

Compiling

To prepare your program for debugging with gdb, you must compile it with the -g flag. So, if your program is in a source file called memsim.c and you want to put the executable in the file memsim, then you would compile with the following command:

```
gcc -g -o memsim memsim.c
```

Invoking and Quitting GDB

To start gdb, just type gdb at the unix prompt. Gdb will give you a prompt that looks like this: (gdb). From that prompt you can run your program, look at variables, etc., using the commands listed below (and others not listed). Or, you can start gdb and give it the name of the program executable you want to debug by saying

```
gdb executable
```

To exit the program just type quit at the (gdb) prompt (actually just typing q is good enough).

Commands

help

Gdb provides online documentation. Just typing help will give you a list of topics. Then you can type help *topic* to get information about that topic (or it will give you more specific terms that you can ask for help about). Or you can just type help *command* and get information about any other command.

file

file *executable* specifies which program you want to debug.

run

run will start the program running under gdb. (The program that starts will be the one that you have previously selected with the file command, or on the unix command line when you started gdb. You can give command line arguments to your program on the gdb command line the same way you would on the unix command line, except that you are saying run instead of the program name:

```
run 2048 24 4
```

You can even do input/output redirection: run > outfile.txt.

break

A "breakpoint" is a spot in your program where you would like to temporarily stop execution in order to check the values of variables, or to try to find out where the program is crashing, etc. To set a breakpoint you use the break command.

break *function* sets the breakpoint at the beginning of *function*. If your code is in multiple files, you might need to specify *filename:function*.

break *linenumber* or break *filename:linenumber* sets the breakpoint to the given line number in the source file. Execution will stop before that line has been executed.

delete

delete will delete all breakpoints that you have set.

delete *number* will delete breakpoint numbered *number*. You can find out what number each breakpoint is by doing info breakpoints. (The command info can also be used to find out a lot of other stuff. Do help info for more information.)

clear

clear *function* will delete the breakpoint set at that function. Similarly for *linenumber*, *filename:function*, and *filename:linenumber*.

continue

continue will set the program running again, after you have stopped it at a breakpoint.

step

step will go ahead and execute the current source line, and then stop execution again before the next source line.

next

next will continue until the next source line in the current function (actually, the current innermost stack frame, to be precise). This is similar to step, except that if the line about to be executed is a function call, then that function call will be completely executed before execution stops again, whereas with step execution will stop at the first line of the function that is called.

until

until is like next, except that if you are at the end of a loop, until will continue execution until the loop is exited, whereas next will just take you back up to the beginning of the loop. This is convenient if you want to see what happens after the loop, but don't want to step through every iteration.

list

list *linenumber* will print out some lines from the source code around *linenumber*. If you give it the argument *function* it will print out lines from the beginning of that function. Just list without any arguments will print out the lines just after the lines that you printed out with the previous list command.

print

print *expression* will print out the value of the expression, which could be just a variable name.

To print out the first 25 (for example) values in an array called list, do

```
print list[0]@25
```

Gprof

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.

How to use gprof

Using the gprof tool is not at all complex. You just need to do the following on a high-level:

- Have profiling enabled while compiling the code
- Execute the program code to produce the profiling data
- Run the gprof tool on the profiling data file (generated in the step above).

Lets try and understand the three steps listed above through a practical example. Following test code will be used throughout the article :

```
//test_gprof.c
#include<stdio.h>
void new_func1(void);
void func1(void)
{
    printf("\n Inside func1 \n");
    int i = 0;
    for(;i<0xffffffff;i++);
    new_func1();
    return;
}
static void func2(void)
{
    printf("\n Inside func2 \n");
    int i = 0;
    for(;i<0xfffffaa;i++);
    return;
}
int main(void)
{
    printf("\n Inside main()\n");
    int i = 0;
    for(;i<0xfffff;i++);
    func1();
    func2();
    return 0;
}
//test_gprof_new.c
#include<stdio.h>
```

```
void new_func1(void)
{
    printf("\n Inside new_func1()\n");
    int i = 0;
    for(;i<0xffffffff;i++);
    return;
}
```

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.

lets compile our code with ‘-pg’ option :

```
$ gcc -Wall -pg test_gprof.c test_gprof_new.c -o test_gprof
```

Please note : The option ‘-pg’ can be used with the gcc command that compiles (-c option), gcc command that links(-o option on object files) and with gcc command that does the both(as in example above).

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. As produced above, all the profiling information is now present in 'analysis.txt'. Lets have a look at this text file :

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

% 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 (explanation follows)

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]
-----
```

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
```

So (as already discussed) we see that this file is broadly divided into two parts :

1. Flat profile
2. Call graph

The individual columns for the (flat profile as well as call graph) are very well explained in the output itself.

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
```

Note that I have used (and will be using) -b option so as to avoid extra information in analysis output.

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

The program was executed and the result was successfully obtained. Thus, CO1 was obtained.

Experiment No.: 2**Aim**

Merge two sorted arrays and store in a third array.

CO1

Use Basic Data Structures and its operations implementations.

Algorithm

1. Start
2. Input two arrays a[] and b[] with size n and n1.
3. Sort arrays
 - 3.1. for(i=0;i<n;i++)
 - 3.2. for(s=i+1;s<n;s++)
 - 3.3. if(a[i]>a[s])
 - 3.4. temp=a[i]
 - 3.5. a[i]=a[s]
 - 3.6. a[s]=temp
4. Merge sorted arrays
 - 4.1. p=n+n1;
 - 4.2 For i=0 to n
 - 4.3 c[k]=a[i]
k++
 - 4.4 for j=0 to n1
 - 4.5 c[k]=b[j];
k++
5. Stop

Procedure

```
#include<stdio.h>

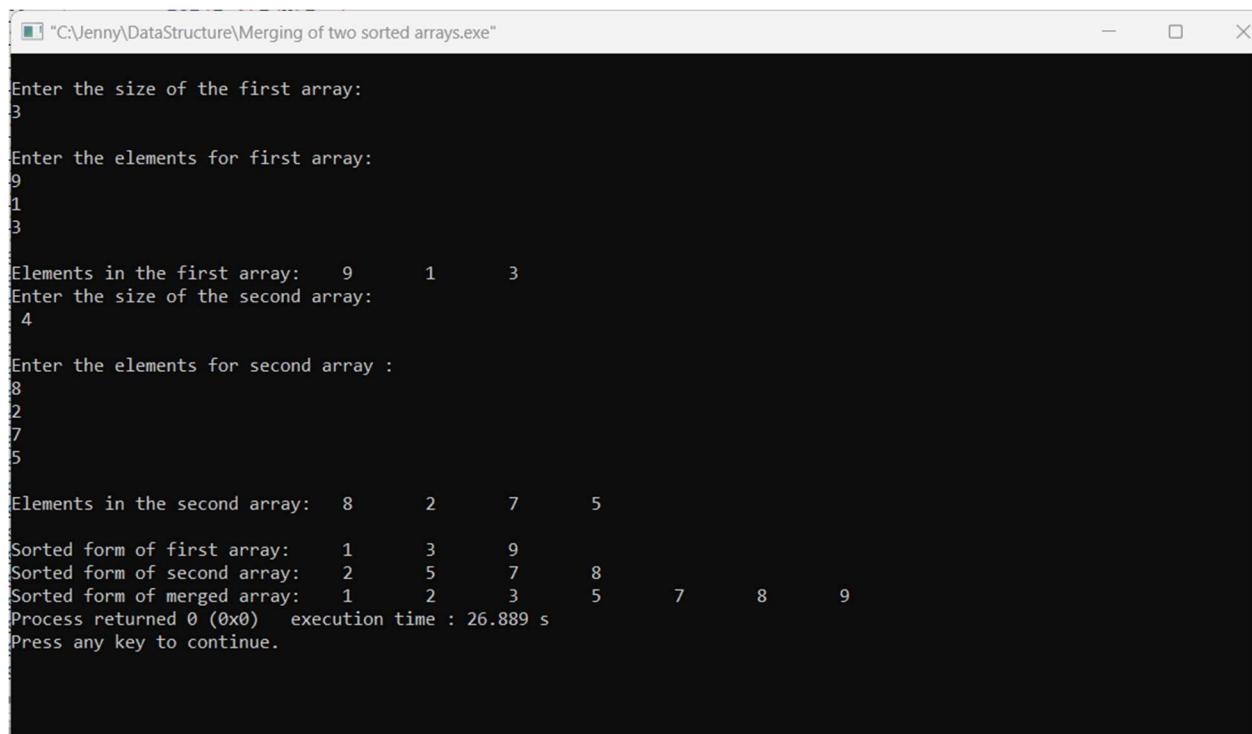
int main()
{
    int n,a[10],b[10],c[20],i,j,k,p,n1,s,t,temp,templ;
    printf("\nEnter the size of the first array:\n");
    scanf("%d",&n);
    printf("\nEnter the elements for first array:\n");
    for(i=0;i<n;i++)
    {
        scanf("%d",&a[i]);
    }
    printf("\nElements in the first array: ");
    for(i=0;i<n;i++)
    {
        printf("\t%d",a[i]);
    }
    printf("\nEnter the size of the second array:\n ");
    scanf("\t%d",&n1);
    printf("\nEnter the elements for second array :\n");
    for(j=0;j<n1;j++)
    {
        scanf("%d",&b[j]);
    }
    printf("\nElements in the second array: ");
    for(j=0;j<n1;j++)
    {
```

```
printf("\t%d",b[j]);
    }
    for(i=0;i<n;i++)
    {
        for(s=i+1;s<n;s++)
        {
            if(a[i]>a[s])
            {
                temp=a[i];
                a[i]=a[s];
                a[s]=temp;
            }
        }
    }
    printf("\n\nSorted form of first array:");
    for(i=0;i<n;i++)
    {
        printf("\t%d",a[i]);
    }
    for(j=0;j<n1;j++)
    {
        for(t=j+1;t<n1;t++)
        {
            if(b[j]>b[t])
            {
                temp1=b[j];
                b[j]=b[t];
                b[t]=temp1;
            }
        }
    }
    printf("\nSorted form of second array:");
```

```
for(j=0;j<n1;j++)
{
printf("\t%d",b[j]);
}
p=n+n1;
for(i=0;i<n;i++)
{
c[k]=a[i];
k++;
}
for(j=0;j<n1;j++)
{
c[k]=b[j];
k++;
}
for(j=0;j<p;j++)
{
for(t=j+1;t<p;t++)
{
if(c[j]>c[t])
{
temp1=c[j];
c[j]=c[t];
c[t]=temp1;
}}}
printf("\nSorted form of merged array:");
for(k=0;k<p;k++)
{
```

```
printf("\t%d",c[k]);  
    }  
return 0;  
}
```

Output Screenshot



```
"C:\Jenny\DataStructure\Merging of two sorted arrays.exe"  
Enter the size of the first array:  
3  
Enter the elements for first array:  
9  
1  
3  
Elements in the first array: 9 1 3  
Enter the size of the second array:  
4  
Enter the elements for second array :  
8  
2  
7  
5  
Elements in the second array: 8 2 7 5  
Sorted form of first array: 1 3 9  
Sorted form of second array: 2 5 7 8  
Sorted form of merged array: 1 2 3 5 7 8 9  
Process returned 0 (0x0) execution time : 26.889 s  
Press any key to continue.
```

Result

The program was executed and the result was successfully obtained. Thus, CO1 was obtained.

Experiment No.: 3**Aim**

Implementation of Singly Linked Stack.

CO1

Use Basic Data Structures and its operations implementations.

Algorithm

1. Start
2. Declare a structure containing a data part as well as an address part
3. Present a menu of operations push,pop,display to the users by switch-case
4. If the operation is push()
 - 4.1. Check whether array is empty or not
 - 4.1.1. If empty create a node in newnode and assign TOP=newnode
 - 4.1.1. Store a value in newnode->data
 - 4.1.2. Else, create a temporary node and assign struct node *temp=TOP
 - 4.1.2.1. While tem->next!=NULL, traverse the linked stack
 - 4.1.2.1.1. Attach a newnode with a value at the end position
 - 4.2. Go back to step 3
5. If operation is pop()
 - 5.1. Check whether list is empty or top is NULL
 - 5.1.1 If yes, print ‘ stack underflow’, else free the last element by iterating list.
 - 5.2. Display the linked stack elements
 - 5.3. Go back to step 3
6. If operation is display, traverse the list items one by one and print the values
7. If operation is for exit, quit the menu and return
8. Stop

Procedure

```
#include<stdio.h>
#include<stdlib.h>
void push();
void pop();
void display();
struct stacknode
{
int data;
struct stacknode *next;
}*top=NULL;
void main()
{
int opt;
do
{
printf("\n SELECT AN OPERATION\n");
printf("\n 1. PUSH OPERATION\n");
printf("\n 2. POP OPERATION\n");
printf("\n 3. VIEW LINKED STACK\n");
printf("\n 4. EXIT\n");
scanf("%d",&opt);
switch(opt)
{
case 1: push();
break;
case 2: pop();
break;
```

```
case 3: display();
break;
case 4: exit(0);
default: printf("\n Invalid Option\n");
}}
while(opt!=4);
}
void push()
{
struct stacknode *newnode;
newnode=(struct stacknode*)malloc(sizeof(struct stacknode));
printf("\n Enter a value :\n");
scanf("%d",&newnode->data);
if(top==NULL)
{
newnode->next=NULL;
top=newnode;
}
else
{
newnode->next=top;
top=newnode;
}
printf("\n Stack elements are\n");
display();
}
void pop()
{
```

```
if(top==NULL)
printf("\n Stack Underflow, Insert element\n");
else
{
struct stacknode *temp=top;
top=temp->next;
printf("\n The top element %d has been popped out...\n",temp->data);
free(temp);
printf("\nStack elements are \n");
display();
}}
void display()
{
struct stacknode *temp=top;
if(top==NULL)
printf("\n STACK IS EMPTY!\n");
else
{
while(temp!=NULL)
{
printf("%d\t",temp->data);
temp=temp->next;
}
}
}
```

Output Screenshot

```

C:\Jenny\DataStructure\linked_stack.exe
SELECT AN OPERATION
1. PUSH OPERATION
2. POP OPERATION
3. VIEW LINKED STACK
4. EXIT
3
STACK IS EMPTY!
SELECT AN OPERATION
1. PUSH OPERATION
2. POP OPERATION
3. VIEW LINKED STACK
4. EXIT
1
Enter a value :
12
Stack elements are
12
SELECT AN OPERATION
1. PUSH OPERATION
2. POP OPERATION
3. VIEW LINKED STACK
4. EXIT
1
Enter a value :
34
Stack elements are
34 12

```

```

3. VIEW LINKED STACK
4. EXIT
1
Enter a value :
23
Stack elements are
23 34 12
SELECT AN OPERATION
1. PUSH OPERATION
2. POP OPERATION
3. VIEW LINKED STACK
4. EXIT
1
Enter a value :
5
Stack elements are
5 23 34 12
SELECT AN OPERATION
1. PUSH OPERATION
2. POP OPERATION
3. VIEW LINKED STACK
4. EXIT
2
The top element 5 has been popped out...
Stack elements are
23 34 12
SELECT AN OPERATION
1. PUSH OPERATION
2. POP OPERATION
3. VIEW LINKED STACK
4. EXIT

```

Result

The program was executed and the result was successfully obtained. Thus, CO1 was obtained.

Experiment No.: 4**Aim**

To implement Circular queue.

CO1

Use Basic Data Structures and its operations implementations.

Algorithm

1. Define a structure to implement a node
2. Declare front and rear as NULL
3. If operation is enqueue(),
 - a. If rear==null, insert newnode to front , set front and rear as address of newnode
 - b. Else travel the list using temporary pointer and attach newnode at end
4. If operation is dequeue(),
 - a. If front==NULL, print ‘ list is empty’
 - b. Otherwise set front as address of second node, front = front->next
5. If display(),
 - a. While (temp->rear!=NULL) , print(DATA(temp)

Procedure

```
#include<stdio.h>
#include<stdlib.h>
void enqueue();
void dequeue();
void display();
struct node
{
int data;
struct node *next;
```

```
*front=NULL;
struct node *rear=NULL;
struct node *newnode;
void create_node()
{
newnode=(struct node*)malloc(sizeof(struct node));
printf("\n Enter the value: ");
scanf("%d",&newnode->data);
}
void main()
{
int opt;
do
{
printf("\n SELECT AN OPERATION\n");
printf("\n1. ENQUEUE\n");
printf("\n2. DEQUEUE\n");
printf("\n3. DISPLAY\n");
printf("\n4. EXIT\n");
scanf("%d",&opt);
switch(opt)
{
case 1: enqueue();
break;
case 2: dequeue();
break;
case 3: display();
break;
```

```
case 4: exit(0);
default: printf("\n Invalid Choice\n");
}
}
while(opt!=4);
}
void enqueue()
{
create_node();
if(front==NULL && rear==NULL)
{
front=newnode;
rear=newnode;
newnode->next=front;
}
else
{
struct node *temp=front;
while(temp->next!=front)
temp=temp->next;
newnode->next=temp->next;
temp->next=newnode;
rear=newnode;
}
display();
}
void dequeue()
{
```



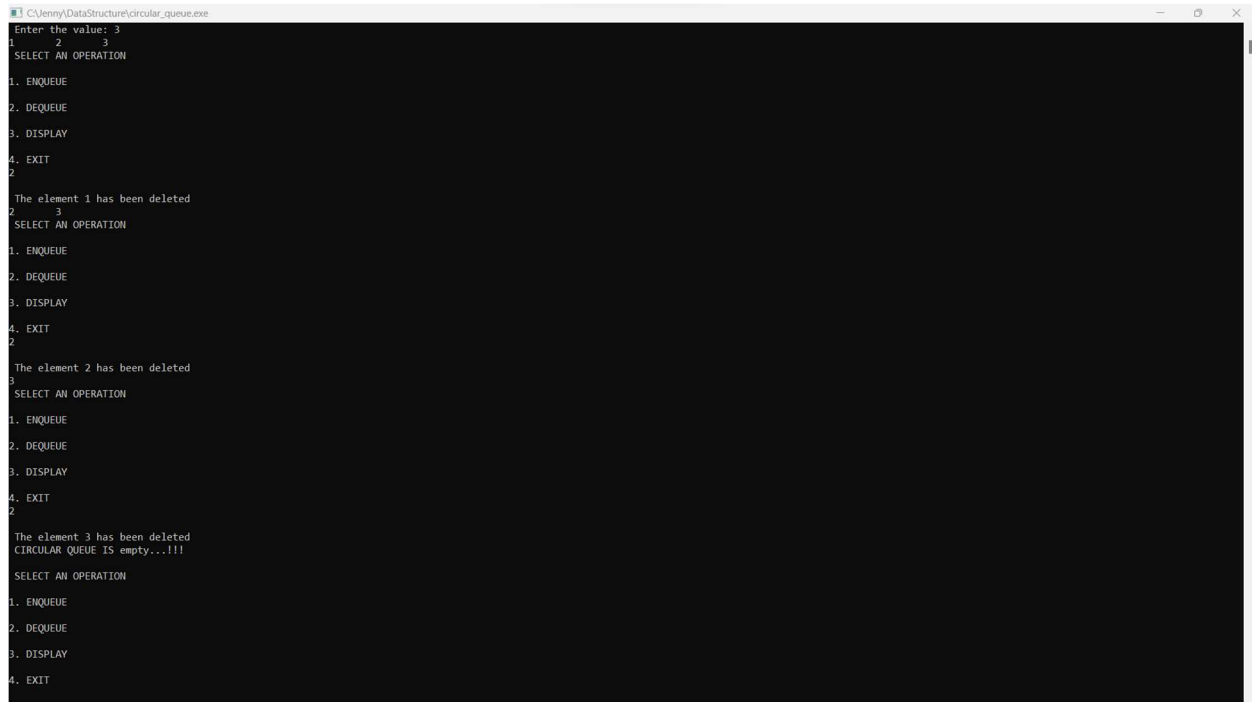
```
if(front==NULL&&rear==NULL)
printf("\n QUEUE IS empty\n");
else
{
if(front->next==front)
{
printf("\n The element %d has been deleted",front->data);
front=NULL;
rear=NULL;
}
else
{
struct node *temp=front;
struct node *temp1=front;
while(temp->next!=front)
temp=temp->next;
temp->next=front->next;
front=front->next;
printf("\n The element %d has been deleted\n",temp1->data);
free(temp1);
}
display();
}
}

void display()
{
if(front==NULL)
printf("\n CIRCULAR QUEUE IS empty...!!!\n");
```

```
else
{
    struct node *temp=front;
    while(temp->next!=front)
    {
        printf("%d\t",temp->data);
        temp=temp->next;
    }
    printf("%d\t",temp->data);
}}
```

Output Screenshot

```
C:\Venny\DataStructure\circular_queue.exe
SELECT AN OPERATION
1. ENQUEUE
2. DEQUEUE
3. DISPLAY
4. EXIT
3
CIRCULAR QUEUE IS empty....!!!
SELECT AN OPERATION
1. ENQUEUE
2. DEQUEUE
3. DISPLAY
4. EXIT
1
Enter the value: 1
1
SELECT AN OPERATION
1. ENQUEUE
2. DEQUEUE
3. DISPLAY
4. EXIT
1
Enter the value: 2
2
SELECT AN OPERATION
1. ENQUEUE
2. DEQUEUE
3. DISPLAY
4. EXIT
1
Enter the value: 3
```



```
C:\Venny\DataStructure\circular_queue.exe
Enter the value: 3
1 2 3
SELECT AN OPERATION

1. ENQUEUE
2. DEQUEUE
3. DISPLAY
4. EXIT
2

The element 1 has been deleted
2 3
SELECT AN OPERATION

1. ENQUEUE
2. DEQUEUE
3. DISPLAY
4. EXIT
2

The element 2 has been deleted
3
SELECT AN OPERATION

1. ENQUEUE
2. DEQUEUE
3. DISPLAY
4. EXIT
2

The element 3 has been deleted
CIRCULAR QUEUE IS empty...!!!

SELECT AN OPERATION

1. ENQUEUE
2. DEQUEUE
3. DISPLAY
4. EXIT
```

Result

The program was executed and the result was successfully obtained. Thus, CO1 was obtained.

Experiment No.: 5**Aim**

To implement a doubly linked list.

CO1

Use Basic Data Structures and its operations implementations.

Algorithm

1. START
2. Display a menu of operations
3. If choice is for insertion
 - 3.1. If beginning
 - 3.1.1.set previous node of newnode to NULL
 - 3.1.2.set next node of newnode to head
 - 3.1.3.set head to point to newnode
 - 3.2. If end
 - 3.2.1.set temp to point to the head (temp=head)
 - 3.2.2.Travel the doubly linked list till the next of temp is null
 - 3.2.3.insert newnode into temp->next
 - 3.2.4.set newnode->next as NULL
 - 3.2.5.set newnode->previous=temp
 - 3.3. If particular position
 - 3.3.1.set temp to point to the first node (head)
 - 3.3.2.travel till the desired position reach
 - 3.3.3.insert the newnode at next to the temp
 - 3.3.4.set previous pointer of newnode to temp
 - 3.3.5.set next pointer of newnode to the current next of temp
 - 3.3.6.set next pointer of temp to newnode
4. If choice is for deletion
 - 4.1. If beginning

- 4.1.1. make temp point to first node
- 4.1.2. set previous link of next link of temp to be NULL
- 4.1.3. Set head pointing to the node next to temp;
- 4.1.4. set NULL at next pointer of temp
- 4.2. If end
 - 4.2.1. assign temp as newnode
 - 4.2.2. traverse the list till next of temp equal to
 - 4.2.3. set next pointer of previous link of temp to NULL
 - 4.2.4. set previous pointer of temp to NULL, and free temp
- 4.3. If specific position
 - 4.3.1. Set temp pointing to the first node (head)
 - 4.3.2. Read a logical position from which the node is to be removed
 - 4.3.3. Remove the desired node by traversing the list
 - 4.3.4. Set the next pointer of node residing before temp as the node after temp
 - 4.3.5. Set previous link of node after temp pointing to node before temp
- 5. If choice is for searching an item in the list, traverse the list by checking that whether the item is matching with the part of current node
 - 5.1. Assign temp = head
 - 5.2. while(temp->data!=item), traverse by temp=temp->next
 - 5.3. If data in temp is equal to item, print 'item found in list'
 - 5.4. Otherwise print "item does not exist"
- 6. If operation is for traversal or display
 - 6.1. Set temp as head
 - 6.2. Print the data contained in temp while reaches last node
- 7. If user's choice is none other than above, print 'invalid choice'
- 8. Continue steps 2 to 7 till the user input an option for exit
- 9. STOP

Procedure

```
#include<stdio.h>
#include<stdlib.h>
int count=0;
void insert_begin();
void insert_end();
void insert_pos();
void delete_begin();
void delete_end();
void delete_pos();
void search_key();
void traverse_list();
struct node
{
int data;
struct node *prev;
struct node *next;
}*head=NULL;
void main()
{
int opt,item;
do
{
printf("\n SELECT AN OPERATION\n");
printf("\n1. INSERTION AT BEGINNING\n");
printf("\n2. INSERTION AT END\n");
printf("\n3. INSERTION AT A GIVEN POSITION\n");
printf("\n4. DELETION FROM BEGINNING\n");
```

```
printf("\n5. DELETION FROM END\n");
printf("\n6. DELETION FROM A GIVEN POSITION\n");
printf("\n7. SEARCH FOR AN ITEM\n");
printf("\n8. DISPLAY LIST\n");
printf("\n9. EXIT\n");
scanf("%d",&opt);
switch(opt)
{
case 1: insert_begin();
break;
case 2: insert_end();
break;
case 3: insert_pos();
break;
case 4: delete_begin();
break;
case 5: delete_end();
break;
case 6: delete_pos();
break;
case 7: search_key();
break;
case 8: traverse_list();
break;
case 9: exit(0);
default: printf("\n Invalid Option\n");
}
}
```

```
while(opt!=9);
}
void insert_begin()
{
int item;
printf("\n Enter a value: ");
scanf("%d",&item);
struct node *newnode;
newnode=(struct node*)malloc(sizeof(struct node));
newnode->data=item;
if(head==NULL)
{
head=newnode;
newnode->prev=NULL;
newnode->next=NULL;
count++;
}
else
{
struct node *temp=head;
temp->prev=newnode;
newnode->prev=NULL;
newnode->next=temp;
head=newnode;
count++;
}
printf("\n The items in the list are:\n");
traverse_list();
```

```
}  
  
void insert_end()  
{  
    int item;  
    printf("\n Enter a value: ");  
    scanf("%d",&item);  
    struct node *newnode;  
    newnode=(struct node*)malloc(sizeof(struct node));  
    newnode->data=item;  
    if(head==NULL)  
    {  
        head=newnode;  
        newnode->prev=NULL;  
        newnode->next=NULL;  
        count++;  
    }  
    else  
    {  
        struct node *temp=head;  
        while(temp->next!=NULL)  
            temp=temp->next;  
        temp->next=newnode;  
        newnode->prev=temp;  
        newnode->next=NULL;  
        count++;  
    }  
    printf("\n The items in the list are\n");  
    traverse_list();  
}
```

```
void insert_pos()
{
int item,pos,i=1;
struct node *temp=head;
printf("\n Enter a value: ");
scanf("%d",&item);
struct node *newnode;
newnode=(struct node*)malloc(sizeof(struct node));
newnode->data=item;
printf("\n Enter the position to which the new node is to be inserted: ");
scanf("%d",&pos);
if(pos>count)
{
printf("\n Invalid position\n");
}
while(temp->next!=NULL&&i!=pos-1)
{
temp=temp->next;
i++;
}
if(i==pos-1)
{
newnode->next=temp->next;
temp->next=newnode;
newnode->prev=temp;
count++;
}
else
```

```
{
if(pos==count)
{
while(temp->next!=NULL)
temp=temp->next;
temp->next=newnode;
newnode->next=NULL;
newnode->prev=temp;
count++;
}
else
printf("\n POSITION not found\n");
}
printf("\n The items in the list are\n");
traverse_list();
}

void delete_begin()
{
struct node *temp=head;
if(head==NULL)
printf("\n Doubly linked list is empty\n");
else
{
if(temp->next==NULL)
{
temp->prev=NULL;
head=NULL;
printf("\n The item %d has been deleted\n",temp->data);
free(temp);
```

```
count--;

    traverse_list();
}

else
{
    head=temp->next;
    temp->next->prev=NULL;
    temp->prev=NULL;
    temp->next=NULL;
    printf("\n The item %d has been deleted from beginning\n",temp->data);
    free(temp);
    count--;
    printf("\n The items in the list are\n");
    traverse_list();
}

}

}

void delete_end()
{
    struct node *temp=head;
    if(head==NULL)
        printf("\n Doubly linked list is empty\n");
    else if(temp->next==NULL)
    {
        printf("\n The item %d has been deleted\n",temp->data);
        temp->prev=NULL;
        temp->next=NULL;
        head=NULL;
    }
}
```

```
free(temp);
count--;
printf("\n The items in the list are\n");
traverse_list();
}
else
{
while(temp->next!=NULL)
temp=temp->next;
temp->prev->next=NULL;
temp->prev=NULL;
printf("\n The item %d has been deleted from end\n",temp->data);
free(temp);
count--;
printf("\n The items in the list are\n");
traverse_list();
}
}
void delete_pos()
{
int pos,i=1;
struct node *temp=head;
if(head==NULL)
printf("\n The doubly linked list is empty\n");
else
{
printf("\n Enter the position of node to be deleted: ");
scanf("%d",&pos);
```

```
if(pos>count)
    printf("\n Position is not within the list\n");
else
{
    while(temp->next!=NULL&&pos!=i)
    {
        temp=temp->next;
        i++;
    }
    temp->prev->next=temp->next;
    temp->prev=NULL;
    temp->next=NULL;
    printf("\n The item %d has been deleted",temp->data);
    free(temp);
    count--;
    printf("\n The items in the doubly linked list are\n");
    traverse_list();
}
}
}

void traverse_list()
{
    struct node *temp=head;
    if(head==NULL)
        printf("\n List is empty\n");
    else
    {
        while(temp!=NULL)
```

```
{  
printf("%d\t",temp->data);  
temp=temp->next;  
}  
}  
}  
  
void search_key()  
{  
int item;  
printf("\n Enter an item to be searched: \n");  
scanf("%d",&item);  
struct node *temp=head;  
while(temp->data!=item&&temp->next!=NULL)  
temp=temp->next;  
if(temp->data==item)  
printf("\n The item %d found in the list",item);  
else  
printf("\n The item %d not found in the list\n",item);  
}
```

```
C:\Jenny\DataStructure\doubly_linkedList.exe
SELECT AN OPERATION
1. INSERTION AT BEGINNING
2. INSERTION AT END
3. INSERTION AT A GIVEN POSITION
4. DELETION FROM BEGINNING
5. DELETION FROM END
6. DELETION FROM A GIVEN POSITION
7. SEARCH FOR AN ITEM
8. DISPLAY LIST
9. EXIT
1
Enter a value: 34
The items in the list are:
34
SELECT AN OPERATION
1. INSERTION AT BEGINNING
2. INSERTION AT END
3. INSERTION AT A GIVEN POSITION
4. DELETION FROM BEGINNING
5. DELETION FROM END
6. DELETION FROM A GIVEN POSITION
7. SEARCH FOR AN ITEM
8. DISPLAY LIST
9. EXIT
2
Enter a value: 56
The items in the list are
34    56
SELECT AN OPERATION
```

```
C:\Jenny\DataStructure\ doubly_linkedList.exe
The items in the list are:
34
SELECT AN OPERATION
1. INSERTION AT BEGINNING
2. INSERTION AT END
3. INSERTION AT A GIVEN POSITION
4. DELETION FROM BEGINNING
5. DELETION FROM END
6. DELETION FROM A GIVEN POSITION
7. SEARCH FOR AN ITEM
8. DISPLAY LIST
9. EXIT
2
Enter a value: 56
The items in the list are:
34 56
SELECT AN OPERATION
1. INSERTION AT BEGINNING
2. INSERTION AT END
3. INSERTION AT A GIVEN POSITION
4. DELETION FROM BEGINNING
5. DELETION FROM END
6. DELETION FROM A GIVEN POSITION
7. SEARCH FOR AN ITEM
8. DISPLAY LIST
9. EXIT
3
Enter a value: 67
Enter the position to which the new node is to be inserted: 2
```



```
C:\Jenny\DataStructure\doubly_linkedlist.exe
Enter the position to which the new node is to be inserted: 2
The items in the list are
34    67    56
SELECT AN OPERATION
1. INSERTION AT BEGINNING
2. INSERTION AT END
3. INSERTION AT A GIVEN POSITION
4. DELETION FROM BEGINNING
5. DELETION FROM END
6. DELETION FROM A GIVEN POSITION
7. SEARCH FOR AN ITEM
8. DISPLAY LIST
9. EXIT
6
Enter the position of node to be deleted: 2
The item 67 has been deleted
The items in the doubly linked list are
34    56
SELECT AN OPERATION
1. INSERTION AT BEGINNING
2. INSERTION AT END
3. INSERTION AT A GIVEN POSITION
4. DELETION FROM BEGINNING
5. DELETION FROM END
6. DELETION FROM A GIVEN POSITION
7. SEARCH FOR AN ITEM
8. DISPLAY LIST
9. EXIT
4
The item 34 has been deleted from beginning

The item 34 has been deleted from beginning
The items in the list are
56
SELECT AN OPERATION
1. INSERTION AT BEGINNING
2. INSERTION AT END
3. INSERTION AT A GIVEN POSITION
4. DELETION FROM BEGINNING
5. DELETION FROM END
6. DELETION FROM A GIVEN POSITION
7. SEARCH FOR AN ITEM
8. DISPLAY LIST
9. EXIT
5
The item 56 has been deleted
The items in the list are
List is empty
SELECT AN OPERATION
1. INSERTION AT BEGINNING
2. INSERTION AT END
3. INSERTION AT A GIVEN POSITION
4. DELETION FROM BEGINNING
5. DELETION FROM END
6. DELETION FROM A GIVEN POSITION
7. SEARCH FOR AN ITEM
8. DISPLAY LIST
9. EXIT
```

Result

The program was executed and the result was successfully obtained. Thus, CO1 was obtained.

Experiment No.: 6**Aim**

To implement Binary Search Tree.

CO3

Understand the practical aspects of Advanced Tree Structures.

Algorithm

1. Start
2. Define a structure for BST
3. Display a menu of operations
4. If inorder traversal
 - 4.1. Visit the left child
 - 4.2. Process the node currently accessed
 - 4.3. Visit the right child
5. If preorder traversal
 - 5.1. Process the node currently accessed
 - 5.2. Visit the left child
 - 5.3. Visit the right child
6. If postorder traversal
 - 6.1. Visit the left child
 - 6.2. Visit the right child
 - 6.3. Process the currently visited node
7. If insertion operaton
 - 7.1. Read a value in key
 - 7.2. If root is NULL, insert new node as root
 - 7.3. Else check if key less than root node
 - 7.3.1. Insert the newnode at left of root node
 - 7.3.2. Else insert newnode at the right of root node

8. If search operation

8.1. Read an item to be searched in item

8.2. Check if item lesser or greater than the root

8.3. If item lesser than root node value

8.3.1. Perform recursive search on the left subtree

8.3.2. Else perform recursive search on the right subtree

9. If deletion operation

9.1. Read a key to be deleted from the bst

9.2. If key is lesser than the root node's value

9.2.1. If the element to be deleted is parent node

9.2.1.1. Replace it with inorder successor

9.2.1.2. Else replace it with inorder predecessor

9.2.2. If element to be deleted is leaf node, simply delete key

10. Stop

Procedure

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

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

```
struct node
```

```
{
```

```
int data;
```

```
struct node *l;
```

```
struct node *r;
```

```
}*root=NULL,*temp=NULL,*t1,*t2;
```

```
void insert();
```

```
void create();
```

```
void search(struct node *t);
```

```
void search1(struct node *t,int data);
```

```
void inorder(struct node *t);
void delete();
void delete1();
int smallest(struct node *t);
int largest(struct node *t);
int flag = 1;
int main()
{
int ch;
printf("\nOPERATIONS ---");
printf("\n1.Insert an element into tree\n");
printf("2.Inorder Traversal\n");
printf("3.Delete a node \n");
printf("4.Exit\n");
do
{
printf("\nEnter your choice : ");
scanf("%d", &ch);
switch (ch)
{
case 1:
insert();
break;
case 2:inorder(root);
break;
case 3:delete();
break;
case 6:printf("\nInvalid option\n");
```

```
exit(0);
default :
printf("Wrong choice, Please enter correct choice ");
break;
}
}while(ch<4);
}
void insert()
{
create();
if (root == NULL)
root = temp;
else
search(root);
}
void create()
{
int data;
printf("Enter data of node to be inserted : ");
scanf("%d", &data);
temp = (struct node *)malloc(1*sizeof(struct node));
temp->data = data;
temp->l = temp->r = NULL;
}
void search(struct node *t)
{
if ((temp->data > t->data) && (t->r != NULL))
search(t->r);
```

```
else if ((temp->data > t->data) && (t->r == NULL))
t->r = temp;
else if ((temp->data < t->data) && (t->l != NULL))
search(t->l);
else if ((temp->data < t->data) && (t->l == NULL))
t->l = temp;
}
void inorder(struct node *t)
{
if (root == NULL)
{
printf("No elements in a tree to display");
return;
}
if (t->l != NULL)
inorder(t->l);
printf("%d ->", t->data);
if (t->r != NULL)
inorder(t->r);
}
void delete()
{
int data;
if (root == NULL)
{
printf("No elements in a tree to delete");
return;
}
```

```
printf("Enter the data to be deleted : ");
scanf("%d", &data);
t1 = root;
t2 = root;
search1(root, data);
}
void search1(struct node *t, int data)
{
if ((data>t->data))
{
t1 = t;
search1(t->r, data);
}
else if ((data < t->data))
{
t1 = t;
search1(t->l, data);
}
else if ((data==t->data))
{
delete1(t);
}
}
void delete1(struct node *t)
{
int k;
if ((t->l == NULL) && (t->r == NULL))
{
```

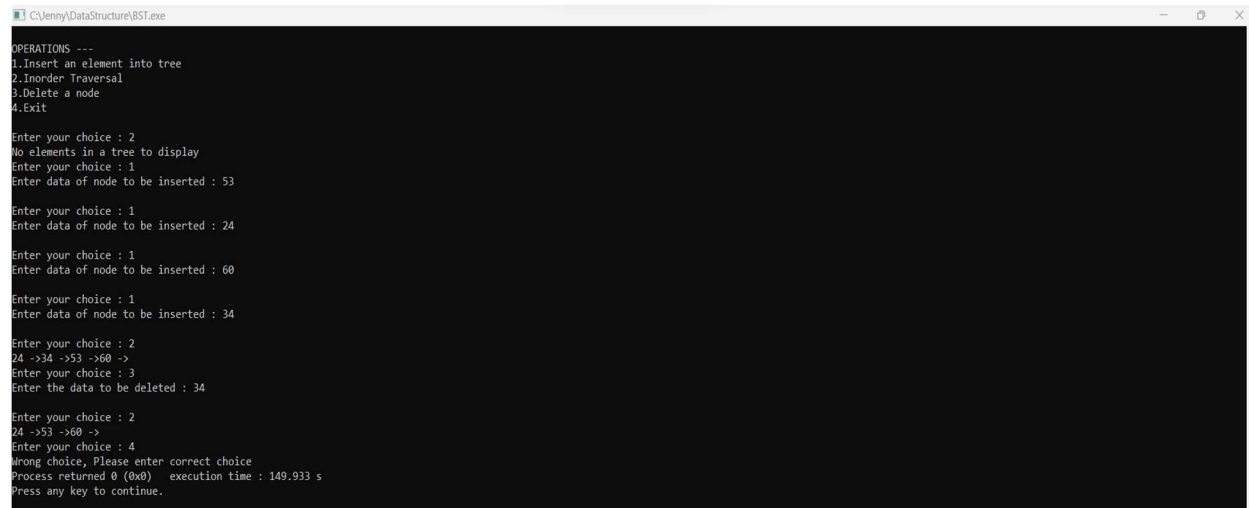
```
if (t1->l == t)
{
t1->l = NULL;
}
else
{
t1->r = NULL;
}
t = NULL;
free(t);
return;
}
else if ((t->r == NULL))
{
if (t1 == t)
{
root = t->l;
t1 = root;
}
else if (t1->l == t)
{
t1->l = t->l;
}
else
{
t1->r = t->l;
}
t = NULL;
free(t);
```



```
return;
}
else if (t->l == NULL)
{
if (t1 == t)
{
root = t->r;
t1 = root;
}
else if (t1->r == t)
t1->r = t->r;
else
t1->l = t->r;
t == NULL;
free(t);
return;
}
else if ((t->l != NULL) && (t->r != NULL))
{
t2 = root;
if (t->r != NULL)
{
k = smallest(t->r);
flag = 1;
}
else
{
k = largest(t->l);
```

```
flag = 2;
}
search1(root, k);
t->data = k;
}
}
int smallest(struct node *t)
{
t2 = t;
if (t->l != NULL)
{
t2 = t;
return(smallest(t->l));
}
else
return (t->data);
}
int largest(struct node *t)
{
if (t->r != NULL)
{
t2 = t;
return(largest(t->r));
}
else
return(t->data);
}
```

Output Screenshot



```
OPERATIONS ---
1.Insert an element into tree
2.Inorder Traversal
3.Delete a node
4.Exit

Enter your choice : 2
No elements in a tree to display
Enter your choice : 1
Enter data of node to be inserted : 53

Enter your choice : 1
Enter data of node to be inserted : 24

Enter your choice : 1
Enter data of node to be inserted : 60

Enter your choice : 1
Enter data of node to be inserted : 34

Enter your choice : 2
24 ->34 ->53 ->60 ->
Enter your choice : 3
Enter the data to be deleted : 34

Enter your choice : 2
24 ->53 ->60 ->
Enter your choice : 4
Wrong choice, Please enter correct choice
Process returned 0 (0x0)   execution time : 149.933 s
Press any key to continue.
```

Result

The program was executed and the result was successfully obtained. Thus, CO3 was obtained.

Experiment No.: 7**Aim**

To implement Set data structure and set operations using Bit Strings.

CO2

Implement the Set and Disjoint Set Data Structures.

Algorithm

1. Start
2. Create two character array
3. Enter a bit string in array1
4. Enter another bit string in array2
5. Display menu of operations
6. If union():
 - a. For i =0 to strlen(array): print(array1[i] or array2[i])
7. If intersection():
 - a. For i=0 to strlen(array): print(array1[i] and array2[i])
8. If set difference():
 - a. Declare an array3 for complementing array2
 - b. Store bitwise negation results on array2 in array3
 - c. For i=0 to strlen(array): print(array1[i] and array3[i])
9. Stop

Procedure

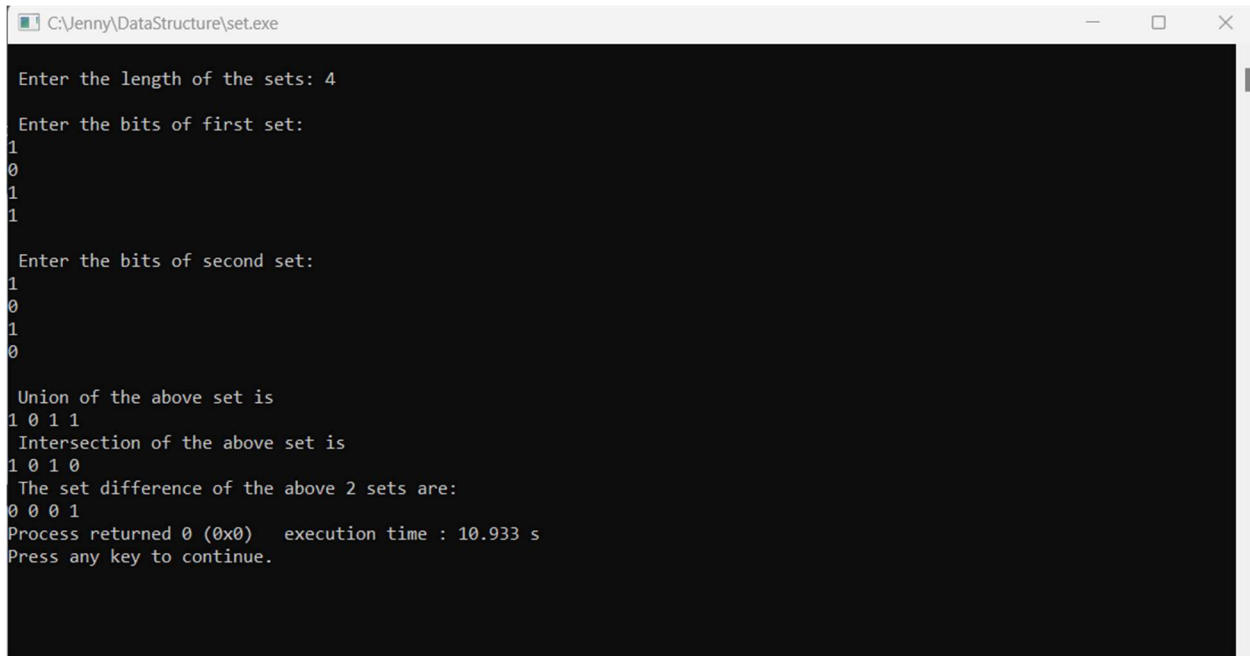
```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
int main()
```

```
{
int len,i=0;
int str1[10],str2[10],str3[10];
int opt,c,d;
printf("\n Enter the length of the sets: ");
scanf("%d",&len);
printf("\n Enter the bits of first set:\n");
for(i=0;i<len;i++)
{
scanf("%d",&c);
if(c<0 || c>1)
printf("\n Input Error \n Please enter in binary format\n");
else
str1[i]=c;
}
printf("\n Enter the bits of second set:\n");
for(i=0;i<len;i++)
{
scanf("%d",&d);
if(d<0 || d>1)
printf("\n Input Error\n Please enter in binary format\n");
else
str2[i]=d;
}
printf("\n Union of the above set is\n");
for(i=0;i<len;i++)
{
if((str1[i]||str2[i])==1)
```

```
printf("1 ");

else
printf("0 ");
}
printf("\n Intersection of the above set is\n");
for(i=0;i<len;i++)
{
if((str1[i]&&str2[i])==1)
printf("1 ");
else
printf("0 ");
}
printf("\n The set difference of the above 2 sets are:\n");
for(i=0;i<len;i++)
{
str3[i]=!(str2[i]);
str3[i]=str1[i]&&str3[i];
printf("%d ",str3[i]);
}
}
```

Output Screenshot



```
C:\Jenny\DataStructure\set.exe

Enter the length of the sets: 4

Enter the bits of first set:
1
0
1
1

Enter the bits of second set:
1
0
1
0

Union of the above set is
1 0 1 1
Intersection of the above set is
1 0 1 0
The set difference of the above 2 sets are:
0 0 0 1
Process returned 0 (0x0)   execution time : 10.933 s
Press any key to continue.
```

Result

The program was executed and the result was successfully obtained. Thus, CO2 was obtained.

Experiment No.: 8**Aim**

Implementation of Binomial Heap.

CO4

Realise Modern Heap Structures for effectively solving advanced Computational problems.

Algorithm**Insertion**

1. create a new binomial heap H1 of one node of value x
2. Unite H1 and H.

Insert Binomial Heap ()

- 1: SET H' = Create Binomial-Heap ()
- 2: SET Parent(x) = NULL,
Child(x) = NULL and
sibling(x) = NULL,
Degree (x) = NULL
- 3: SET Head(H'] = x
- 4: SET Head(H] = Union_Binomial-Heap (H, H')
- 5: END

Time Complexity - $O(\log n)$

Union

1. Merge H1 and H2, i.e. link the roots of H1 and H2 in non-decreasing order.
2. Restoring binomial heap by linking binomial trees of the same degree together: traverse the linked list, keep track of three pointers, prev, pt and next.
Case 1 degrees of ptr and next are not same, move ahead.
Case 2 If degree of next->next is also same, move ahead.
Case 3 If key of ptr is smaller than or equal to key of next, make next as a child of ptr by linking it with ptr.
Case 4: If key of ptr is greater than next, then make ptr as child of next.

Procedure

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

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

```
#include <limits.h>
```

```
struct node
```

```
{
    int key;
    int degree;
    struct node *sibling;
    struct node *child;
};

struct node *newNode(int key)
{
    struct node *temp = (struct node *)malloc(sizeof(struct node));
    temp->key = key;
    temp->degree = 0;
    temp->sibling = temp->child = NULL;
    return temp;
}

struct node *mergeBinomialTrees(struct node *a, struct node *b)
{
    if (a == NULL)
        return b;
    if (b == NULL)
        return a;
    struct node *result;
    if (a->degree <= b->degree)
    {
        result = a;
        result->sibling = mergeBinomialTrees(a->sibling, b);
    }
    else
    {
        result = b;
```

```
result->sibling = mergeBinomialTrees(a, b->sibling);
    }
    return result;
}

struct node *unionBinomialHeap(struct node *head1, struct node *head2)
{
    struct node *head = mergeBinomialTrees(head1, head2);
    if (head == NULL)
        return head;
    struct node *prev = NULL, *curr = head, *next = curr->sibling;
    while (next != NULL)
    {
        if (curr->degree != next->degree || (next->sibling != NULL && next->sibling->degree ==
curr->degree))
        {
            prev = curr;
            curr = next;
        }
        else
        {
            if (curr->key <= next->key)
            {
                curr->sibling = next->sibling;
                next->sibling = curr->child;
                curr->child = next;
                curr->degree++;
            }
            else
            {

```

```
if (prev == NULL)
    head = next;
else
    prev->sibling = next;
curr->sibling = next->child;
next->child = curr;
next->degree++;
curr = next;
}
}
next = curr->sibling;
}
return head;
}

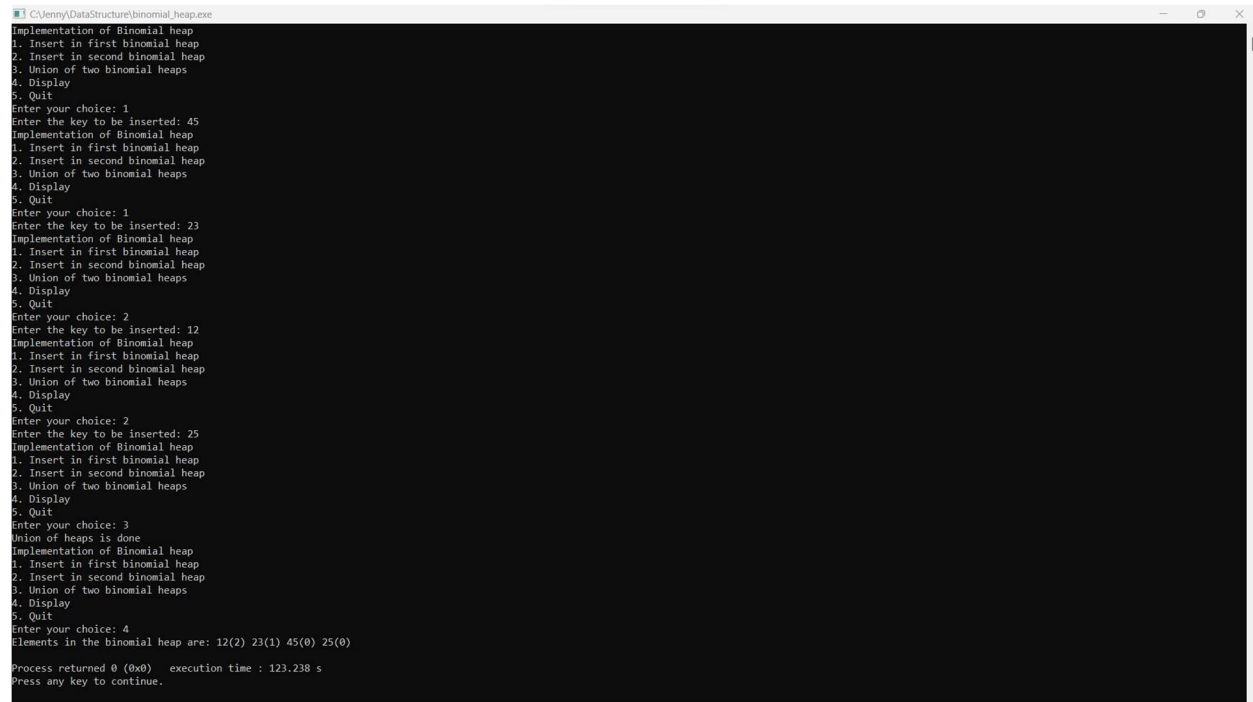
void insert(struct node **head, int key)
{
    struct node *temp = newNode(key);
    *head = unionBionomialHeap(*head, temp);
}

void display(struct node *head)
{
    if (head == NULL)
        return;
    struct node *temp = head;
    while (temp != NULL)
    {
        printf("%d(%d) ", temp->key, temp->degree);
        display(temp->child);
    }
}
```

```
temp = temp->sibling;
    }
}
int main()
{
    struct node *head1 = NULL;
    struct node *head2 = NULL;
    int choice, key;
    while (1)
    {
        printf("Implementation of Binomial heap\n");
        printf("1. Insert in first binomial heap\n");
        printf("2. Insert in second binomial heap\n");
        printf("3. Union of two binomial heaps\n");
        printf("4. Display\n");
        printf("5. Quit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice)
        {
            case 1:
                printf("Enter the key to be inserted: ");
                scanf("%d", &key);
                insert(&head1, key);
                break;
            case 2:
                printf("Enter the key to be inserted: ");
                scanf("%d", &key);
```

```
insert(&head2, key);
    break;
case 3:
    head1 = unionBionomialHeap(head1, head2);
    printf("Union of heaps is done\n");
    break;
case 4:
    printf("Elements in the binomial heap are: ");
    display(head1);
    printf("\n");
case 5:
    exit(0);
default:
    printf("Wrong choice\n");
    break;
}
}
return 0;
}
```

Output Screenshot



```

C:\Jenny\DataStructure\binomial_heap.exe
Implementation of Binomial heap
1. Insert in first binomial heap
2. Insert in second binomial heap
3. Union of two binomial heaps
4. Display
5. Quit
Enter your choice: 1
Enter the key to be inserted: 45
Implementation of Binomial heap
1. Insert in first binomial heap
2. Insert in second binomial heap
3. Union of two binomial heaps
4. Display
5. Quit
Enter your choice: 1
Enter the key to be inserted: 23
Implementation of Binomial heap
1. Insert in first binomial heap
2. Insert in second binomial heap
3. Union of two binomial heaps
4. Display
5. Quit
Enter your choice: 2
Enter the key to be inserted: 12
Implementation of Binomial heap
1. Insert in first binomial heap
2. Insert in second binomial heap
3. Union of two binomial heaps
4. Display
5. Quit
Enter your choice: 2
Enter the key to be inserted: 25
Implementation of Binomial heap
1. Insert in first binomial heap
2. Insert in second binomial heap
3. Union of two binomial heaps
4. Display
5. Quit
Enter your choice: 3
Union of heaps is done
Implementation of Binomial heap
1. Insert in first binomial heap
2. Insert in second binomial heap
3. Union of two binomial heaps
4. Display
5. Quit
Enter your choice: 4
Elements in the binomial heap are: 12(2) 23(1) 45(0) 25(0)

Process returned 0 (0x0)   execution time : 123.238 s
Press any key to continue.
```

Result

The program was executed and the result was successfully obtained. Thus, CO4 was obtained.

Experiment No.: 9**Aim**

Implementation of Depth First Search.

CO5

Implement Advanced Graph algorithms suitable for solving advanced computational problems.

Algorithm

1. Define a Stack of size total number of vertices in the graph.
2. Select any vertex as starting point for traversal. Visit that vertex and push it on to the Stack.
3. Visit any one of the non-visited adjacent vertices of a vertex which is at the top of stack and push it on to the stack.
4. Repeat step 3 until there is no new vertex to be visited from the vertex which is at the top of the stack.
5. When there is no new vertex to visit then use back tracking and pop one vertex from the stack.
6. Repeat steps 3, 4 and 5 until stack becomes Empty.
7. When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph.

Procedure

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#define MAX 6
int vertex_count = 0;
struct vertex {
    char data;
    bool visited;
};
struct vertex *graph[MAX];
int adj_matrix[MAX][MAX];
```

```
int stack[MAX];
int top = -1;
void push(int data){
    stack[++top]=data;
}
int pop(){
    return stack[top--];
}
int peek(){
    return stack[top];
}
bool is_stack_empty(){
    return top == -1;
}
void add_vertex(char data){
    struct vertex *new = (struct vertex*)malloc(sizeof(struct vertex));
    new->data = data;
    new->visited = false;
    graph[vertex_count]=new;
    vertex_count++;
}
void add_edge(int start,int end){
    adj_matrix[start][end]=1;
    adj_matrix[end][start]=1;
}
int adj_vertex(int vertex_get){
    int i;
    for(i=0;i<vertex_count;i++){
```



```
if(adj_matrix[vertex_get][i] == 1 && graph[i]->visited == false){
    return i;
}
return -1;
}

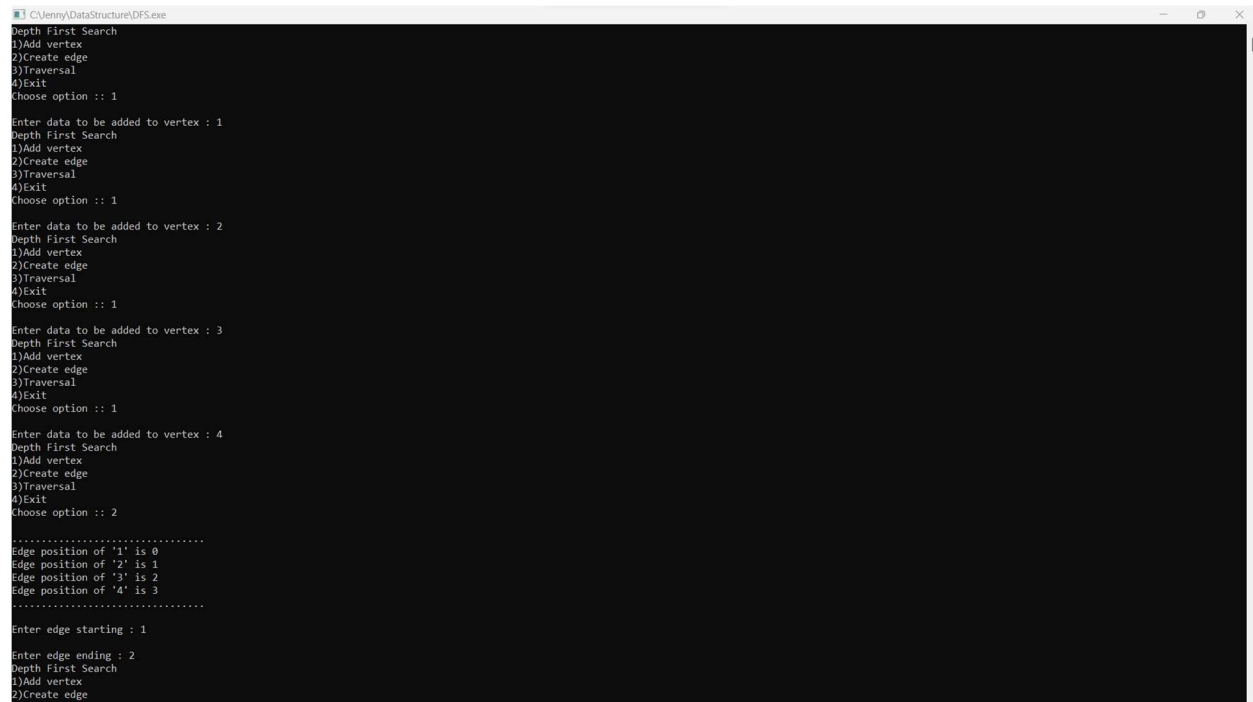
void display_vertex(int pos){
    printf("%c",graph[pos]->data);
}

void dfs(){
    int i;
    int unvisited;
    printf("\n||||||||||||||||||||\n");
    graph[0]->visited =true;
    display_vertex(0);
    push(0);
    while(!is_stack_empty()){
        int unvisited = adj_vertex(peek());
        if(unvisited == -1){
            pop();
        }
        else{
            graph[unvisited]->visited = true;
            display_vertex(unvisited);
            push(unvisited);
        }
    }
    printf("\n||||||||||||||||||||\n");
    for(i=0;i<vertex_count;i++){
        graph[i]->visited = false;
    }
}
```

```
    }}  
void show(){  
    int i;  
    printf("\n.....\n");  
    for(i=0;i<vertex_count;i++){  
        printf("Edge position of '%c' is %d\n",graph[i]->data,i);  
    }  
    printf(".....\n");  
}  
int main(){  
    int opt;  
    char data;  
    int edge_1,edge_2;  
    int i, j;  
    for(i = 0; i < MAX; i++)  
        for(j = 0; j < MAX; j++)  
            adj_matrix[i][j] = 0;  
    do{  
        printf("Depth First Search");  
        printf("\n1)Add vertex \n2)Create edge \n3)Traversal \n4)Exit \nChoose option :: ");  
        scanf("%d",&opt);  
        switch(opt){  
            case 1:  
                printf("\nEnter data to be added to vertex : ");  
                scanf(" %c", &data);  
                add_vertex(data);  
                break;  
            case 2:  
                show();  
            case 3:  
                break;  
            case 4:  
                return 0;  
            default:  
                break;  
        }  
    }while(opt < 5);  
}
```

```
printf("\nEnter edge starting : ");
    scanf("%d",&edge_1);
    printf("\nEnter edge ending : ");
    scanf("%d",&edge_2);
    if(vertex_count-1 < edge_1 || vertex_count-1 < edge_2){
        printf("\nThere is no vertex !!\n");
    }
    else{
        add_edge(edge_1,edge_2);
    }
    break;
case 3:
    dfs();
case 4:
    exit(0);
    break;
default:
    printf("\nInvalid option try again !! ...");
}
}
while(opt!=0);
    return 0;
}
```

Output Screenshot



```
C:\Venny\DataStructure\DFS.exe
Depth First Search
1)Add vertex
2)Create edge
3)Traversal
4)Exit
Choose option :: 1

Enter data to be added to vertex : 1
Depth First Search
1)Add vertex
2)Create edge
3)Traversal
4)Exit
Choose option :: 1

Enter data to be added to vertex : 2
Depth First Search
1)Add vertex
2)Create edge
3)Traversal
4)Exit
Choose option :: 1

Enter data to be added to vertex : 3
Depth First Search
1)Add vertex
2)Create edge
3)Traversal
4)Exit
Choose option :: 1

Enter data to be added to vertex : 4
Depth First Search
1)Add vertex
2)Create edge
3)Traversal
4)Exit
Choose option :: 2

.....
Edge position of '1' is 0
Edge position of '2' is 1
Edge position of '3' is 2
Edge position of '4' is 3
.....

Enter edge starting : 1

Enter edge ending : 2
Depth First Search
1)Add vertex
2)Create edge
```

Result

The program was executed and the result was successfully obtained. Thus, CO5 was obtained.

Experiment No.: 10**Aim**

Implementation of Breadth First Search.

CO5

Implement Advanced Graph algorithms suitable for solving advanced computational problems.

Algorithm

1. Define a Queue of size total number of vertices in the graph.
2. Select any vertex as starting point for traversal. Visit that vertex and insert it into the Queue.
3. Visit all the non-visited adjacent vertices of the vertex which is at front of the Queue and insert them into the Queue.
4. When there is no new vertex to be visited from the vertex which is at front of the Queue then delete that vertex.
5. Repeat steps 3 and 4 until queue becomes empty.
6. When queue becomes empty, then produce final spanning tree by removing unused edges from the graph.

Procedure

```
#include<stdio.h>
#include<stdlib.h>
#include<stdbool.h>
#define MAX 10
int vertex_count =0;
struct vertex {
    char data;
    bool visited;
};
struct vertex *graph[MAX];
int adj_matrix[MAX][MAX];
```

```
int queue[MAX];
int rear=-1;
int front=0;
int queue_count=0;
void enqueue(int data){
    queue[++rear]=data;
    queue_count++;
}
int dequeue(){
    queue_count--;
    return queue[front++];
}
bool is_queue_empty(){
    return queue_count == 0;
}
void add_vertex(char data){
    struct vertex *new = (struct vertex*)malloc(sizeof(struct vertex));
    new->data = data;
    new->visited = false;
    graph[vertex_count]=new;
    vertex_count++;
}
void add_edge(int start,int end){
    adj_matrix[start][end]=1;
    adj_matrix[end][start]=1;
}
int adj_vertex(int vertex_get){
    int i;
```

```
for(i=0;i<vertex_count;i++){
    if(adj_matrix[vertex_get][i] == 1 && graph[i]->visited == false){
        return i;
    }
}
return -1;
}

void display_vertex(int pos){
    printf("%c -> ",graph[pos]->data);
}

void bfs(struct vertex *new,int start){
    if(!new){
        printf("\nNothing to display\n");
        return;
    }
    int i;
    int unvisited;
    printf("\n||||||||||||||||||||||||||||\n");
    new->visited =true;
    display_vertex(start);
    enqueue(start);
    while(!is_queue_empty()){
        int pop_vertex = dequeue();
        //printf("\npoped : %d",pop_vertex);
        while((unvisited = adj_vertex(pop_vertex))!=-1){
            graph[unvisited]->visited = true;
            display_vertex(unvisited);
            enqueue(unvisited);
        }
    }
}
```

```

    }

    printf("\n||||||||||||||||||||\n");

    for(i=0;i<vertex_count;i++){
        graph[i]->visited = false;
    }

void show(){
    int i;

    printf("\n.....\n");

    for(i=0;i<vertex_count;i++){
        printf("Edge position of '%c' is %d\n",graph[i]->data,i);
    }

    printf(".....\n");
}

int main(){
    int opt;

    char data;

    int edge_1,edge_2;

    int i, j;

    int start;

    for(i = 0; i < MAX; i++) // set adjacency
        for(j = 0; j < MAX; j++) // matrix to 0
            adj_matrix[i][j] = 0;

    do{
        printf("\n1)Add vertex \n2)Create edge \n3)Traversal \n4)Exit \nChoose option ::
");

        scanf("%d",&opt);

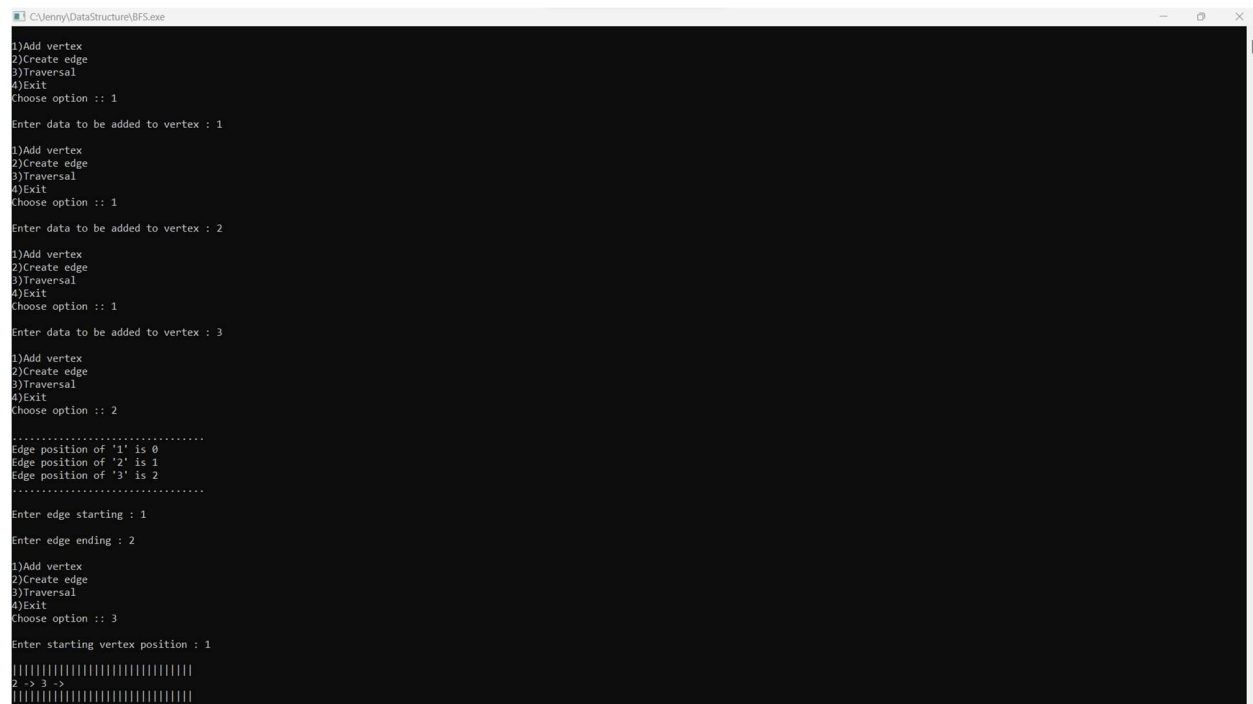
        switch(opt){
            case 1:
                printf("\nEnter data to be added to vertex : ");
                scanf(" %c", &data);

```

```
        add_vertex(data);
        break;
    case 2:
        show();

        printf("\nEnter edge starting : ");
        scanf("%d",&edge_1);
        printf("\nEnter edge ending : ");
        scanf("%d",&edge_2);
        if(vertex_count-1 < edge_1 || vertex_count-1 < edge_2){
            printf("\nThere is no vertex !!\n");
        }
        else{
            add_edge(edge_1,edge_2);
        }
        break;
    case 3:
        printf("\nEnter starting vertex position : ");
        scanf("%d",&start);
        bfs(graph[start],start);
        break;
    case 4:
        exit(0);
    default:
        printf("\nInvalid option try again !! ...");
    }
} while(opt!=0);
return 0;
}
```

Output Screenshot



```
C:\Venny\DataStructure\BFS.exe
1)Add vertex
2)Create edge
3)Traversal
4)Exit
Choose option :: 1
Enter data to be added to vertex : 1
1)Add vertex
2)Create edge
3)Traversal
4)Exit
Choose option :: 1
Enter data to be added to vertex : 2
1)Add vertex
2)Create edge
3)Traversal
4)Exit
Choose option :: 1
Enter data to be added to vertex : 3
1)Add vertex
2)Create edge
3)Traversal
4)Exit
Choose option :: 2
.....
Edge position of '1' is 0
Edge position of '2' is 1
Edge position of '3' is 2
.....
Enter edge starting : 1
Enter edge ending : 2
1)Add vertex
2)Create edge
3)Traversal
4)Exit
Choose option :: 3
Enter starting vertex position : 1
|||||
2 -> 3 ->
|||||
```

Result

The program was executed and the result was successfully obtained. Thus, CO5 was obtained.

Experiment No.: 11**Aim**

Implementation of Prim's Algorithm.

CO5

Implement Advanced Graph algorithms suitable for solving advanced computational problems.

Algorithm

MST PRIMS(G, w, t)

1. For each $u \in V[G]$
 2. Do $key[u] \leftarrow -\infty$
 3. $\Pi[u] \leftarrow \text{NIL}$
 4. $Key[\Pi] \leftarrow 0$
 5. $Q \leftarrow V[G]$
 6. While $Q \neq \emptyset$
- Do $u \leftarrow \text{extract min}(Q)$
- For each $V \in \text{Adj}[u]$
- Do if $V \in Q$ and $w[u, v] < key[V]$
- Then $\Pi[V] \leftarrow u$
- $Key[V] \leftarrow w[u, v]$

Procedure

```
#include <limits.h>
#include <stdbool.h>
#include <stdio.h>
#define V 5
int minKey(int key[], bool mstSet[])
{
    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;
}
int printMST(int parent[], int graph[V][V])
{

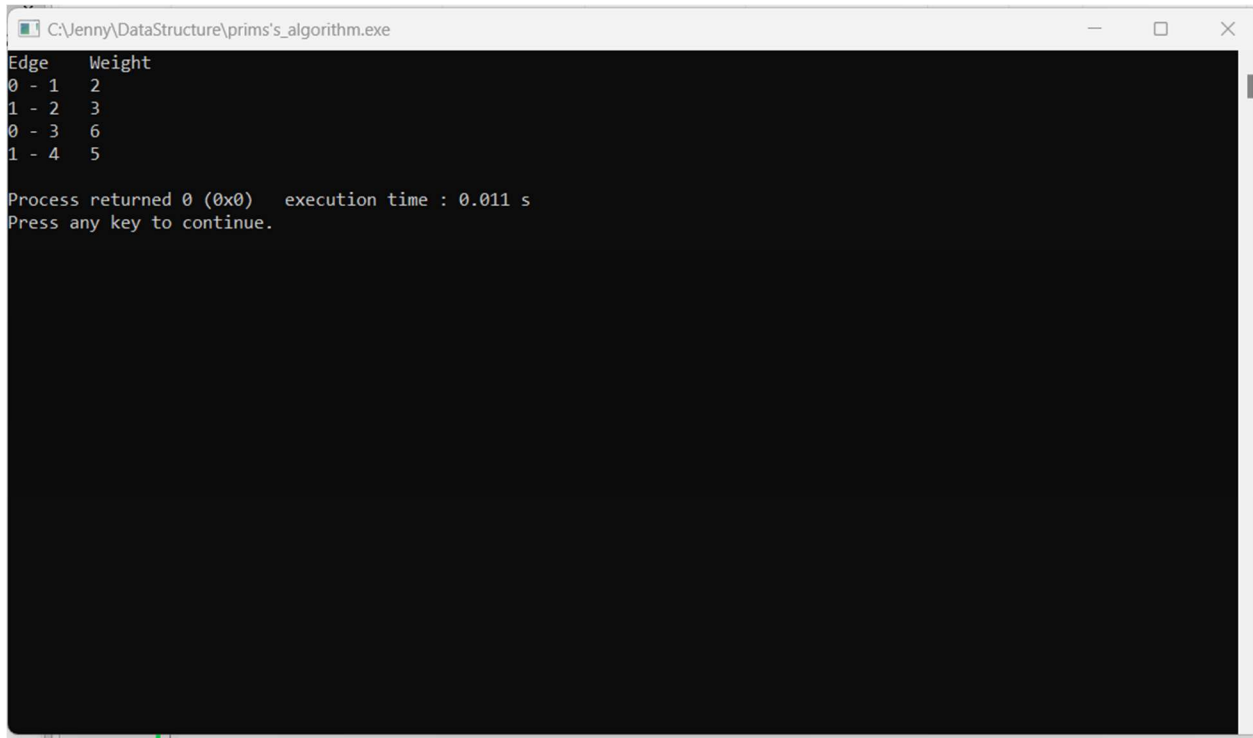
```

```
printf("Edge \tWeight\n");
for (int i = 1; i < V; i++)
    printf("%d - %d \t%d \n", parent[i], i, graph[i][parent[i]]);
}

void primMST(int graph[V][V])
{
    int parent[V];
    int key[V];
    bool mstSet[V];
    for (int i = 0; i < V; i++)
        key[i] = INT_MAX, mstSet[i] = false;
    key[0] = 0;
    parent[0] = -1;
    for (int count = 0; count < V - 1; count++) {
        int u = minKey(key, mstSet);
        mstSet[u] = true;
        for (int v = 0; v < V; v++)
            if (graph[u][v] && mstSet[v] == false && graph[u][v] < key[v])
                parent[v] = u, key[v] = graph[u][v];
    }
    printMST(parent, graph);
}

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 } };
    primMST(graph);
    return 0;
}
```

Output Screenshot



```
C:\Jenny\DataStructure\prims's_algorithm.exe

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

Process returned 0 (0x0)   execution time : 0.011 s
Press any key to continue.
```

Result

The program was executed and the result was successfully obtained. Thus, CO5 was obtained.

Experiment No.: 12**Aim**

Implementation of Kruskal's Algorithm.

CO5

Implement Advanced Graph algorithms suitable for solving advanced computational problems.

Algorithm

- 1.Start
- 2.KRUSKAL(G):
 $A = \emptyset$
- 3.For each vertex $v \in G.V$:
 MAKE-SET(v)
- 4.For each edge $(u, v) \in G.E$ ordered by increasing order by weight(u, v):
 if FIND-SET(u) \neq FIND-SET(v):
5. $A = A \cup \{(u, v)\}$
 UNION(u, v)
- 6.return A
- 7.Stop

Procedure

```
#include <stdio.h>

#define MAX 30

typedef struct edge {
    int u, v, w;
} edge;

typedef struct edge_list {
    edge data[MAX];
    int n;
} edge_list;

edge_list elist;

int Graph[MAX][MAX], n;
```

```
edge_list spanlist;
void kruskalAlgo();
int find(int belongs[], int vertexno);
void applyUnion(int belongs[], int c1, int c2);
void sort();
void print();
void kruskalAlgo() {
    int belongs[MAX], i, j, cno1, cno2;
    elist.n = 0;
    for (i = 1; i < n; i++)
        for (j = 0; j < i; j++) {
            if (Graph[i][j] != 0) {
                elist.data[elist.n].u = i;
                elist.data[elist.n].v = j;
                elist.data[elist.n].w = Graph[i][j];
                elist.n++;
            }
        }
    sort();
    for (i = 0; i < n; i++)
        belongs[i] = i;
    spanlist.n = 0;
    for (i = 0; i < elist.n; i++) {
        cno1 = find(belongs, elist.data[i].u);
        cno2 = find(belongs, elist.data[i].v);
        if (cno1 != cno2) {
            spanlist.data[spanlist.n] = elist.data[i];
            spanlist.n = spanlist.n + 1;
        }
    }
}
```

```
    applyUnion(belongs, cno1, cno2);
    }
}

int find(int belongs[], int vertexno) {
    return (belongs[vertexno]);
}

void applyUnion(int belongs[], int c1, int c2) {
    int i;
    for (i = 0; i < n; i++)
        if (belongs[i] == c2)
            belongs[i] = c1;
}

void sort() {
    int i, j;
    edge temp;
    for (i = 1; i < elist.n; i++)
        for (j = 0; j < elist.n - 1; j++)
            if (elist.data[j].w > elist.data[j + 1].w) {
                temp = elist.data[j];
                elist.data[j] = elist.data[j + 1];
                elist.data[j + 1] = temp;
            }
}

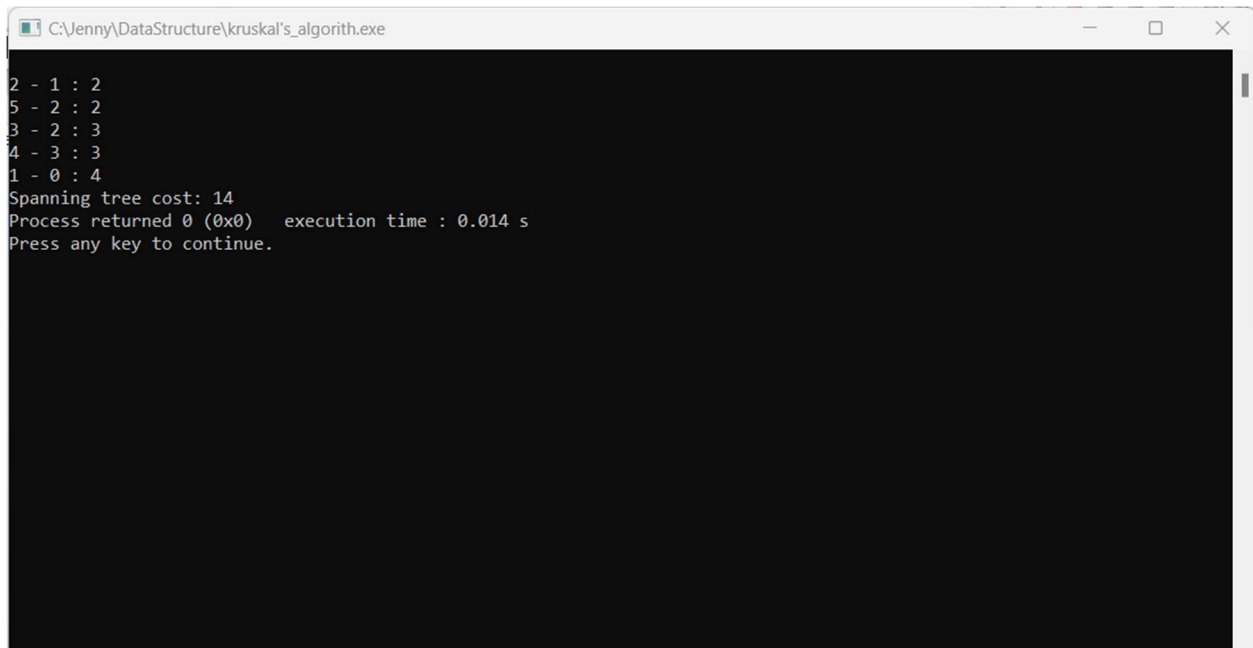
void print() {
    int i, cost = 0;
    for (i = 0; i < spanlist.n; i++) {
        printf("\n%d - %d : %d", spanlist.data[i].u, spanlist.data[i].v, spanlist.data[i].w);
    }
}
```



```
cost = cost + spanlist.data[i].w;
}
printf("\nSpanning tree cost: %d", cost);
}
int main() {
    int i, j, total_cost;
    n = 6;
    Graph[0][0] = 0;
    Graph[0][1] = 4;
    Graph[0][2] = 4;
    Graph[0][3] = 0;
    Graph[0][4] = 0;
    Graph[0][5] = 0;
    Graph[0][6] = 0;
    Graph[1][0] = 4;
    Graph[1][1] = 0;
    Graph[1][2] = 2;
    Graph[1][3] = 0;
    Graph[1][4] = 0;
    Graph[1][5] = 0;
    Graph[1][6] = 0;
    Graph[2][0] = 4;
    Graph[2][1] = 2;
    Graph[2][2] = 0;
    Graph[2][3] = 3;
    Graph[2][4] = 4;
    Graph[2][5] = 0;
    Graph[2][6] = 0;
```

```
Graph[3][0] = 0;
    Graph[3][1] = 0;
    Graph[3][2] = 3;
    Graph[3][3] = 0;
    Graph[3][4] = 3;
    Graph[3][5] = 0;
    Graph[3][6] = 0;
    Graph[4][0] = 0;
    Graph[4][1] = 0;
    Graph[4][2] = 4;
    Graph[4][3] = 3;
    Graph[4][4] = 0;
    Graph[4][5] = 0;
    Graph[4][6] = 0;
    Graph[5][0] = 0;
    Graph[5][1] = 0;
    Graph[5][2] = 2;
    Graph[5][3] = 0;
    Graph[5][4] = 3;
    Graph[5][5] = 0;
    Graph[5][6] = 0;
    kruskalAlgo();
    print();
}
```

Output Screenshot



```
C:\Jenny\DataStructure\kruskal's_algorithm.exe
2 - 1 : 2
5 - 2 : 2
3 - 2 : 3
4 - 3 : 3
1 - 0 : 4
Spanning tree cost: 14
Process returned 0 (0x0) execution time : 0.014 s
Press any key to continue.
```

Result

The program was executed and the result was successfully obtained. Thus, CO5 was obtained.