Experiment No.: 1

Aim

Familiarization with GDB Advanced use of GCC: Important options -o, -c, -D, -1, -I, -g, -O, -save, -temps, -pg Important commands-break, run, next, print, display, help Using gproof: Compile, Execute and Profile

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)

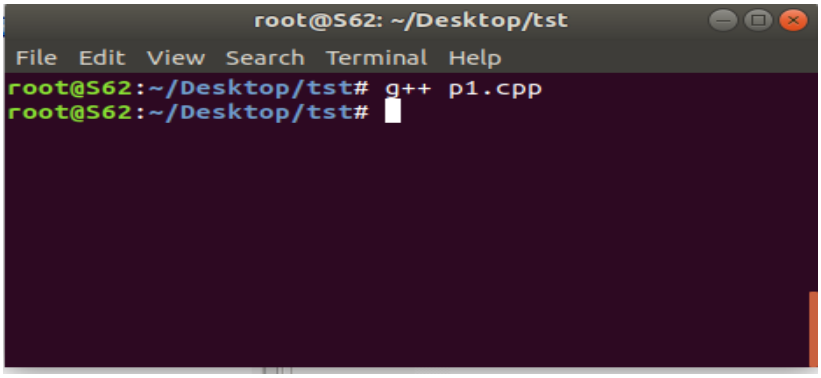
$ sudo g++ -o p1 p1.cpp

1. $ sudo g++ p1.cpp

To compile your c++ code, use:

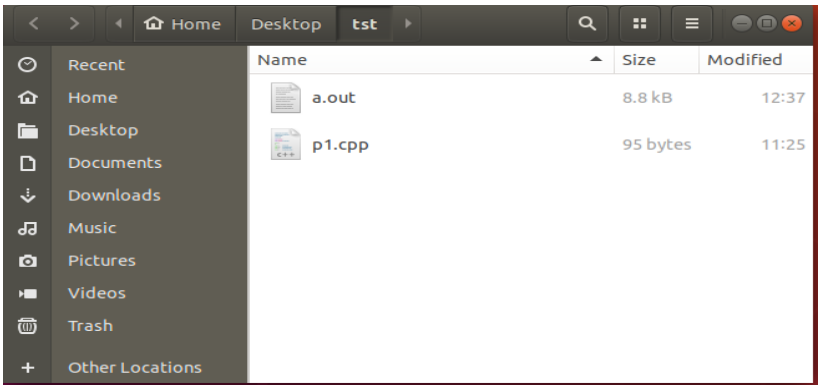
g++ p1.cpp

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:

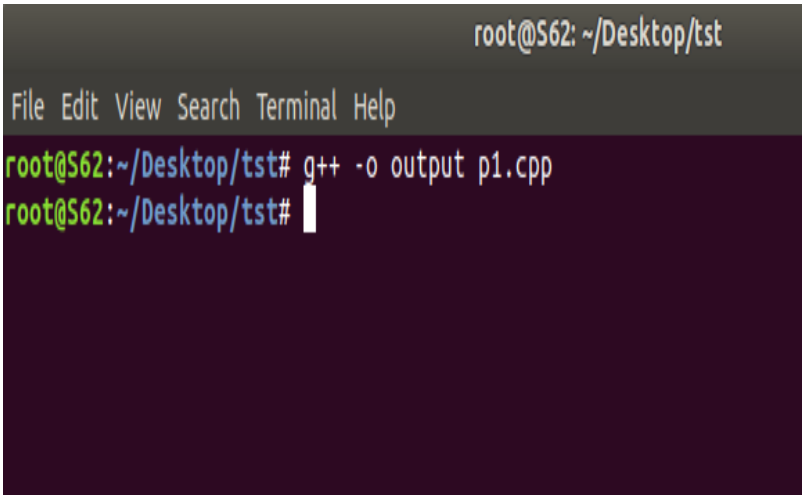
./a.out



$ sudo g++ -o output p1.cpp

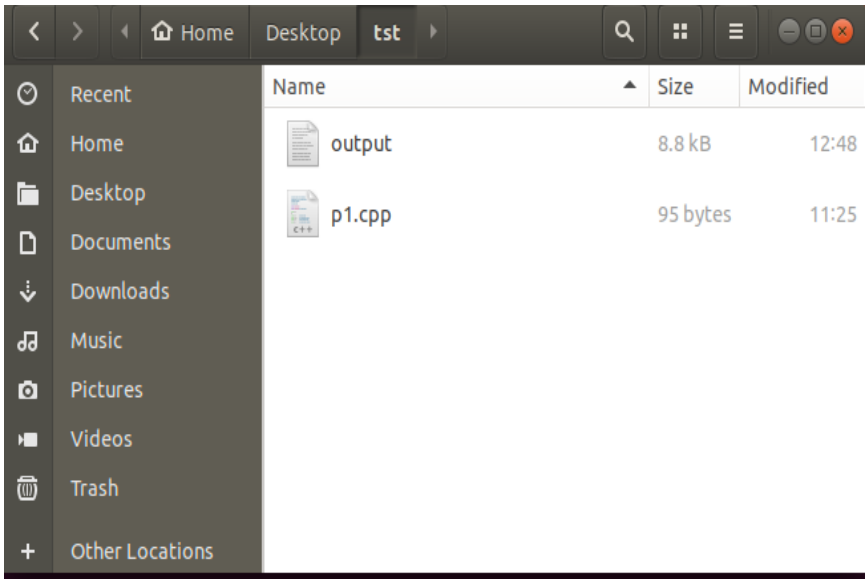
To specify the name of the compiled output file, so that it is not named a.out**, use** -o **with your g++ command.**

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

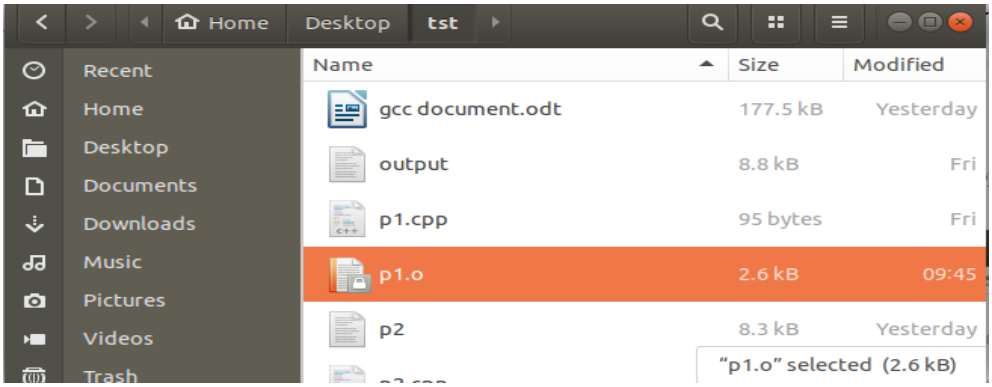


GCC: Important Options

➔ *-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.

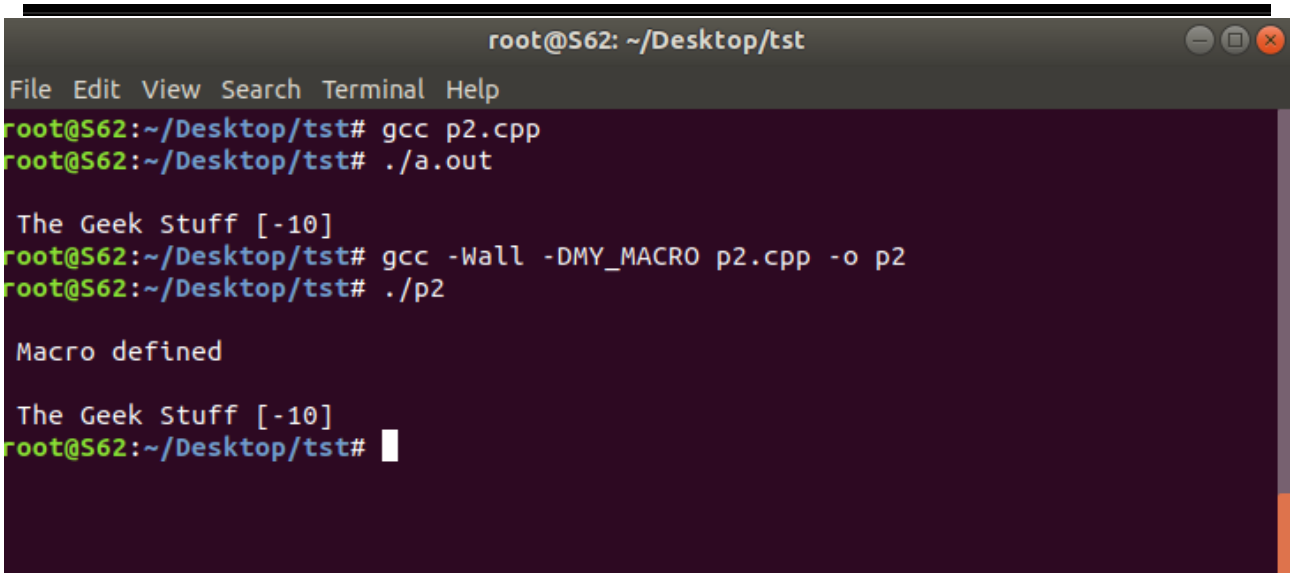
Here is an example :

```
#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.



➔ *-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)
{
  usigned 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<0xffffffaa;i++);
   return;
}
int main(void)
{
   printf("\n Inside main()\n");
   int i = 0;
   for(;i<0xffffff;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<0xfffffee;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

STEP 1: Start
STEP 2: Declare variables
STEP 3: Reed size of first array
STEP 4: Read size of second array
STEP 5: Read elements of first array
STEP 6: Read elements of second array
STEP 7: Print first array
STEP 8: Print second array
STEP 9: Sort first array
for i=0 to i++
for j=i+1 to j++
if(a1[i]>a1[j])
temp = a1[j]
a1[j] = a1[i]
a1[i] = temp
STEP 10: Sort second array
for i=0 to i++
for j=i+1 to j++
if(a2[i]>a2[j])
temp = a2[j]
a2[j] = a1[i]
a2[i] = temp
STEP 11: Print sorted first array
STEP 12: Print sorted second array
STEP 13: Merge first and second array
STEP 14: Print merged array
for i = 1 to i++
a3[i] = a1[j]
a3[i+x] = a2[i]
STEP 15: Sort merged array
for i=0 to i++

```
            for j=i+1 to j++
                  if(a3[i]>a3[j])
                  temp = a3[j]
                  a3[j] = a3[i]
                  a3[i] = temp
```
STEP 16: Print merged and sorted array
STEP 17: End

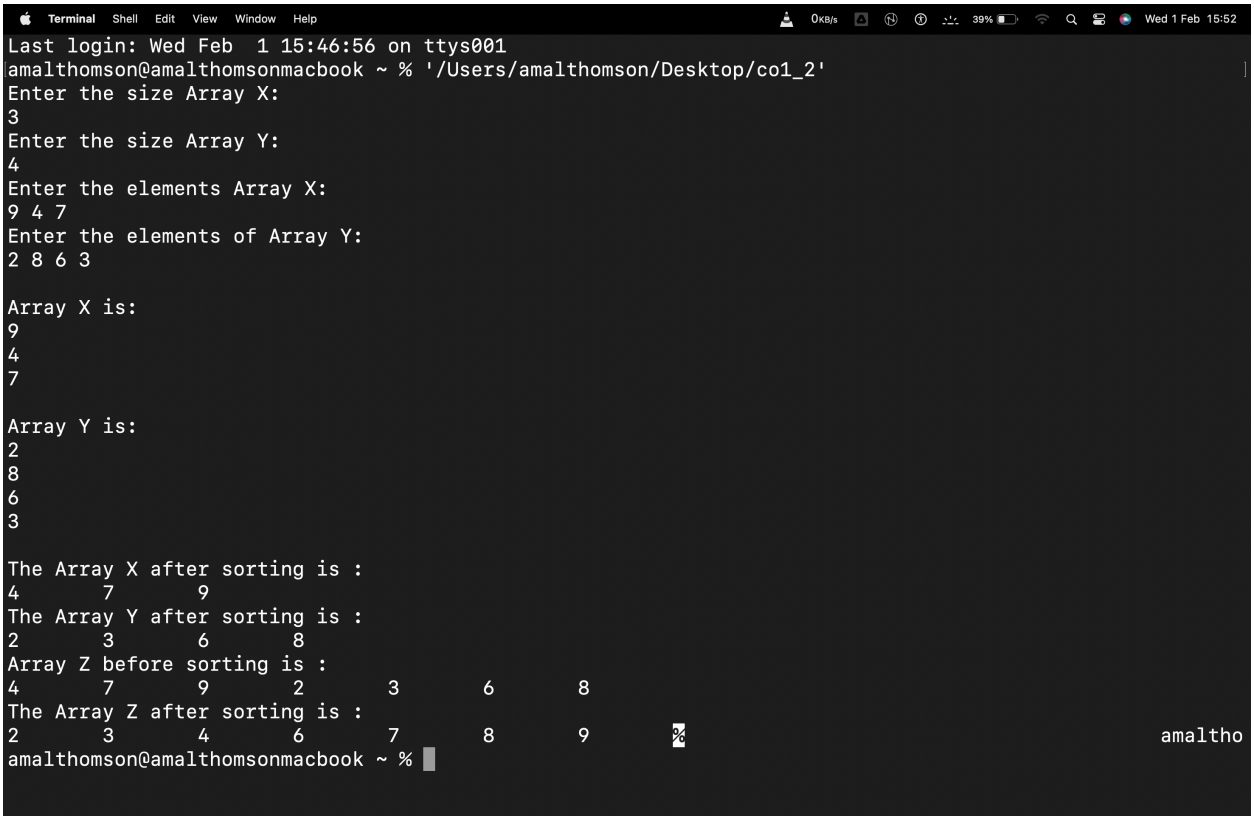
Source Code

```
#include<stdio.h>
void main()
{
int i,j;
int x,y,z;
int a1[10],a2[10],a3[10];
int temp;
printf("Enter the size Array X:\n");
scanf("%d",&x);
printf("Enter the size Array Y:\n");
scanf("%d",&y);
printf("Enter the elements Array X:\n");
for(i=0;i<x;i++)
{
   scanf("%d",&a1[i]);
}
printf("Enter the elements of Array Y:\n");
for(i=0;i<y;i++)
{
   scanf("%d",&a2[i]);
}
printf("\nArray X is:\n");
for(i=0;i<x;i++)
{
printf("%d\n", a1[i]);
}
printf("\nArray Y is:\n");
for(i=0;i<y;i++)
{
printf("%d\n", a2[i]);
}
for(i=0;i<x;i++)
{
```

```
   for(j=i+1;j<x;j++)
   {
    if( a1[i]>a1[j])
    {
      temp=a1[j];
      a1[j]=a1[i];
      a1[i]=temp;
    }
   }
}
printf("\nThe Array X after sorting is :\n");
for(i=0;i<x;i++)
{
   printf("%d\t",a1[i]);
}
for(i=0;i<y;i++)
{
   for(j=i+1;j<y;j++)
   {
    if( a2[i]>a2[j])
    {
      temp=a2[j];
      a2[j]=a2[i];
      a2[i]=temp;
    }
   }
}
printf("\nThe Array Y after sorting is :\n");
for(i=0;i<y;i++)
{
   printf("%d\t",a2[i]);
}
z=x+y;
for(i=0;i<x;i++)
{
   a3[i]=a1[i];
}
for(i=0;i<y;i++)
{
   a3[i+x]=a2[i];
}
printf("\nArray Z before sorting is :\n");
```

```
for(i=0;i<z;i++)
{
   printf("%d\t",a3[i]);
}
for(i=0;i<z;i++)
{
   for(j=i+1;j<z;j++)
   {
    if( a3[i]>a3[j])
    {
      temp=a3[j];
      a3[j]=a3[i];
      a3[i]=temp;
    }
   }
}
printf("\nThe Array Z after sorting is :\n");
for(i=0;i<z;i++)
{
   printf("%d\t",a3[i]);
}
}
```

Output Screenshot



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

Inserting At Beginning

Step 1 Create a newNode with given value.
Step 2 Check whether list is Empty (head == NULL)
Step 3 If it is Empty then, set newNode→next = NULL and head = newNode.
Step 4 If it is Not Empty then, set newNode→next = head and head = newNode.

Inserting At End

Step 1 Create a newNode with given value and newNode → next as NULL.
Step 2 Check whether list is Empty (head == NULL).
Step 3 If it is Empty then, set head = newNode.
Step 4 If it is Not Empty then, define a node pointer temp and initialize with head.
Step 5 Keep moving the temp to its next node until it reaches to the last node in the list (until temp → next is equal to NULL).
Step 6 Set temp → next = newNode.

Inserting At Specific location

Step 1 Create a newNode with given value.
Step 2 Check whether list is Empty (head == NULL)
Step 3 If it is Empty then, set newNode → next = NULL and head = newNode.
Step 4 If it is Not Empty then, define a node pointer temp and initialize with head.
Step 5 Keep moving the temp to its next node until it reaches to the node after which we want to insert the newNode (until temp1 → data is equal to location, here location is the node value after which we want to insert the newNode).
Step 6 Every time check whether temp is reached to last node or not. If it is reached to last node then display 'Given node is not found in the list!!! Insertion not possible!!!' and terminate

the function. Otherwise move the temp to next node.
Step 7 Finally, Set 'newNode → next = temp → next' and 'temp → next = newNode'

Deleting from Beginning

Step 1 Check whether list is Empty (head == NULL)

Step 2 If it is Empty then, display 'List is Empty!!! Deletion is not possible' and terminate the function.

Step 3 If it is Not Empty then, define a Node pointer 'temp' and initialize with head.

Step 4 Check whether list is having only one node (temp → next == NULL)

Step 5 If it is TRUE then set head = NULL and delete temp (Setting Empty list conditions)

Step 6 If it is FALSE then set head = temp → next, and delete temp.

Deleting from End

Step 1 Check whether list is Empty (head == NULL)

Step 2 If it is Empty then, display 'List is Empty!!! Deletion is not possible' and terminate the function.

Step 3 If it is Not Empty then, define two Node pointers 'temp1' and 'temp2' and initialize 'temp1' with head.

Step 4 Check whether list has only one Node (temp1 → next == NULL)

Step 5 If it is TRUE. Then, set head = NULL and delete temp1. And terminate the function. (Setting Empty list condition)

Step 6 If it is FALSE. Then, set 'temp2 = temp1 ' and move temp1 to its next node. Repeat the same until it reaches to the last node in the list. (until temp1 → next == NULL)

Step 7 Finally, Set temp2 → next = NULL and delete temp1.

Deleting a Specific Node

Step 1 Check whether list is Empty (head == NULL)

Step 2 If it is Empty then, display 'List is Empty!!! Deletion is not possible' and terminate the function.

Step 3 If it is Not Empty then, define two Node pointers 'temp1' and 'temp2' and initialize 'temp1' with head.

Step 4 Keep moving the temp1 until it reaches to the exact node to be deleted or to the last node. And every time set 'temp2 = temp1' before moving the 'temp1' to its next node.

Step 5 If it is reached to the last node then display 'Given node not found in the list! Deletion not possible!!!'. And terminate the function.

Step 6 -f it is reached to the exact node which we want to delete, then check whether list is having only one node or not

Step 7 If list has only one node and that is the node to be deleted, then set head = NULL and delete temp1 (free(temp1)).

Step 8 If list contains multiple nodes, then check whether temp1 is the first node in the list (temp1 == head).

Step 9 If temp1 is the first node then move the head to the next node (head = head → next) and delete temp1.

Step 10 If temp1 is not first node then check whether it is last node in the list (temp1 → next == NULL).

Step 11 If temp1 is last node then set temp2 → next = NULL and delete temp1 (free(temp1)).

Step 12 f temp1 is not first node and not last node then set temp2 → next = temp1 → next and delete temp1 (free(temp1)).

Displaying a Single Linked List

Step 1 Check whether list is Empty (head == NULL)
Step 2 If it is Empty then, display 'List is Empty!!!' and terminate the function.
Step 3 If it is Not Empty then, define a Node pointer 'temp' and initialize with head.
Step 4 Keep displaying temp → data with an arrow (--->) until temp reaches to the last node
Step 5 Finally display temp → data with arrow pointing to NULL (temp → data ---> NULL).

Source Code

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
   int data;
   struct node *next;
};
struct node *head;
void beginsert ();
void lastinsert ();
void randominsert();
void begin_delete();
void last_delete();
void random_delete();
void display();
void search();
void main ()
{
   int choice =0;
   while(choice != 9)
   {
      printf("\nMain Menu\n");
      printf("\nChoose Operation\n");
      printf("\n1.Insert in begining\n2.Insert at last\n3.Insert at any random
location\n4.Delete from Beginning\n5.Delete from last\n6.Delete node after specified
location\n7.Search for an element\n8.Show\n9.Exit\n");
      printf("\nEnter your choice?\n");
      scanf("\n%d",&choice);
```

```
        switch(choice)
        {
            case 1:
            beginsert();
            break;
            case 2:
            lastinsert();
            break;
            case 3:
            randominsert();
            break;
            case 4:
            begin_delete();
            break;
            case 5:
            last_delete();
            break;
            case 6:
            random_delete();
            break;
            case 7:
            search();
            break;
            case 8:
            display();
            break;
            case 9:
            exit(0);
            break;
            default:
            printf("INVALID INPUT");
        }
    }
}
void beginsert()
{
    struct node *ptr;
    int item;
    ptr = (struct node *) malloc(sizeof(struct node *));
```

```
    if(ptr == NULL)
    {
        printf("\nOVERFLOW");
    }
    else
    {
        printf("\nEnter value\n");
        scanf("%d",&item);
        ptr->data = item;
        ptr->next = head;
        head = ptr;
        printf("\nINSERTION SUCCESSFUL");
    }
}

void lastinsert()
{
    struct node *ptr,*temp;
    int item;
    ptr = (struct node*)malloc(sizeof(struct node));
    if(ptr == NULL)
    {
        printf("\nOVERFLOW");
    }
    else
    {
        printf("\nEnter value?\n");
        scanf("%d",&item);
        ptr->data = item;
        if(head == NULL)
        {
            ptr -> next = NULL;
            head = ptr;
            printf("\nINSERTION SUCCESSFUL");
        }
        else
        {
            temp = head;
            while (temp -> next != NULL)
```

```
            {
               temp = temp -> next;
            }
            temp->next = ptr;
            ptr->next = NULL;
            printf("\nINSERTION SUCCESSFUL");
         }
      }
}

void randominsert()
{
   int i,loc,item;
   struct node *ptr, *temp;
   ptr = (struct node *) malloc (sizeof(struct node));
   if(ptr == NULL)
   {
      printf("\nOVERFLOW");
   }
   else
   {
      printf("\nEnter element value");
      scanf("%d",&item);
      ptr->data = item;
      printf("\nEnter the location after which you want to insert ");
      scanf("\n%d",&loc);
      temp=head;
      for(i=0;i<loc;i++)
      {
         temp = temp->next;
         if(temp == NULL)
         {
            printf("\nINSERTION FAILED\n");
            return;
         }
      }
      ptr ->next = temp ->next;
      temp ->next = ptr;
      printf("\nINSERTION SUCCESSFUL");
```

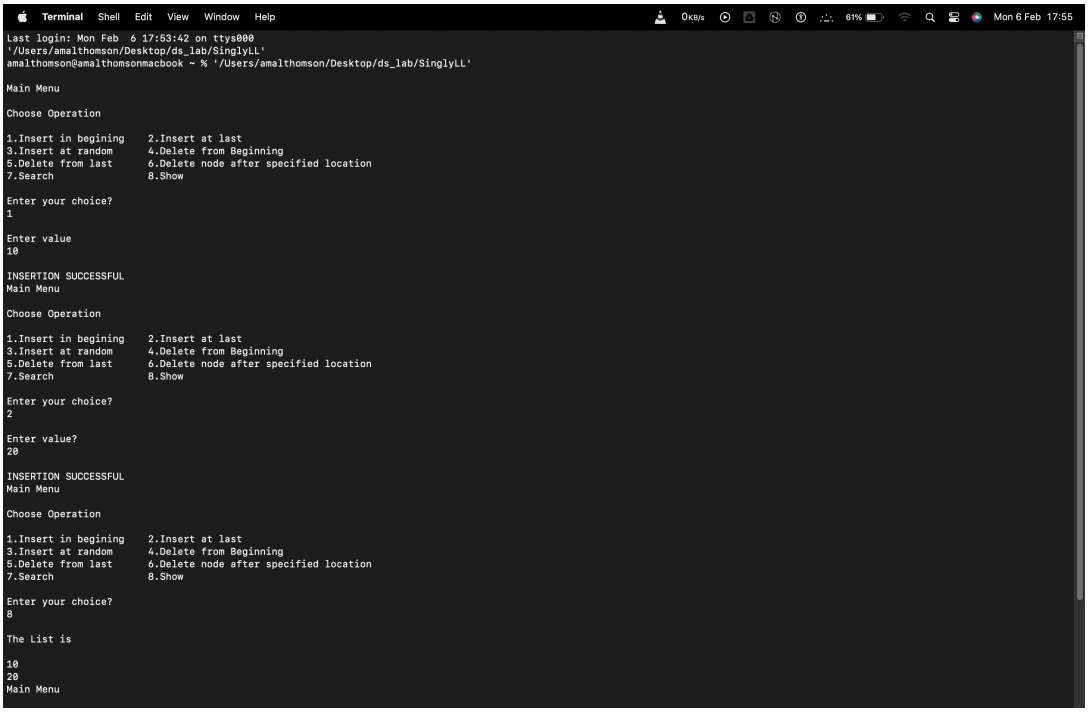
```
    }
}

void begin_delete()
{
    struct node *ptr;
    if(head == NULL)
    {
        printf("\nList is empty\n");
    }
    else
    {
        ptr = head;
        head = ptr->next;
        free(ptr);
        printf("\DELETION SUCCESSFUL\n");
    }
}

void last_delete()
{
    struct node *ptr,*ptr1;
    if(head == NULL)
    {
        printf("\nlist is empty");
    }
    else if(head -> next == NULL)
    {
        head = NULL;
        free(head);
        printf("\DELETION SUCCESSFUL\n");
    }
    else
    {
        ptr = head;
        while(ptr->next != NULL)
        {
            ptr1 = ptr;
            ptr = ptr ->next;
```

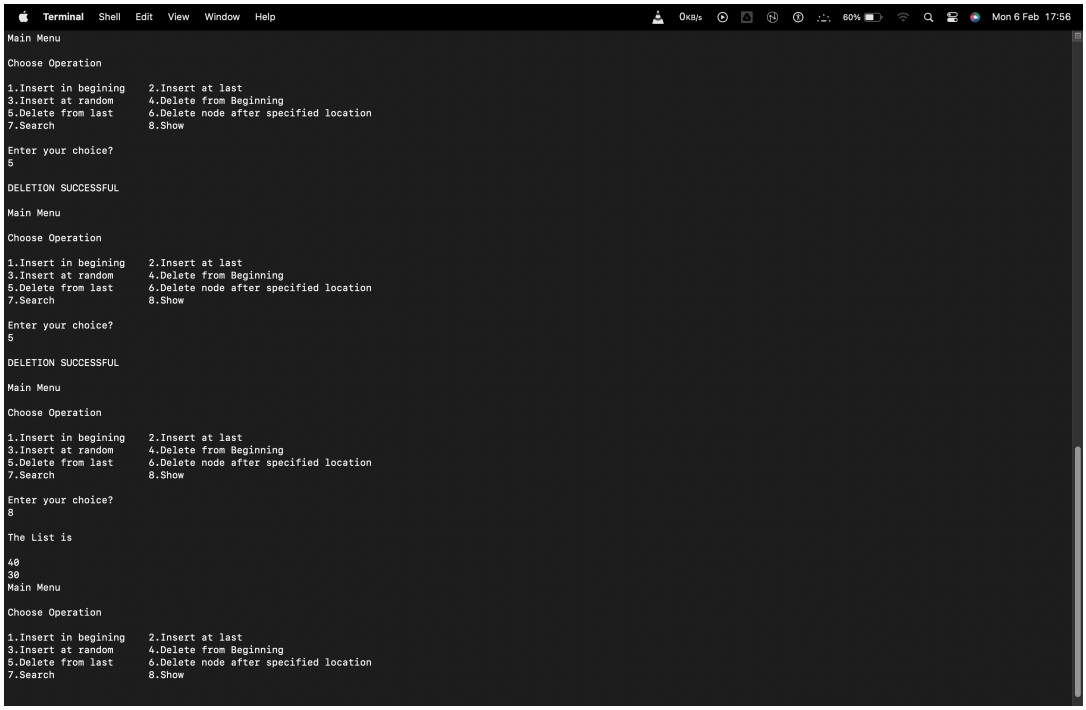
```
        }
        ptr1->next = NULL;
        free(ptr);
        printf("\nDELETION SUCCESSFUL\n");
    }
}

void random_delete()
{
    struct node *ptr,*ptr1;
    int loc,i;
    printf("\n Enter the location of the node after which you want to perform deletion \n");
    scanf("%d",&loc);
    ptr=head;
    for(i=0;i<loc;i++)
    {
        ptr1 = ptr;
        ptr = ptr->next;

        if(ptr == NULL)
        {
            printf("\nDELETION FAILED");
            return;
        }
    }
    ptr1 ->next = ptr ->next;
    free(ptr);
    printf("\nDeleted node %d ",loc+1);
}

void search()
{
    struct node *ptr;
    int item,i=0,flag;
    ptr = head;
    if(ptr == NULL)
    {
        printf("\nEmpty List\n");
    }
```

```
    else
    {
        printf("\nEnter item which you want to search?\n");
        scanf("%d",&item);
        while (ptr!=NULL)
        {
            if(ptr->data == item)
            {
                printf("item found at location %d ",i+1);
                flag=0;
            }
            else
            {
                flag=1;
            }
            i++;
            ptr = ptr -> next;
        }
        if(flag==1)
        {
            printf("Item not found\n");
        }
    }
}
void display()
{
    struct node *ptr;
    ptr = head;
    if(ptr == NULL)
    {
        printf("EMPTY LIST");
    }
    else
    {
        printf("\nThe List is \n");
        while (ptr!=NULL)
        {
            printf("\n%d",ptr->data);
            ptr = ptr -> next;
```

```
        }
    }
}
```

Output Screenshot





Result

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

Experiment No.: 4

Aim

Implementation of Circular Queue.

CO1

Use Basic Data Structures and its operations implementations.

Algorithm

ENQUEUE

```
Step 1  IF (REAR+1)%MAX = FRONT
        Write " OVERFLOW "
        Goto step 4
        [End OF IF]
Step 2  IF FRONT = -1 and REAR = -1
        SET FRONT = REAR = 0
        ELSE IF REAR = MAX - 1 and FRONT ! = 0
        SET REAR = 0
        ELSE
        SET REAR = (REAR + 1) % MAX
        [END OF IF]
Step 3  SET QUEUE[REAR] = VAL
```

DEQUEUE

```
Step 1  IF FRONT = -1
        Write " UNDERFLOW "
        Goto Step 4
        [END of IF]
Step 2  SET VAL = QUEUE[FRONT]
Step 3  IF FRONT = REAR
        SET FRONT = REAR = -1
        ELSE
        IF FRONT = MAX -1
        SET FRONT = 0
        ELSE
        SET FRONT = FRONT + 1
        [END of IF]
        [END OF IF]
Step 4  EXIT
```

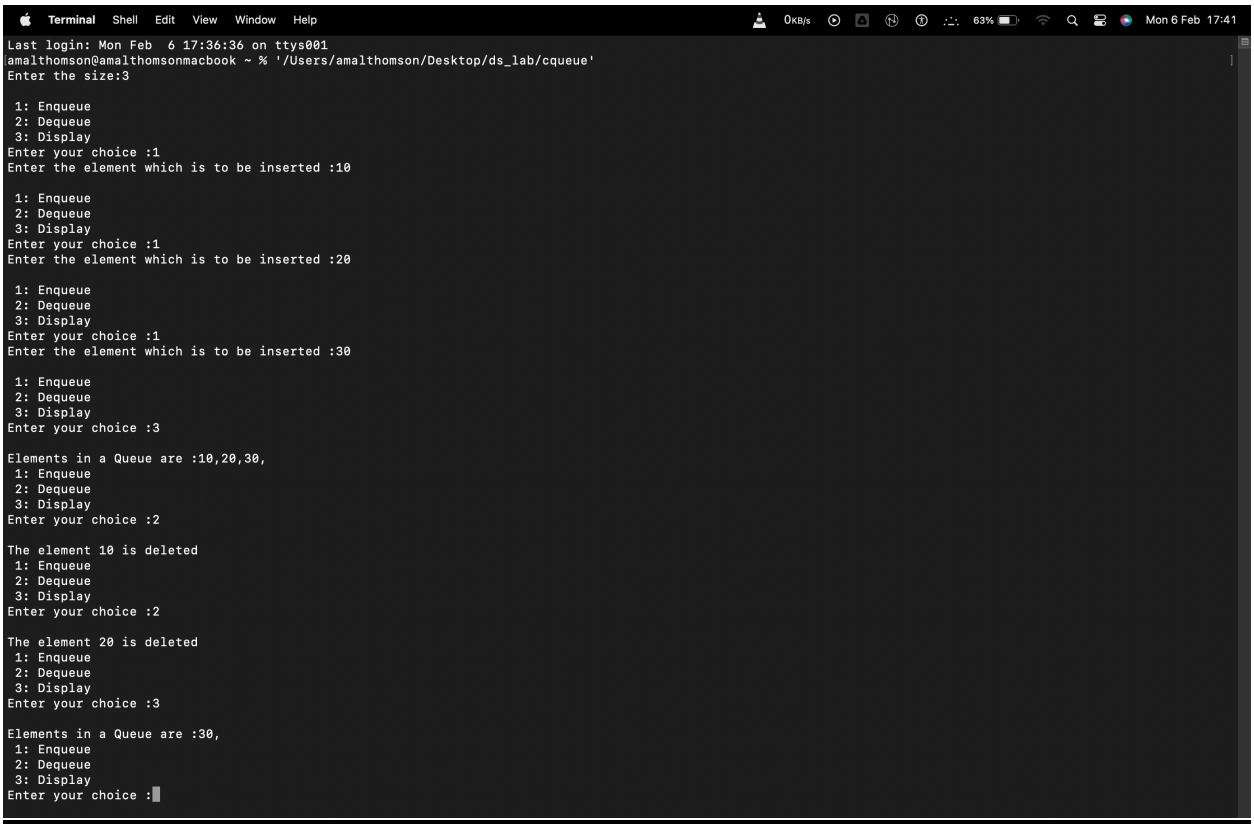
Source Code

```
#include <stdio.h>
int queue[50],max;
int front=-1,rear=-1;
void enqueue(int element,int max)
{
   if(front==-1)
   {
      front=0;
      rear=0;
      queue[rear]=element;
   }
   else if((rear+1)%max==front)
   {
      printf("Queue is full");
   }
   else
   {
      rear=(rear+1)%max;
      queue[rear]=element;
   }
}
int dequeue(int max)
{
   if(front==-1)
   {
      printf("\nQueue is empty.");
   }
 else if(front==rear)
{
  printf("\nThe element %d is deleted", queue[front]);
  front=-1;
  rear=-1;
}
else
{
   printf("\nThe element %d is deleted", queue[front]);
  front=(front+1)%max;
}
}
void display(int max)
```

```
{
   int i=front;
   if(front==-1)
   {
      printf("\n Queue is empty..");
   }
   else
   {
      printf("\nElements in a Queue are :");
      while(i<=rear)
      {
         printf("%d,", queue[i]);
         i++;
      }
   }
}
int main()
{
   int choice=1,x;
   printf("Enter the size:");
   scanf("%d",&max);
   while(choice<4 && choice!=0)
   {
   printf("\n 1: Enqueue");
   printf("\n 2: Dequeue");
   printf("\n 3: Display ");
   printf("\nEnter your choice :");
   scanf("%d", &choice);
   switch(choice)
   {
      case 1:
      printf("Enter the element which is to be inserted :");
      scanf("%d", &x);
      enqueue(x,max);
      break;
      case 2:
      dequeue(max);
      break;
      case 3:
      display(max);

   }
```

```
}
   return 0;
}
```

Output Screenshot



Result

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

Experiment No.: 5

Aim

Implementation of Doubly Linked list.

CO1

Use Basic Data Structures and its operations implementations.

Algorithm

Inserting At Beginning of the list

Step 1 Create a newNode with given value and newNode → previous as NULL.
Step 2 Check whether list is Empty (head == NULL)
Step 3 If it is Empty then, assign NULL to newNode → next and newNode to head.
Step 4 If it is not Empty then, assign head to newNode → next and newNode to head.

Inserting At End of the list

Step 1 Create a newNode with given value and newNode → next as NULL.
Step 2 Check whether list is Empty (head == NULL)
Step 3 If it is Empty, then assign NULL to newNode → previous and newNode to head.
Step 4 If it is not Empty, then, define a node pointer temp and initialize with head.
Step 5 Keep moving the temp to its next node until it reaches to the last node in the list (until temp → next is equal to NULL).
Step 6 Assign newNode to temp → next and temp to newNode → previous.

Inserting At Specific location in the list (After a Node)

Step 1 Create a newNode with given value.
Step 2 Check whether list is Empty (head == NULL)
Step 3 If it is Empty then, assign NULL to both newNode → previous & newNode → next and set newNode to head.
Step 4 If it is not Empty then, define two node pointers temp1 & temp2 and initialize temp1 with head.
Step 5 Keep moving the temp1 to its next node until it reaches to the node after which we want to insert the newNode (until temp1 → data is equal to location, here location is the node value after which we want to insert the newNode).
Step 6 Every time check whether temp1 is reached to the last node. If it is reached to the last node then display 'Given node is not found in the list!!! Insertion not possible!!!' and terminate the function. Otherwise move the temp1 to next node.
Step 7 Assign temp1 → next to temp2, newNode to temp1 → next, temp1 to newNode → previous, temp2 to newNode → next and newNode to temp2 → previous.

Deleting from Beginning of the list

Step 1 Check whether list is Empty (head == NULL)
Step 2 If it is Empty then, display 'List is Empty!!! Deletion is not possible' and terminate the function.
Step 3 If it is not Empty then, define a Node pointer 'temp' and initialize with head.
Step 4 Check whether list is having only one node (temp → previous is equal to temp → next)
Step 5 If it is TRUE, then set head to NULL and delete temp (Setting Empty list conditions)
Step 6 If it is FALSE, then assign temp → next to head, NULL to head → previous and delete temp.

Deleting from End of the list

Step 1 Check whether list is Empty (head == NULL)
Step 2 If it is Empty, then display 'List is Empty!!! Deletion is not possible' and terminate the function.
Step 3 If it is not Empty then, define a Node pointer 'temp' and initialize with head.
Step 4 Check whether list has only one Node (temp → previous and temp → next both are NULL)
Step 5 If it is TRUE, then assign NULL to head and delete temp. And terminate from the function. (Setting Empty list condition)
Step 6 If it is FALSE, then keep moving temp until it reaches to the last node in the list. (until temp → next is equal to NULL)
Step 7 Assign NULL to temp → previous → next and delete temp.

Deleting a Specific Node from the list

Step 1 Check whether list is Empty (head == NULL)
Step 2 If it is Empty then, display 'List is Empty!!! Deletion is not possible' and terminate the function.
Step 3 If it is not Empty, then define a Node pointer 'temp' and initialize with head.
Step 4 Keep moving the temp until it reaches to the exact node to be deleted or to the last node.
Step 5 If it is reached to the last node, then display 'Given node not found in the list! Deletion not possible!!!' and terminate the fuction.
Step 6 If it is reached to the exact node which we want to delete, then check whether list is having only one node or not
Step 7 If list has only one node and that is the node which is to be deleted then set head to NULL and delete temp (free(temp)).
Step 8 If list contains multiple nodes, then check whether temp is the first node in the list (temp == head).
Step 9 If temp is the first node, then move the head to the next node (head = head → next), set head of previous to NULL (head → previous = NULL) and delete temp.
Step 10 If temp is not the first node, then check whether it is the last node in the list (temp → next == NULL).
Step 11 If temp is the last node then set temp of previous of next to NULL (temp → previous → next = NULL) and delete temp (free(temp)).

Step 12If temp is not the first node and not the last node, then set temp of previous of next to temp of next (temp → previous → next = temp → next), temp of next of previous to temp of previous (temp → next → previous = temp → previous) and delete temp (free(temp)).

Displaying a Double Linked List

Step 1 Check whether list is Empty (head == NULL)
Step 2 If it is Empty, then display 'List is Empty!!!' and terminate the function.
Step 3 If it is not Empty, then define a Node pointer 'temp' and initialize with head.
Step 4 Display 'NULL <--- '.
Step 5 Keep displaying temp → data with an arrow (<===>) until temp reaches to the last node
Step 6 Finally, display temp → data with arrow pointing to NULL (temp → data ---> NULL).

Source Code

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
   struct node *prev;
   struct node *next;
   int data;
};
struct node *head;
void insertion_beginning();
void insertion_last();
void insertion_specified();
void deletion_beginning();
void deletion_last();
void deletion_specified();
void display();
void search();
void main ()
{
int choice =0;
   while(choice != 9)
   {
      printf("\nMain Menu\n");
      printf("\nChoose one option from the following list ...\n");
      printf("\n1.Insert in begining\t2.Insert at last\t3.Insert at any random location\n4.Delete
from Beginning\t5.Delete from last\t6.Delete the node after the given
data\n7.Search\t\t8.Show\t\t\t9.Exit\n");
      printf("\nEnter your choice?:\n");
```

```
        scanf("\n%d",&choice);
        switch(choice)
        {
            case 1:
            insertion_beginning();
            break;
            case 2:
            insertion_last();
            break;
            case 3:
            insertion_specified();
            break;
            case 4:
            deletion_beginning();
            break;
            case 5:
            deletion_last();
            break;
            case 6:
            deletion_specified();
            break;
            case 7:
            search();
            break;
            case 8:
            display();
            break;
            case 9:
            exit(0);
            break;
            default:
            printf("Please enter valid choice..");
        }
    }
}
void insertion_beginning()
{
   struct node *ptr;
   int item;
   ptr = (struct node *)malloc(sizeof(struct node));
   if(ptr == NULL)
   {
```

```
      printf("\nOVERFLOW");
    }
    else
    {
     printf("\nEnter Item value");
     scanf("%d",&item);
    if(head==NULL)
    {
       ptr->next = NULL;
       ptr->prev=NULL;
       ptr->data=item;
       head=ptr;
    }
    else
    {
       ptr->data=item;
       ptr->prev=NULL;
       ptr->next = head;
       head->prev=ptr;
       head=ptr;
    }
    printf("\nNode inserted\n");
}
}
void insertion_last()
{
    struct node *ptr,*temp;
    int item;
    ptr = (struct node *) malloc(sizeof(struct node));
    if(ptr == NULL)
    {
       printf("\nOVERFLOW");
    }
    else
    {
       printf("\nEnter value");
       scanf("%d",&item);
        ptr->data=item;
       if(head == NULL)
       {
          ptr->next = NULL;
          ptr->prev = NULL;
```

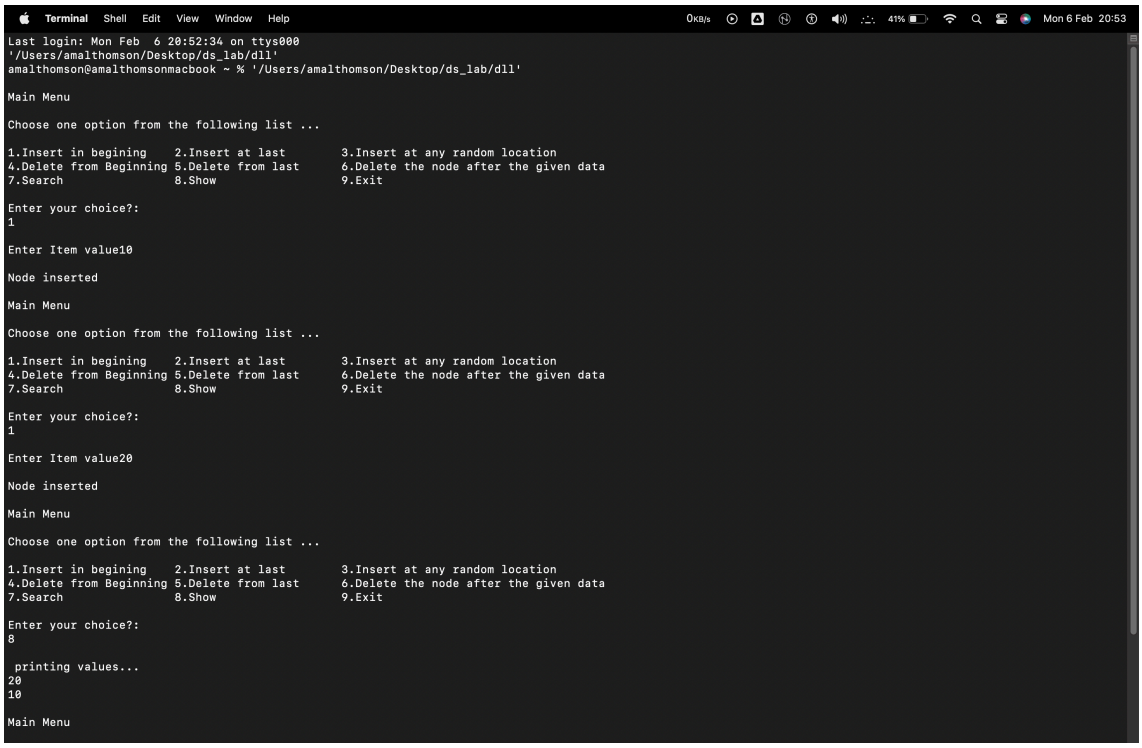
```
        head = ptr;
      }
      else
      {
        temp = head;
        while(temp->next!=NULL)
        {
           temp = temp->next;
        }
        temp->next = ptr;
        ptr ->prev=temp;
        ptr->next = NULL;
        }
      }
    printf("\nnode inserted\n");
   }
void insertion_specified()
{
  struct node *ptr,*temp;
  int item,loc,i;
  ptr = (struct node *)malloc(sizeof(struct node));
  if(ptr == NULL)
  {
     printf("\n OVERFLOW");
  }
  else
  {
     temp=head;
     printf("Enter the location");
     scanf("%d",&loc);
     for(i=0;i<loc;i++)
     {
        temp = temp->next;
        if(temp == NULL)
        {
           printf("\n There are less than %d elements", loc);
           return;
        }
     }
     printf("Enter value");
     scanf("%d",&item);
     ptr->data = item;
```

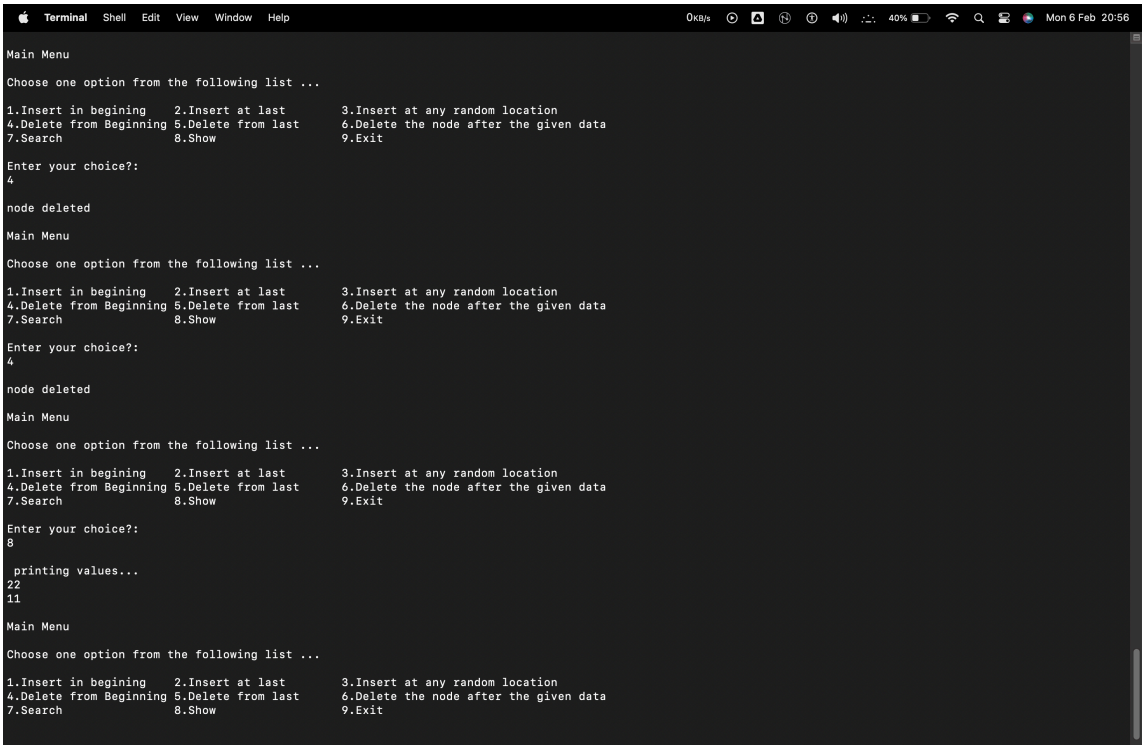
```
      ptr->next = temp->next;
      ptr -> prev = temp;
      temp->next = ptr;
      temp->next->prev=ptr;
      printf("\nnode inserted\n");
   }
}
void deletion_beginning()
{
   struct node *ptr;
   if(head == NULL)
   {
      printf("\n UNDERFLOW");
   }
   else if(head->next == NULL)
   {
      head = NULL;
      free(head);
      printf("\nnode deleted\n");
   }
   else
   {
      ptr = head;
      head = head -> next;
      head -> prev = NULL;
      free(ptr);
      printf("\nnode deleted\n");
   }
}
void deletion_last()
{
   struct node *ptr;
   if(head == NULL)
   {
      printf("\n UNDERFLOW");
   }
   else if(head->next == NULL)
   {
      head = NULL;
      free(head);
      printf("\nnode deleted\n");
   }
```

```
    else
    {
        ptr = head;
        if(ptr->next != NULL)
        {
            ptr = ptr -> next;
        }
        ptr -> prev -> next = NULL;
        free(ptr);
        printf("\nnode deleted\n");
    }
}
void deletion_specified()
{
    struct node *ptr, *temp;
    int val;
    printf("\n Enter the data after which the node is to be deleted : ");
    scanf("%d", &val);
    ptr = head;
    while(ptr -> data != val)
    ptr = ptr -> next;
    if(ptr -> next == NULL)
    {
        printf("\nCan't delete\n");
    }
    else if(ptr -> next -> next == NULL)
    {
        ptr ->next = NULL;
    }
    else
    {
        temp = ptr -> next;
        ptr -> next = temp -> next;
        temp -> next -> prev = ptr;
        free(temp);
        printf("\nnode deleted\n");
    }
}
void display()
{
    struct node *ptr;
    printf("\n printing values...\n");
```

```
    ptr = head;
    while(ptr != NULL)
    {
        printf("%d\n",ptr->data);
        ptr=ptr->next;
    }
}
void search()
{
    struct node *ptr;
    int item,i=0,flag;
    ptr = head;
    if(ptr == NULL)
    {
        printf("\nEmpty List\n");
    }
    else
    {
        printf("\nEnter item which you want to search?\n");
        scanf("%d",&item);
        while (ptr!=NULL)
        {
            if(ptr->data == item)
            {
                printf("\nitem found at location %d ",i+1);
                flag=0;
                break;
            }
            else
            {
                flag=1;
            }
            i++;
            ptr = ptr -> next;
        }
        if(flag==1)
        {
            printf("\nItem not found\n");
        }
    }
}
```

Output Screenshot





Result

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

Experiment No.: 6

Aim

Implementation of 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 operation
   - 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

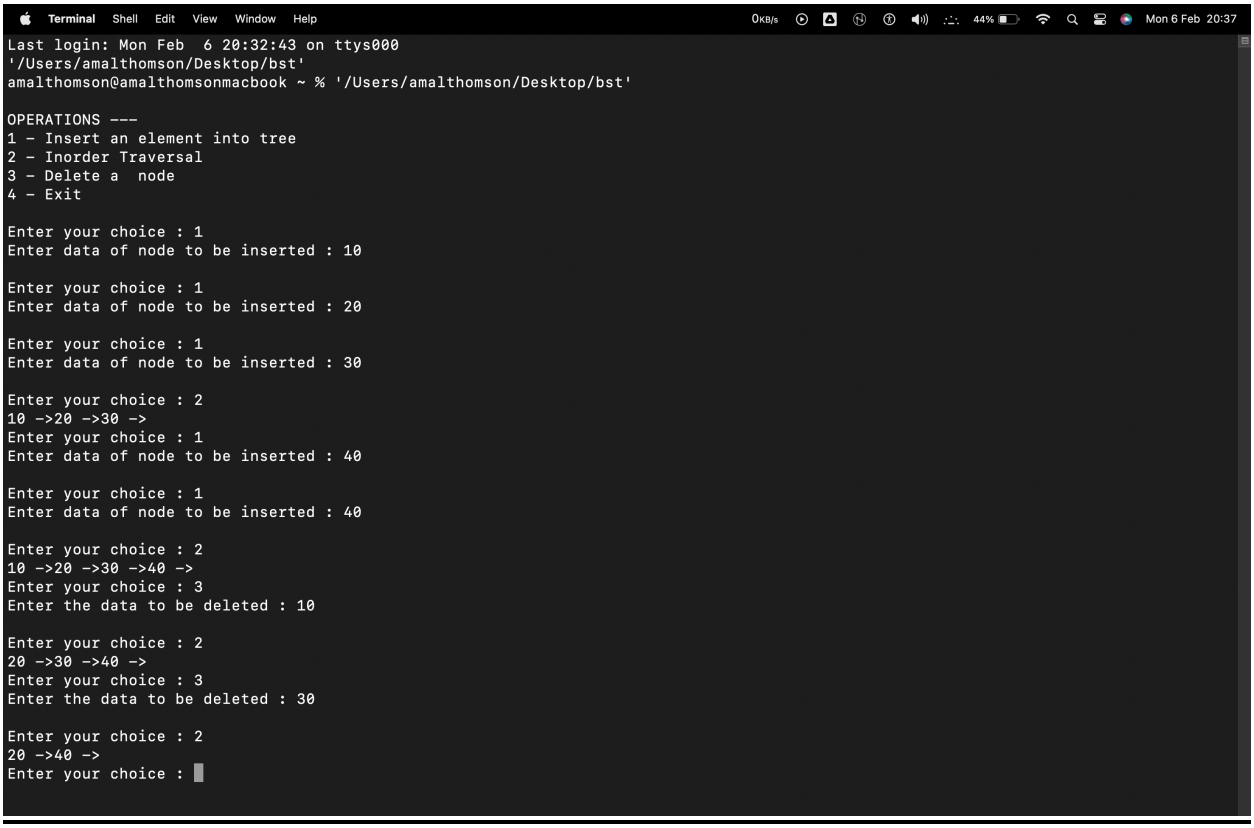
Source Code

```
#include <stdio.h>
#include <stdlib.h>
int a[20],b[20],c[20];
void set_union(int a[],int b[],int m){
printf("after Union operation\n");
for(int i=0;i<m;i++){
   c[i]=a[i]||b[i];
   printf("%d\t",c[i]);
}
return;
}
void set_intersection(int a[],int b[],int m){
printf("after intersection operation\n");
for(int i=0;i<m;i++){
   c[i]=a[i]&&b[i];
   printf("%d\t",c[i]);
}
return;
}
void set_difference(int a[],int b[],int m){
printf("after Difference operation\n");
for(int i=0;i<m;i++){
   c[i]=!b[i]&&a[i];
   printf("%d\t",c[i]);
}
return;
}
void main(){
int m,n,p;
printf("enter the size of 1st set\n");
scanf("%d",&m);
printf("enter the zeros and ones based on condition\n");
for(int i=0;i<m;i++){
   main:
   scanf("%d",&p);
   if (p==0 || p==1){
      a[i]=p;
   }
   else{
      printf("set only accept 0's and 1's please enter a valid number");
```

```
        goto main;
    }
}
printf("enter the size of 2nd set\n");
scanf("%d",&n);
printf("enter the zeros and ones based on condition\n");
for(int i=0;i<n;i++){
    main2:
    scanf("%d",&p);
    if (p==0 || p==1){
        b[i]=p;
    }
    else{
        printf("set only accept 0's and 1's please enter a valid number");
        goto main2;
    }
}
while(1){
    int x;
    printf("\n----------------------SET MENU-----------------------\n");
    printf("1. Union\n 2. Intersection\n 3. Difference\n 0. exit\n enter the option below\n");
    scanf("%d",&x);
    switch(x){
    case 1: if(m==n)
                set_union(a,b,m);
            else
                printf("union perform only same size of array\n");
                exit(1);
            break;
    case 2: if(m==n)
                set_intersection(a,b,m);
            else
                printf("intersection perform only same size of array\n");
                exit(1);
            break;
    case 3: if(m==n)
                set_difference(a,b,m);
            else
                printf("difference perform only same size of array\n");
                exit(1);
            break;
    case 0:exit(1);
```

```
   default: printf("invalid option\n");
   }
}
}
```

Output Screenshot



Result

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

Experiment No.: 7

Aim

Implementation of Set data structure and Set operations.

CO2

Implement the Set and Disjoint Set Data Structures.

Algorithm

1. Start
2. Create two character array
3. Enter a bit string in array 1
4. Enter another bit string in array2
5. Display menu of operations
6. If union():
    a. For i =0 to strlen(array): print(aray1[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] or array3[i])
9. Stop

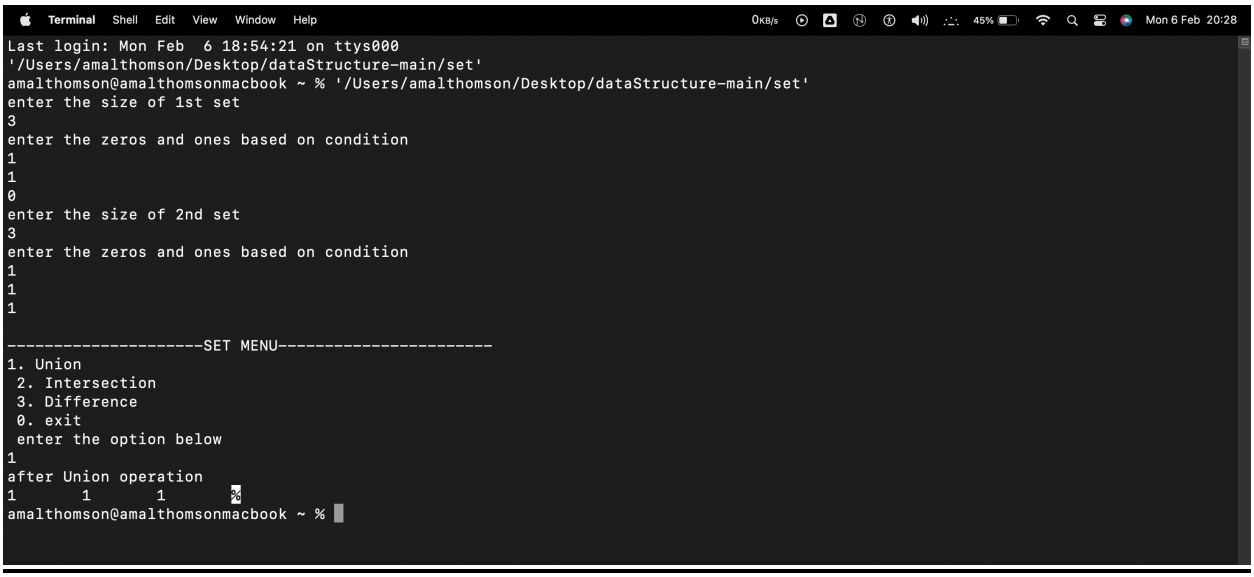
Source Code

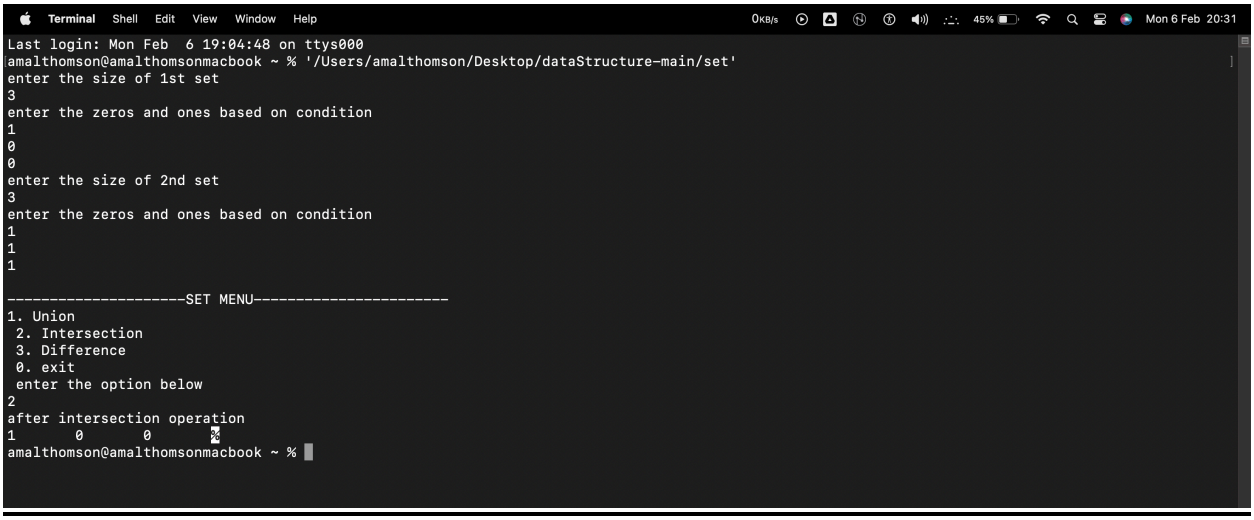
```
#include <stdio.h>
#include <stdlib.h>
int a[20],b[20],c[20];
void set_union(int a[],int b[],int m){
printf("after Union operation\n");
for(int i=0;i<m;i++){
   c[i]=a[i]||b[i];
   printf("%d\t",c[i]);
}
return;
}
void set_intersection(int a[],int b[],int m){
```

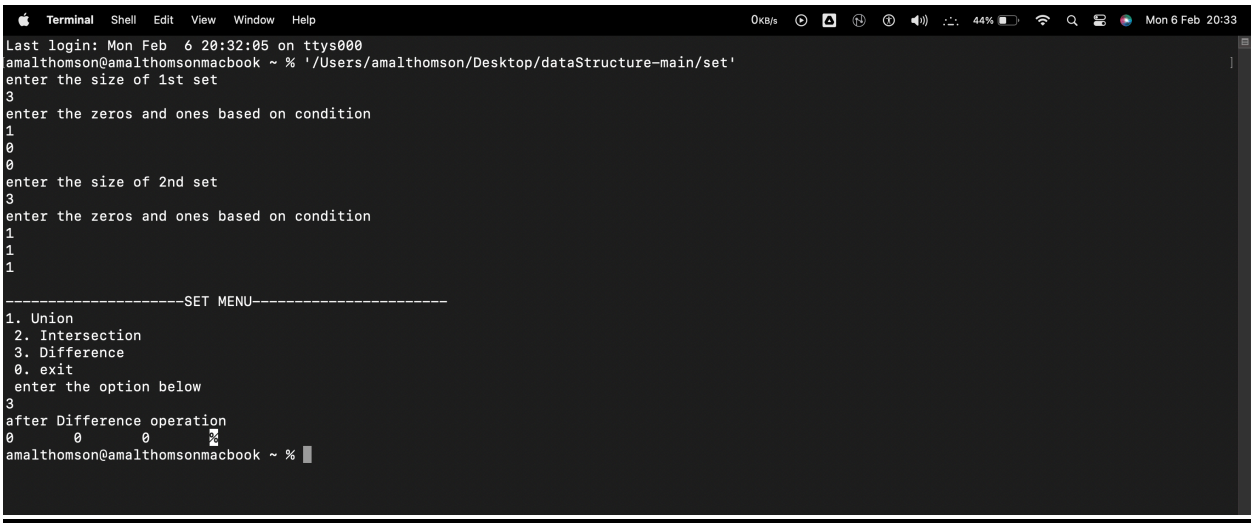
```
printf("after intersection operation\n");
for(int i=0;i<m;i++){
   c[i]=a[i]&&b[i];
   printf("%d\t",c[i]);
}
return;
}
void set_difference(int a[],int b[],int m){
printf("after Difference operation\n");
for(int i=0;i<m;i++){
   c[i]=!b[i]&&a[i];
   printf("%d\t",c[i]);
}
return;
}
void main(){
int m,n,p;
printf("enter the size of 1st set\n");
scanf("%d",&m);
printf("enter the zeros and ones based on condition\n");
for(int i=0;i<m;i++){
   main:
   scanf("%d",&p);
   if (p==0 || p==1){
      a[i]=p;
   }
   else{
      printf("set only accept 0's and 1's please enter a valid number");
      goto main;
   }
}
printf("enter the size of 2nd set\n");
scanf("%d",&n);
printf("enter the zeros and ones based on condition\n");
for(int i=0;i<n;i++){
   main2:
   scanf("%d",&p);
   if (p==0 || p==1){
      b[i]=p;
   }
   else{
      printf("set only accept 0's and 1's please enter a valid number");
```

```
      goto main2;
   }
}
while(1){
   int x;
   printf("\n----------------------SET MENU-----------------------\n");
   printf("1. Union\n 2. Intersection\n 3. Difference\n 0. exit\n enter the option below\n");
   scanf("%d",&x);
   switch(x){
   case 1: if(m==n)
              set_union(a,b,m);
           else
              printf("union perform only same size of array\n");
              exit(1);
           break;
   case 2: if(m==n)
              set_intersection(a,b,m);
           else
              printf("intersection perform only same size of array\n");
              exit(1);
           break;
   case 3: if(m==n)
              set_difference(a,b,m);
           else
              printf("difference perform only same size of array\n");
              exit(1);
           break;
   case 0:exit(1);
   default:
           printf("invalid option\n");
   }
}
}
```

Output Screenshot







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

Step 1. create a new binomial heap H1 of one node of value x
Step 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 - 0 (logn)

Union

Step 1 Merge H1 and H2, i.e. link the roots of H1 and H2 in non-decreasing order.
Step 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.

Source Code

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
struct node
{
   int key;
   int degree;
```

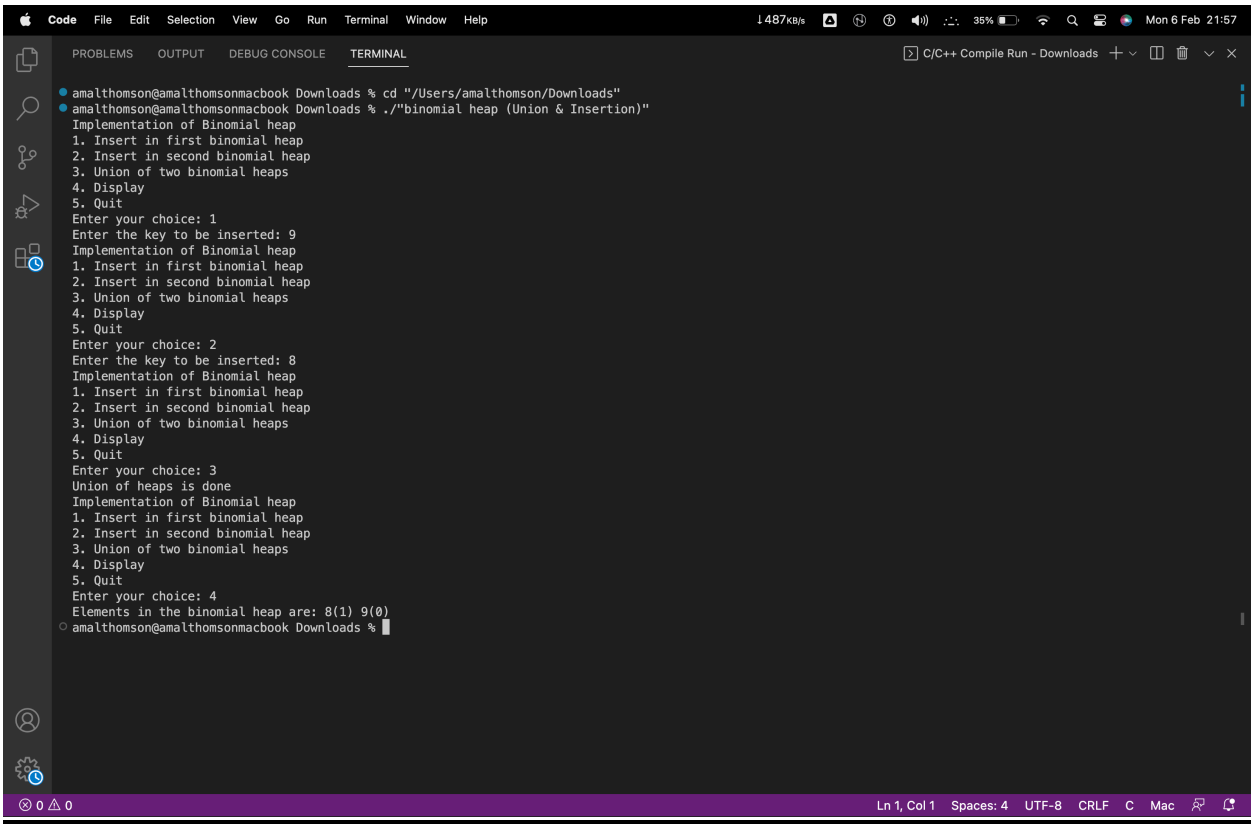
```
   struct node *sibling;
   struct node *child;
};
struct node *newNode(int key)
{
   struct node *temp = (struct node *)malloc(sizeof(struct node));
   temp->key = key;
   temp->degree = 0;
   temp->sibling = temp->child = NULL;
   return temp;
}
struct node *mergeBinomialTrees(struct node *a, struct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



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

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

Source Code

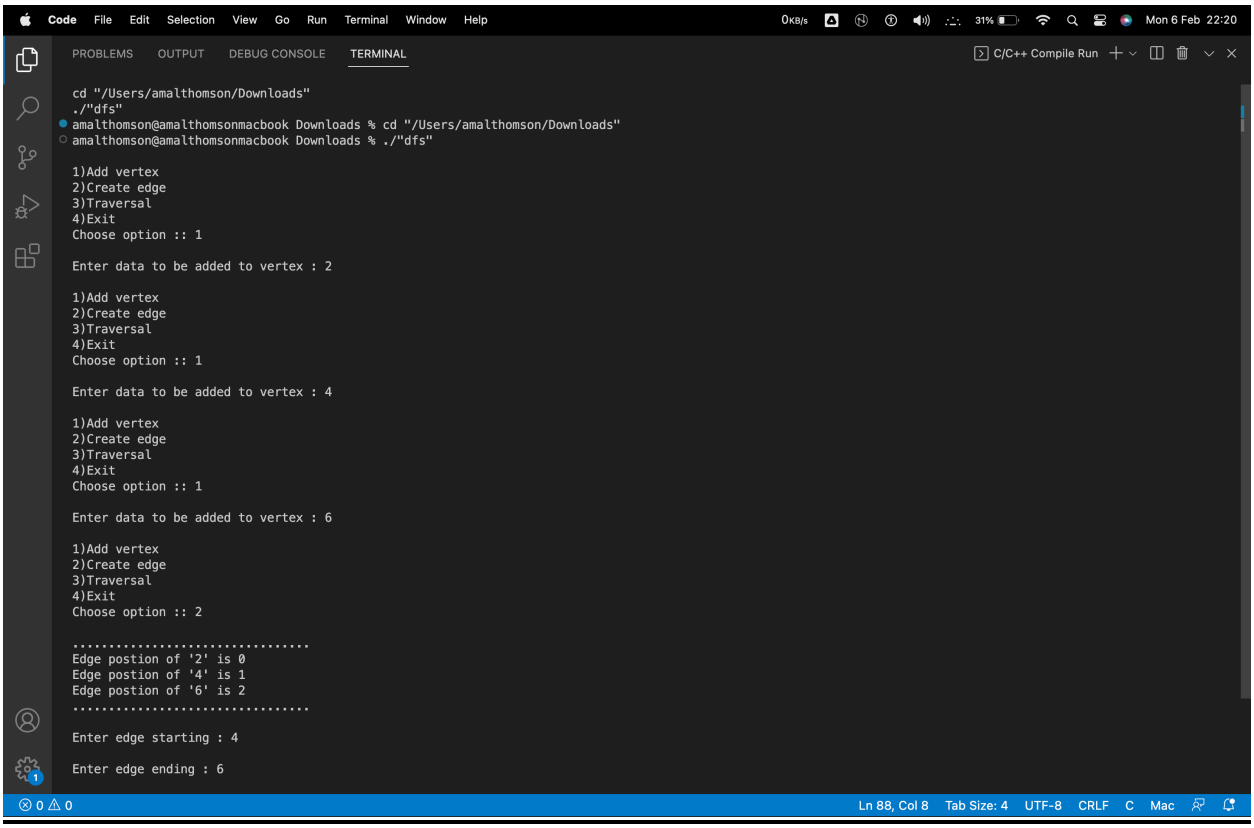
```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#define MAX 6
int vertex_count =0;
struct vertex{
        char data;
        bool visited;
};
struct vertex *graph[MAX];
int adj_matrix[MAX][MAX];
int stack[MAX];
int top = -1;
void push(int data){
        stack[++top]=data;
```

```
}
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 postion 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\n");
                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:
                        dfs();
                case 4:
                        exit(0);
                        break;
                default:
                        printf("\nInvalid option try again !! ...");
            }
        }while(opt!=0);
        return 0;
}
```

Output Screenshot



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

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

Source Code

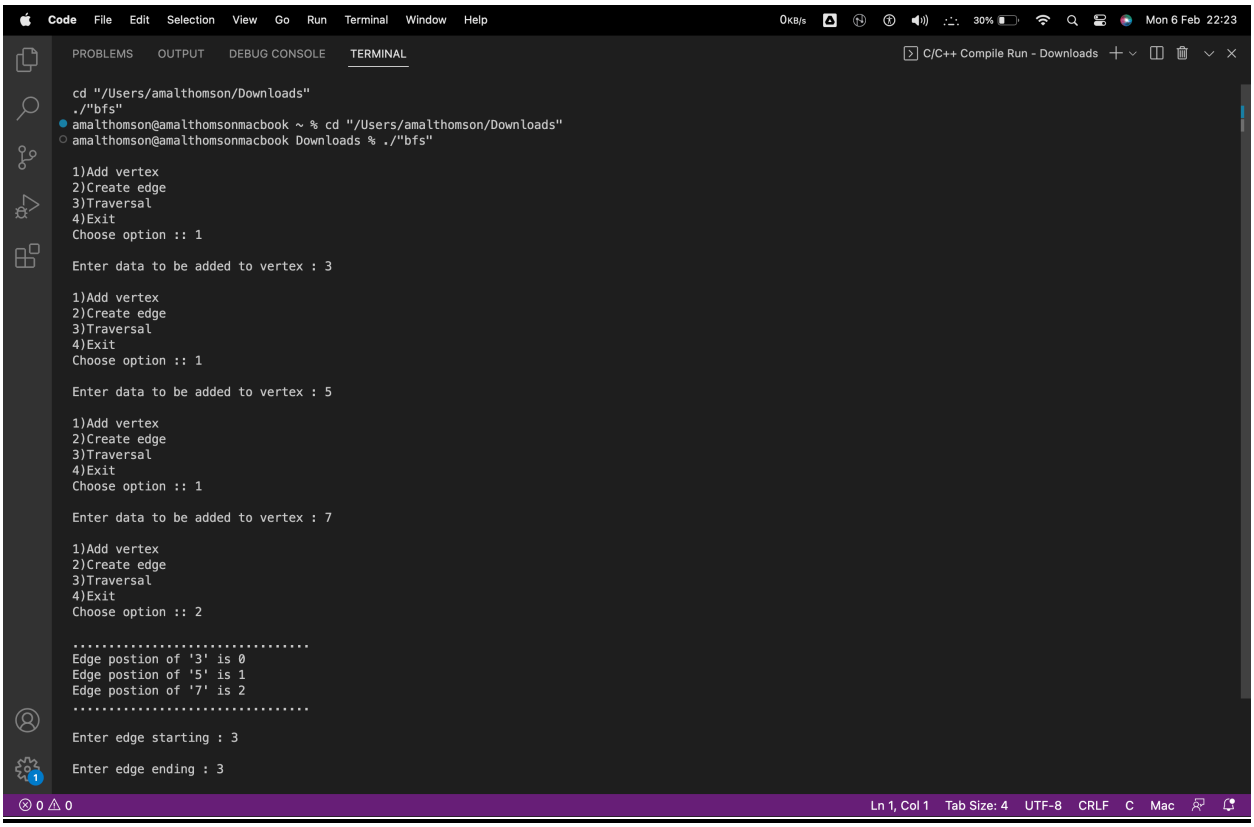
```
#include<stdio.h>
#include<stdlib.h>
#include<stdbool.h>
#define MAX 10
int vertex_count =0;
struct vertex{
        char data;
        bool visited;
};
struct vertex *graph[MAX];
int adj_matrix[MAX][MAX];
int queue[MAX];
int rear=-1;
int front=0;
int queue_count=0;
void enqueue(int data){
        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 postion 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



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 uϵV[G]
2. Do key[u]<-∞
3. Π[u]<-NIL
4. Key[Π]<- 0
5. Q<-V[G]
6. While Q≠Ø

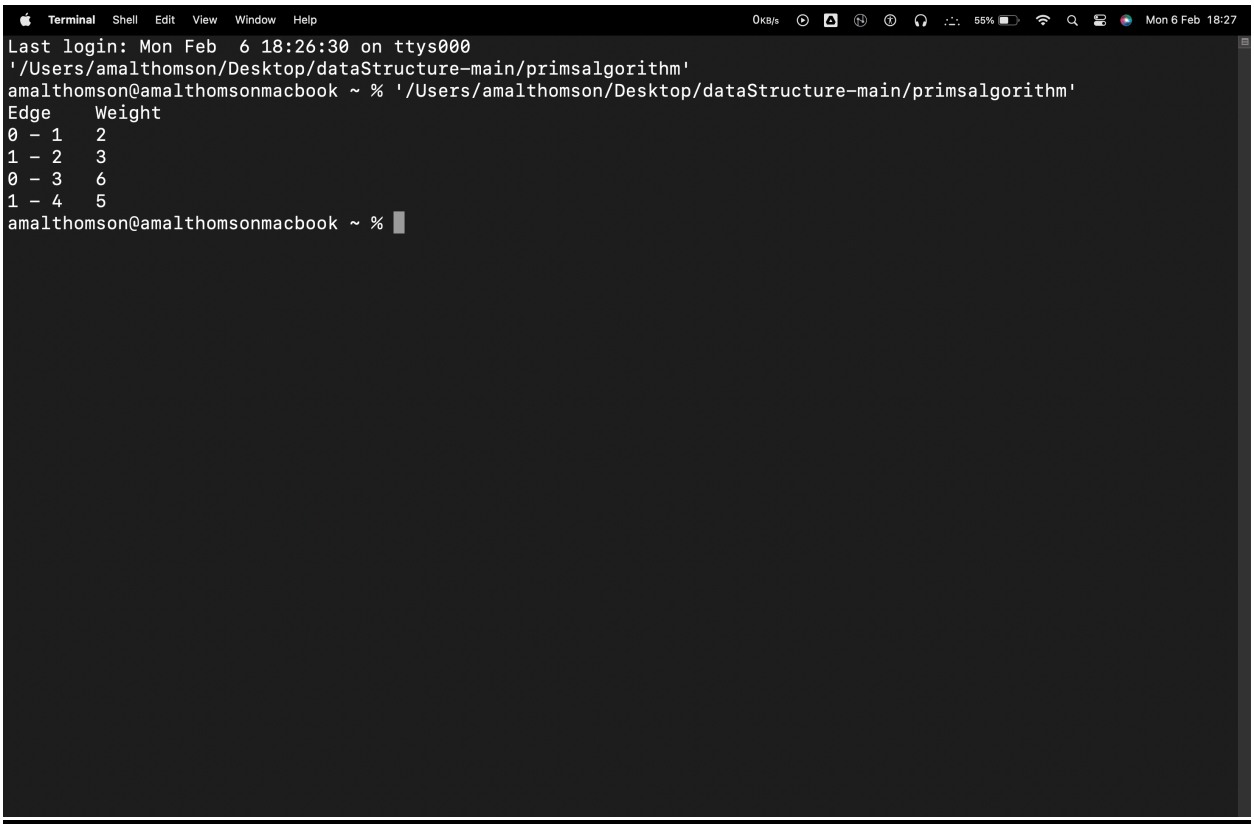
Do u<-extract min(Q)
For each VϵAdj[u]
Do if VϵQ and w[u,v]<key[V]
Then Π[V]<-u
Key[V]<-w[u,v]

Source Code

```
#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



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

KRUSKAL(G):
A = ∅
For each vertex v ∈ G.V:
MAKE-SET(v)
For each edge (u, v) ∈ G.E ordered by increasing order by weight(u, v):
if FIND-SET(u) ≠ FIND-SET(v):
A = A ∪ {(u, v)}
UNION(u, v)
return A

Source Code

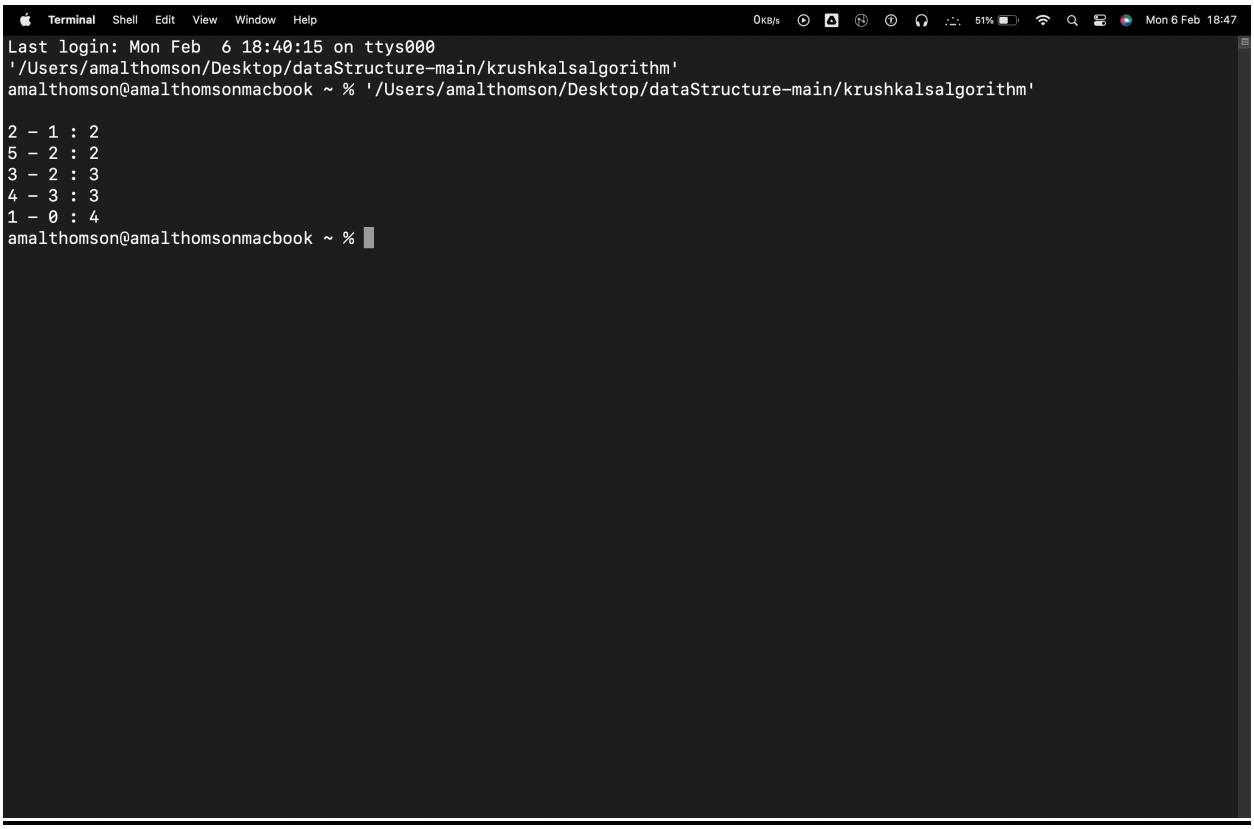
```
#include <stdio.h>
#define MAX 30
typedef struct edge {
  int u, v, w;
} edge;
typedef struct edge_list {
  edge data[MAX];
  int n;
} edge_list;
edge_list elist;
int Graph[MAX][MAX], n;
edge_list spanlist;
void kruskalAlgo();
int find(int belongs[], int vertexno);
void applyUnion(int belongs[], 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



Result

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