

Experiment No.: 1

Aim:

Familiarization with GDB Advanced use of GCC: Important options -o, -c, -D, -l, -I, -g, -O, -save, -temp, -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)

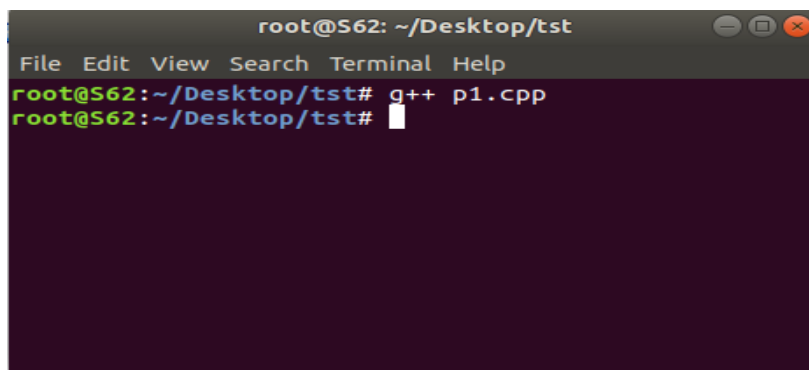
```
$ sudo g++ -o p1 p1.cpp
```

1. \$ sudo g++ p1.cpp

To compile your c++ code, use:

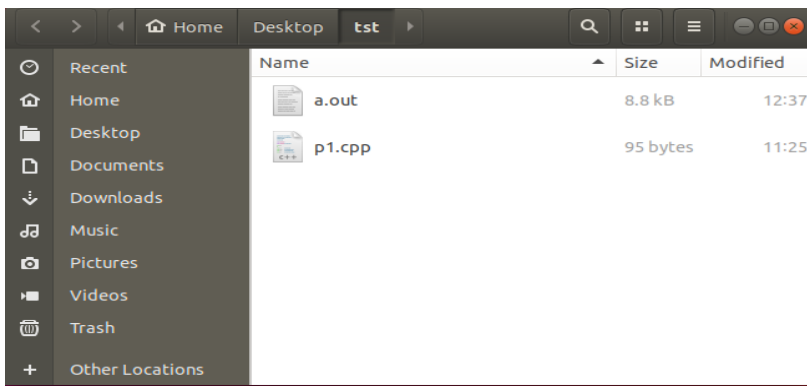
```
g++ p1.cpp
```

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

A screenshot of a terminal window titled 'root@S62: ~/Desktop/tst'. The window has a menu bar with 'File', 'Edit', 'View', 'Search', 'Terminal', and 'Help'. The terminal shows the command 'g++ p1.cpp' being executed, with the prompt changing from 'root@S62:~/Desktop/tst#' to 'root@S62:~/Desktop/tst#'. The background is dark purple.

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

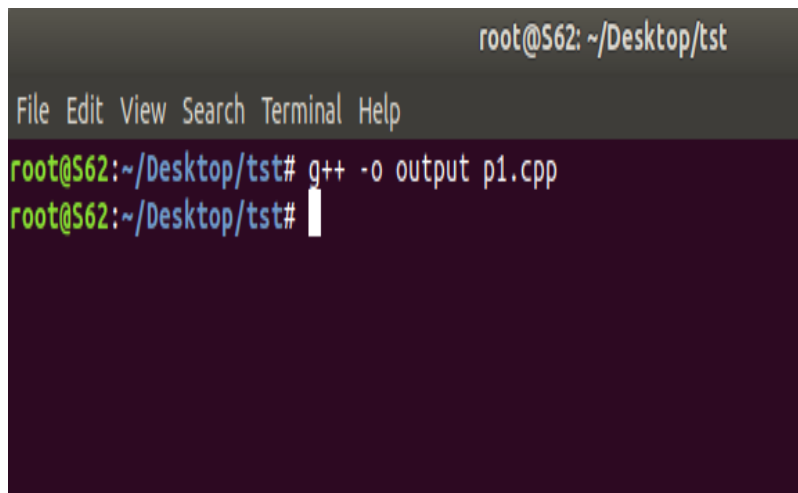
```
./a.out
```



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

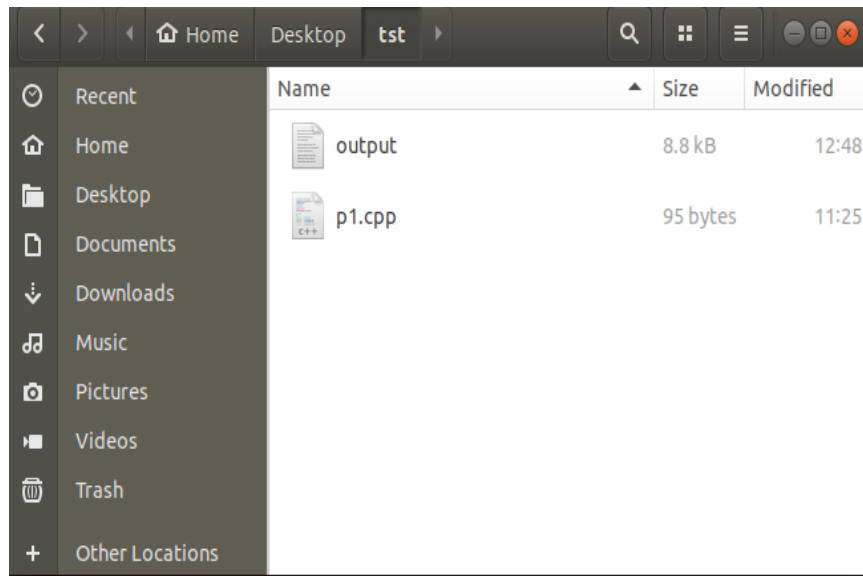
To specify the name of the compiled output file, so that it is not named a.out, use -o with your g++ command.

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

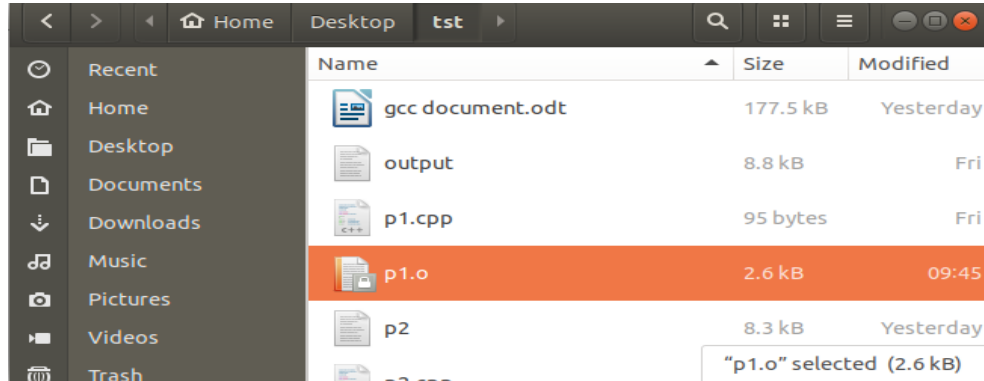


GCC: Important Options

→ **-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 memsimmemsim.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

```
gdbexecutable
```

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<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 -pgtest_gprof.ctest_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.ctest_gprof_new.c
```

```
$ ./test_gprof
```

```
Inside main()
```

```
Inside func1
```

```
Inside new_func1()
```

```
Inside func2
```

```
$ ls
```

```
gmon.out test_gprof test_gprof.ctest_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.ctest_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_gprofmon.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_gprofmon.out> analysis.txt
```

3. Print only flat profile using -p

In case only flat profile is required then :

```
$ gprof -p -b test_gprofmon.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_gprofmon.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_gprofmon.out> analysis.txt
```

6. Print only call graph information using -q

```
gprof -q -b test_gprofmon.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_gprofmon.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_gprofmon.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. Read two arrays of size M and N respectively
3. Print the values of each of the arrays
4. Merge array 1 and array 2 into a third array array3
5. Using bubble sort or selection sort or exchange sort , sort the array3
 - 5.1. For i=0 to i=n-1
 - 5.1.1. For j=0 to j<n-i
 - 5.1.2..if (array3[j]>array3[j+1])
 - 5.1.2.1. TEMP = array3[i+1];
 - 5.1.2.2. array3[j+1]=array3[j];
 - 5.1.2.3. array3[j]= TEMP;
6. Display the final sorted merged array
7. STOP

Procedure:

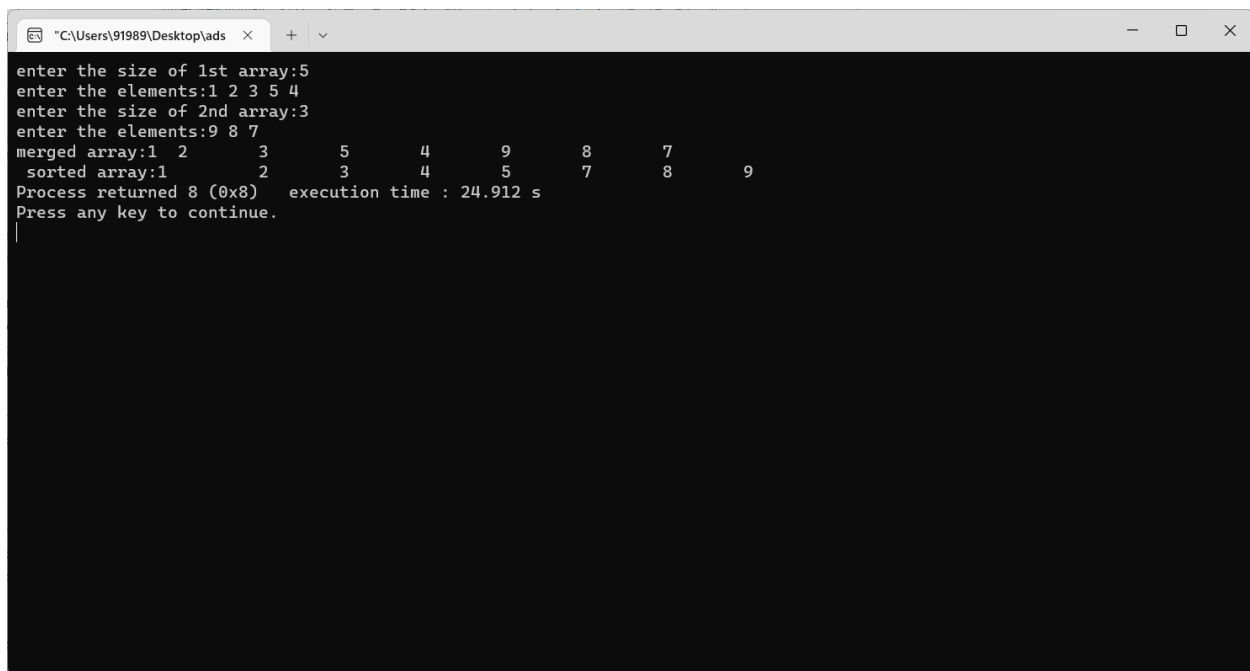
```
#include<stdio.h>

void main()
{
    int arr1[100],arr2[100],arr3[100];
    int n1,n2,n3;
    int i,j,a;
    printf("enter the size of 1st array:");
```

```
scanf("%d",&n1);
printf("enter the elements:");
for(i=0;i<n1;i++)
{
    scanf("%d",&arr1[i]);
}
printf("enter the size of 2nd array:");
scanf("%d",&n2);
printf("enter the elements:");
for(i=0;i<n2;i++)
{
    scanf("%d",&arr2[i]);
}
printf("merged array:");
n3=n1+n2;
for(i=0;i<n1;i++)
{
    arr3[i]=arr1[i];
}
for(i=0;i<n2;i++)
{
    arr3[i+n1]=arr2[i];
}
for(i=0;i<n3;i++)
{
    printf("%d\t",arr3[i]);
}
printf("\n sorted array:");
for(i=0;i<n3;i++)
{
    for(j=i;j<n3;j++)
```

```
{  
    if(arr3[i]>arr3[j])  
    {  
        a=arr3[i];  
        arr3[i]=arr3[j];  
        arr3[j]=a;  
    }  
}  
printf("%d \t",arr3[i]);  
}  
}
```

Output Screenshot



```
"C:\Users\91989\Desktop\ads" X + v  
enter the size of 1st array:5  
enter the elements:1 2 3 5 4  
enter the size of 2nd array:3  
enter the elements:9 8 7  
merged array:1 2 3 5 4 9 8 7 9  
sorted array:1 2 3 4 5 7 8 9  
Process returned 8 (0x8) execution time : 24.912 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:

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 If 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).

Procedure:

```
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>
void insertAtBeginning(int);
void insertAtEnd(int);
void between(int,int);
void deleteFromBeg(int);
void deleteFromEnd(int);
void deleteSpecificLocation(int);
void display();
struct Node
{
    int data;
    struct Node*next;
}*head = NULL,*temp;
void main()
{
    int value,choice,loc1;
    while(1)
    {
        printf("***menu***\n");
        printf("Insert\n1.At beginning\n2.At end\n3.Between node\nDELETE \n4.At beginning\n5.At end\n6.Between node\n7.display\n");
        printf("enter your choice:");
        scanf("%d",&choice);
        switch(choice)
        {
            case 1: printf("Enter the value to be insert: ");
                    scanf("%d",&value);
                    insertAtBeginning(value);
```

```
        break;
    case 2:printf("enter a value you want to insert:");
        scanf("%d",&value);
        insertAtEnd(value);
        break;
    case 3:
        printf("Enter the location: ");
        scanf("%d",&loc1);
        printf("Enter the value: ");
        scanf("%d",&value);
        between(value,loc1);
        break;
    case 4:printf("Enter the value: ");
        scanf("%d",&value);
        deleteFromBeg(value);
        break;
    case 5:printf("Enter the value: ");
        scanf("%d",&value);
        deleteFromEnd(value);
        break;
    case 6:
        printf("Enter the value: ");
        scanf("%d",&value);
        deleteSpecificLocation(value);
        break;
    case 7:display();
        break;
        default: printf("\nWrong selection!!! Please try again!!!\n");
    }
}
}
}
void insertAtBeginning(int value)
{
    struct Node*newNode=malloc(sizeof(struct Node));
    newNode->data=value;
    if(head==NULL)
    {
        newNode->next=NULL;
        head=newNode;
    }
    else
    {
        newNode->next=head;
        head=newNode;
    }
}
void insertAtEnd(int value)
{
```

```
    struct Node *newNode=malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = NULL;
    if(head == NULL)
        head = newNode;
    else
    {
        struct Node *temp = head;
        while(temp->next != NULL)
        { temp = temp->next; }
        temp->next = newNode;
    }
}
}
void between(int value,int loc1)
{
    struct Node*newnode;
    newnode=(struct Node*)malloc(sizeof(struct Node));
    newnode->data=value;
    if(head==NULL)
    {
        printf("The list is empty");
        exit(0);
    }

    else
    {
        temp=head;
        while(temp->data!=loc1)
        {
            if(temp->next==NULL)
            {
                printf("Location not found\n");
                exit(0);
            }
            else
                temp=temp->next;
        }
        newnode->next=temp->next;
        temp->next=newnode;
    }

}

}
void deleteFromBeg(int value)
{
    struct Node *temp;
    if(head == NULL)
    {
```

```
        printf("List is already empty");
    }
    else
    {
        temp=head;
        head=head->next;
        free(temp);
    }
}
void deleteFromEnd(int value)
{
    struct Node*prev;
    temp=head;
    while(temp->next!=NULL)
    {
        prev=temp;
        temp=temp->next;
    }
    if(temp==head)
    {
        head=NULL;
        free(temp);
    }
    else
    {
        prev->next= NULL;
    }
    free(temp);
}
void deleteSpecificLocation(int value)
{
    struct Node*prev;
    temp=head;
    while(temp->next!=NULL)
    {
        prev=temp;
        temp=temp->next;
    }
    if(head==NULL)
    {
        printf("There is no element to delete");
    }
    temp=head;
    if((temp->next!=NULL)&&(temp->data==value))
    {
        printf("The node %d deleted %d and now list is empty",temp->data);
        temp=temp->next;
        free(temp);
    }
}
```

```

    }
    while((temp!=NULL)&&(temp->data!=value))
    {
        prev=temp;
        temp=temp->next;
    }
    if(temp==NULL)
    {
        printf("Element not Found");
    }
    else
    {
        prev->next=temp->next;
        free(temp);
    }
}
void display()
{
    if(head == NULL)
        printf("\nList is Empty!!!\n");
    else
    {
        temp= head;
        while(temp->next!=NULL)
        {
            printf("%d-->",temp->data);
            temp=temp->next;
        }
        printf("%d-->NULL\n",temp->data);
    }
}

```

Output Screenshot

```

***menu***
Insert
1.At beginning
2.At end
3.Between node
DELETE
4.At beginning
5.At end
6.Between node
7.display
enter your choice:1
Enter the value to be insert: 2
***menu***
Insert
1.At beginning
2.At end
3.Between node
DELETE
4.At beginning
5.At end
6.Between node
7.display
enter your choice:1
Enter the value to be insert: 9
***menu***
Insert
1.At beginning
2.At end
3.Between node
DELETE
4.At beginning
5.At end
6.Between node
7.display
enter your choice:2
enter a value you want to insert:6
***menu***
Insert
1.At beginning
2.At end

```

```
"C:\Users\91989\Desktop\lads  x  +  v
5.At end
6.Between node
7.display
enter your choice:2
enter a value you want to insert:7
***menu***
Insert
1.At beginning
2.At end
3.Between node
DELETE
4.At beginning
5.At end
6.Between node
7.display
enter your choice:3
Enter the location: 6
Enter the value: 4
***menu***
Insert
1.At beginning
2.At end
3.Between node
DELETE
4.At beginning
5.At end
6.Between node
7.display
enter your choice:7
9-->2-->6-->4-->7-->NULL
***menu***
Insert
1.At beginning
2.At end
3.Between node
DELETE
4.At beginning
5.At end
6.Between node
7.display
```

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:**Implementation of Circular Queue**

Step 1 - Include all the header files which are used in the program and define a constant 'SIZE' with specific value.

Step 2 - Declare all user defined functions used in circular queue implementation.

Step 3 - Create a one dimensional array with above defined SIZE (int cQueue[SIZE])

Step 4 - Define two integer variables 'front' and 'rear' and initialize both with '-1'. (int front = -1, rear = -1)

Step 5 - Implement main method by displaying menu of operations list and make suitable function calls to perform operation selected by the user on circular queue.

enQueue(value) - Inserting value into the Circular Queue

Step 1 - Check whether queue is FULL. ((rear == SIZE-1 && front == 0) || (front == rear+1))

Step 2 - If it is FULL, then display "Queue is FULL!!! Insertion is not possible!!!" and terminate the function.

Step 3 - If it is NOT FULL, then check rear == SIZE - 1 && front != 0 if it is TRUE, then set rear = -1.

Step 4 - Increment rear value by one (rear++), set queue[rear] = value and check 'front == -1' if it is TRUE, then set front = 0.

deQueue() - Deleting a value from the Circular Queue

Step 1 - Check whether queue is EMPTY. (front == -1 && rear == -1)

Step 2 - If it is EMPTY, then display "Queue is EMPTY!!! Deletion is not possible!!!" and terminate the function.

Step 3 - If it is NOT EMPTY, then display queue[front] as deleted element and increment the front value by one (front ++). Then check whether front == SIZE, if it is TRUE, then set front = 0. Then check whether both front - 1 and rear are equal (front - 1 == rear), if it TRUE, then set both front and rear to '-1' (front = rear = -1).

display() - Displays the elements of a Circular Queue

Step 1 - Check whether queue is EMPTY. (front == -1)

Step 2 - If it is EMPTY, then display "Queue is EMPTY!!!" and terminate the function.

Step 3 - If it is NOT EMPTY, then define an integer variable 'i' and set 'i = front'.

Step 4 - Check whether 'front <= rear', if it is TRUE, then display 'queue[i]' value and increment 'i' value by one (i++). Repeat the same until 'i <= rear' becomes FALSE.

Step 5 - If 'front <= rear' is FALSE, then display 'queue[i]' value and increment 'i' value by one (i++). Repeat the same until 'i <= SIZE - 1' becomes FALSE.

Step 6 - Set i to 0.

Step 7 - Again display 'cQueue[i]' value and increment i value by one (i++). Repeat the same until 'i <= rear' becomes FALSE.

Procedure:

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

```
#define MAX 10
```

```
int queue[MAX];
```

```
int front = -1, rear = -1;
```

```
void enqueue(int value) {
```

```
    if ((rear + 1) % MAX == front) {
```

```
        printf("Queue is full\n");
```

```
    } else {
```

```
        rear = (rear + 1) % MAX;
```

```
        queue[rear] = value;
```

```
        if (front == -1) {
```

```
            front = 0;
```

```
        }
```

```
    }
```

```
}
```

```
void dequeue() {
```

```
    if (front == -1) {
```

```
        printf("Queue is empty\n");
```

```
    } else {
```

```
        printf("Dequeued value: %d\n", queue[front]);
```

```
        if (front == rear) {
            front = -1;
            rear = -1;
        } else {
            front = (front + 1) % MAX;
        }
    }
}
```

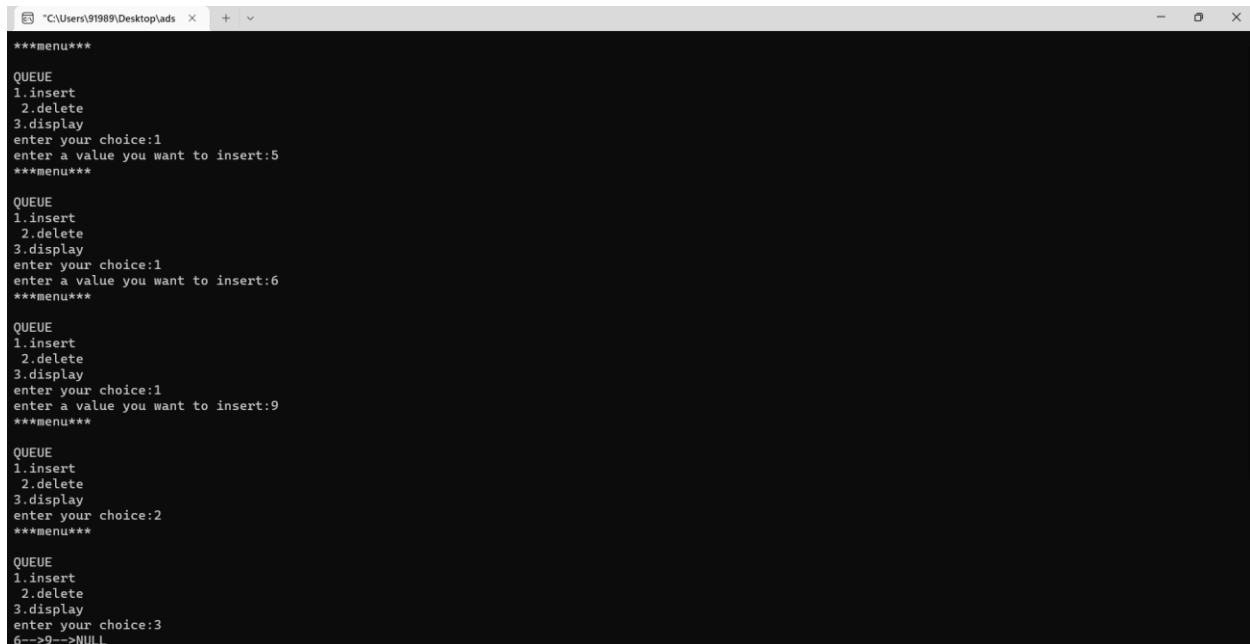
```
void display() {
    if (front == -1) {
        printf("Queue is empty\n");
    } else {
        for (int i = front; i <= rear; i++) {
            printf("%d ", queue[i]);
        }
        printf("\n");
    }
}
```

```
int main() {
    int choice, value;
    printf("*****MENU*****\n\n");
    while (1) {
        printf("1. Enqueue\n");
        printf("2. Dequeue\n");
        printf("3. Display\n");
        printf("4. Quit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("Enter the value to be enqueued: ");
                scanf("%d", &value);
                enqueue(value);
                printf("Successfully Enqueued !!\n");
            case 2:
                dequeue();
            case 3:
                display();
            case 4:
                return 0;
        }
    }
}
```

```
        break;
        case 2:
            dequeue();
            printf("Successfully Dequeued !!\n");
            break;
        case 3:

            display();
            break;
        case 4:
            exit(0);
        default:
            printf("Invalid choice\n");
    }
}
return 0;
}
```

Output Screenshot



```
***menu***
QUEUE
1.insert
2.delete
3.display
enter your choice:1
enter a value you want to insert:5
***menu***

QUEUE
1.insert
2.delete
3.display
enter your choice:1
enter a value you want to insert:6
***menu***

QUEUE
1.insert
2.delete
3.display
enter your choice:1
enter a value you want to insert:9
***menu***

QUEUE
1.insert
2.delete
3.display
enter your choice:2
***menu***

QUEUE
1.insert
2.delete
3.display
enter your choice:3
6-->9-->NULL
```

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 function.

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 12 - If 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).

Procedure:

```
#include<stdio.h>
#include<stdlib.h>
void insertAtBeg(int);
void insertAtEnd(int);
int insertAtLocation(int,int);
void deleteFromBeg();
void deleteFromEnd();
void deleteFromLocation(int);
void display();
struct Node
{
    int data;
    struct Node *next;
    struct Node *prev;
}*head=NULL,*prev=NULL,*next=NULL,*temp;
void main()
```

```

{
    int a,c,l;
    while(1)
    {
        printf("\n**DOUBLY LINKED LIST*\n");
        printf(" Insert \n 1.at beginning\t 2.at end\t 3. at specific location\n Delete \n 4.from
beginning\t 5.delete from end\t 6.delete from specific location\n 7.display\n 8.Exit\n");
        printf("Enter your choice:");
        scanf("%d",&c);
        switch(c)
        {
            case 1:printf("Enter the value:");
                scanf("%d",&a);
                insertAtBeg(a);
                break;
            case 2:printf("Enter the value: ");
                scanf("%d",&a);
                insertAtEnd(a);
                break;
            case 3:
                printf("Enter the location: ");
                scanf("%d",&l);
                printf("Enter the value:");
                scanf("%d",&a);
                insertAtLocation(a,l);
                break;
            case 4:
                deleteFromBeg();
                break;
            case 5:
                deleteFromEnd();
                break;
            case 6:printf("Enter the value: ");
                scanf("%d",&a);
                deleteFromLocation(a);
                break;
            case 7:display();
                break;
            case 8:exit(0);
                break;
            default:printf("Wrong selection!!!\n");
        }
    }
}

void insertAtBeg(int a)
{
    struct Node * newNode= (struct Node*)malloc(sizeof(struct Node));

```

```
newNode->data = a;
newNode->prev = NULL;
if(head==NULL)
{
    newNode->next=NULL;
    head=newNode;
}

else
{
    newNode->next=head;
    head->prev=newNode;
    head=newNode;
}

}
void insertAtEnd(int a)
{
    struct Node * newNode= (struct Node*)malloc(sizeof(struct Node));
    newNode->data = a;
    newNode->next = NULL;
    if(head==NULL)
    {
        newNode->prev=NULL;
        head=newNode;
    }

    else
    {
        temp=head;
        while(temp->next!=NULL)
        {
            temp=temp->next;
        }
        temp->next=newNode;
        newNode->prev=temp;
    }
}
int insertAtLocation(int a,int l)
{
    struct Node * newNode= (struct Node*)malloc(sizeof(struct Node));
    newNode->data = a;
    //newnode->next=NULL;
    // newNode->prev=NULL;
    if(head==NULL)
    {
```

```
    printf("The list is empty");
    //exit(0);
}

else
{
    int f=0;
    temp=head;
    while(temp!=NULL)
    {
        if(temp->data==l)
        {
            f=1;
            break;
        }
        temp=temp->next;
        //printf("\n\n%d",temp->data);
    }
    if(f==0)
    {
        printf("Location not found\n");
        return 0;
    }
    else
    {

        if(temp->next==NULL)
        {
            temp->next=newNode;
            newNode->next=NULL;
            newNode->prev=temp;
        }
        else
        {

            newNode->next=temp->next;
            newNode->next->prev=newNode;
            newNode->prev=temp;
            temp->next=newNode;

        }

    }
}
```

```
}

void deleteFromBeg()
{
    if(head == NULL)
    {
        printf("List is empty");
        return 0;
    }
    if(head->next==NULL)
    {
        temp=head;
        head=NULL;
        free(temp);
    }
    else
    {
        temp=head;
        head=head->next;
        head->prev=NULL;

        free(temp);
    }
}

void deleteFromEnd()
{
    if(head == NULL)
    {
        printf("List is empty");
        return 0;
    }
    if(head->next==NULL)
    {
        temp=head;
        head=NULL;
        free(temp);
    }
    else
    {
        temp=head;
        while(temp->next!=NULL)
        {
```

```
        temp=temp->next;
    }
    temp->prev->next=NULL;
    free(temp);
}
}
void deleteFromLocation(int a)
{
    if(head==NULL)
    {
        printf("Empty");
    }
    else
    {
        struct Node *temp=head;
        while(temp->data!=a)
        {
            if(temp->next==NULL)
            {
                printf("Value not found");
                return 0;
            }
            else
            {
                temp=temp->next;
            }
        }
        if(temp==head)
        {
            head=NULL;
            free(temp);
        }
        else
        {
            temp->prev->next=temp->next;
            free(temp);
        }
        printf("Node deleted");
    }
}
```

```
void display()
{
    if(head==NULL)
        printf("List is empty");
    else
```

```
{
temp=head;
printf("\n list elements are-\n");
while(temp->next!=NULL)
{
printf("%d<===>",temp->data);
temp=temp->next;
}

printf("%d<===>NULL",temp->data);
}
}
```

Output Screenshot

```
**DOUBLY LINKED LIST*
Insert
1.at beginning 2.at end      3. at specific location
Delete
4.from beginning      5.delete from end      6.delete from specific location
7.display
8.Exit
Enter your choice:1
Enter the value:2

**DOUBLY LINKED LIST*
Insert
1.at beginning 2.at end      3. at specific location
Delete
4.from beginning      5.delete from end      6.delete from specific location
7.display
8.Exit
Enter your choice:1
Enter the value:5

**DOUBLY LINKED LIST*
Insert
1.at beginning 2.at end      3. at specific location
Delete
4.from beginning      5.delete from end      6.delete from specific location
7.display
8.Exit
Enter your choice:2
Enter the value: 9

**DOUBLY LINKED LIST*
Insert
1.at beginning 2.at end      3. at specific location
Delete
4.from beginning      5.delete from end      6.delete from specific location
7.display
8.Exit
Enter your choice:3
Enter the location: 9
Enter the value:
```

Result

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

Experiment No.: 6**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(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] or array3[i])
9. Stop

Procedure:

```
#include<stdio.h>
#include<stdlib.h>
void setunion();
void setintersection();
void setdifference();
void setequality();
int s1[10],s2[10],n1,n2,s[10],i;
int main()
{
printf("Enter the number of elements in first set : ");
scanf("%d",&n1);
```

```
printf("Enter the elements : ");
for(i=0;i<n1;i++)
{
    scanf("%d",&s1[i]);
}
printf("Enter the number of elements in second set : ");
scanf("%d",&n2);
printf("Enter the elements : ");
for(i=0;i<n2;i++)
{
    scanf("%d",&s2[i]);
}
int c;
do
{
    printf("\n\n****MENU****\n1.Union\n2.Intersection\n3.Set difference\n4.Equality of
set\n5.Exit\nEnter your choice : ");
    scanf("%d",&c);
    switch(c)
    {
        case 1:
        {
            setunion();
            break;
        }
        case 2:
        {
            setintersection();
            break;
        }
        case 3:
```

```
{
    setdifference();
    break;
}
case 4:
{
    setequality();
    break;
}
case 5:
{
    exit(0);
}
default:
{
    printf("Invalid choice ...");
}
}
}while(c!=6);
}
void setunion()
{
    if(n1!=n2)
    {
        printf("Two sets are not compatible to union");
    }
    else
    {
        for(i=0;i<n1;i++)
        {
            s[i]=s1[i] || s2[i];
        }
    }
}
```

```
    }
    printf("first set : ");
    for(i=0;i<n1;i++)
    {
        printf("%d\t",s1[i]);
    }
    printf("\nsecond set : ");
    for(i=0;i<n2;i++)
    {
        printf("%d\t",s2[i]);
    }
    printf("\nResultant set : ");
    for(i=0;i<n2;i++)
    {
        printf("%d\t",s[i]);
    }
}

}

void setintersection()
{
    if(n1!=n2)
    {
        printf("Two sets are not compatible to intersection");
    }
    else
    {
        for(i=0;i<n1;i++)
        {
            s[i]=s1[i] && s2[i];
        }
    }
}
```

```
    printf("first set : ");
    for(i=0;i<n1;i++)
    {
        printf("%d\t",s1[i]);
    }
    printf("\nsecond set : ");
    for(i=0;i<n2;i++)
    {
        printf("%d\t",s2[i]);
    }
    printf("\nResultant set : ");
    for(i=0;i<n2;i++)
    {
        printf("%d\t",s[i]);
    }
}

}

void setdifference()
{
    int s3[10];
    if(n1!=n2)
    {
        printf("Two sets are not compatible for difference");
    }
    else
    {
        for(i=0;i<n1;i++)
        {
            s3[i]=!s2[i];
        }
    }
}
```

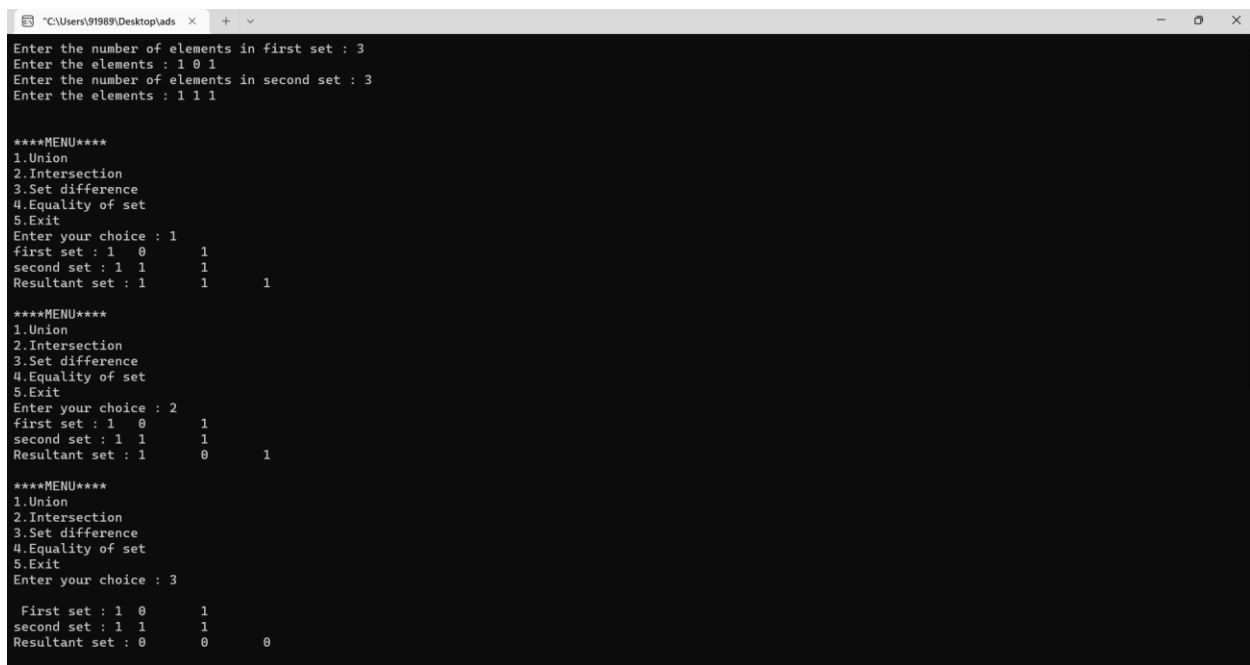
```
        for(i=0;i<n1;i++)
        {
            s[i]=s1[i] && s3[i];
        }
        printf("\n First set : ");
        for(i=0;i<n2;i++)
        {
            printf("%d\t",s1[i]);
        }
        printf("\nsecond set : ");
        for(i=0;i<n2;i++)
        {
            printf("%d\t",s2[i]);
        }
        printf("\nResultant set : ");
        for(i=0;i<n2;i++)
        {
            printf("%d\t",s[i]);
        }
    }

}

void setequality()
{
    if(n1!=n2)
    {
        printf("\nTwo sets are not equal.");
    }
    else
    {
        for(i=0;i<n1;i++)
```

```
{  
    if(s1[i]!=s2[i])  
    {  
        printf("\nThe sets are not equal");  
        return 0;  
    }  
}  
printf("\nThe sets are equal.");  
}  
}
```

Output Screenshot



```
"C:\Users\91989\Desktop\lads  x  +  v  
Enter the number of elements in first set : 3  
Enter the elements : 1 0 1  
Enter the number of elements in second set : 3  
Enter the elements : 1 1 1  
  
****MENU****  
1.Union  
2.Intersection  
3.Set difference  
4.Equality of set  
5.Exit  
Enter your choice : 1  
first set : 1  0      1  
second set : 1  1      1  
Resultant set : 1      1      1  
  
****MENU****  
1.Union  
2.Intersection  
3.Set difference  
4.Equality of set  
5.Exit  
Enter your choice : 2  
first set : 1  0      1  
second set : 1  1      1  
Resultant set : 1      0      1  
  
****MENU****  
1.Union  
2.Intersection  
3.Set difference  
4.Equality of set  
5.Exit  
Enter your choice : 3  
First set : 1  0      1  
second set : 1  1      1  
Resultant set : 0      0      0
```

Result

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

Experiment No.: 7**Aim:**

Implementation of Binary Search Tree

CO3:

Understand the practical aspects of Advanced Tree Structures.

Algorithm:**Search Operation in BST**

Step 1 - Read the search element from the user.

Step 2 - Compare the search element with the value of root node in the tree.

Step 3 - If both are matched, then display "Given node is found!!!" and terminate the function

Step 4 - If both are not matched, then check whether search element is smaller or larger than that node value.

Step 5 - If search element is smaller, then continue the search process in left subtree.

Step 6- If search element is larger, then continue the search process in right subtree.

Step 7 - Repeat the same until we find the exact element or until the search element is compared with the leaf node

Step 8 - If we reach to the node having the value equal to the search value then display "Element is found" and terminate the function.

Step 9 - If we reach to the leaf node and if it is also not matched with the search element, then display "Element is not found" and terminate the function.

Insertion Operation in BST

Step 1 - Create a newNode with given value and set its left and right to NULL.

Step 2 - Check whether tree is Empty.

Step 3 - If the tree is Empty, then set root to newNode.

Step 4 - If the tree is Not Empty, then check whether the value of newNode is smaller or larger than the node (here it is root node).

Step 5 - If newNode is smaller than or equal to the node then move to its left child. If newNode is larger than the node then move to its right child.

Step 6- Repeat the above steps until we reach to the leaf node (i.e., reaches to NULL).

Step 7 - After reaching the leaf node, insert the newNode as left child if the newNode is smaller or equal to that leaf node or else insert it as right child.

Deletion Operation in BST

Case 1: Deleting a leaf node

Step 1 - Find the node to be deleted using search operation

Step 2 - Delete the node using free function (If it is a leaf) and terminate the function.

Case 2: Deleting a node with one child

Step 1 - Find the node to be deleted using search operation

Step 2 - If it has only one child then create a link between its parent node and child node.

Step 3 - Delete the node using free function and terminate the function.

Case 3: Deleting a node with two children

Step 1 - Find the node to be deleted using search operation

Step 2 - If it has two children, then find the largest node in its left subtree (OR) the smallest node in its right subtree.

Step 3 - Swap both deleting node and node which is found in the above step.

Step 4 - Then check whether deleting node came to case 1 or case 2 or else goto step 2

Step 5 - If it comes to case 1, then delete using case 1 logic.

Step 6- If it comes to case 2, then delete using case 2 logic.

Step 7 - Repeat the same process until the node is deleted from the tree.

Procedure:

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

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

```
struct node
```

```
{
```

```
    int key;
```

```
    struct node *left, *right;
```

```
}*root=NULL;
```

```
struct node* newNode(int item)
```

```
{
```

```
    struct node* temp
```

```
        = (struct node*)malloc(sizeof(struct node));
```

```
    temp->key = item;
```

```
temp->left = temp->right = NULL;

return temp;
}

void inorder(struct node* root)
{
    if (root != NULL) {
        inorder(root->left);
        printf("%d ", root->key);
        inorder(root->right);
    }
}

struct node* insert(struct node* node, int key)
{
    if (node == NULL)
        return newNode(key);

    if (key < node->key)
        node->left = insert(node->left, key);
    else
        node->right = insert(node->right, key);

    return node;
}

struct node* minValueNode(struct node* node)
{
    struct node* current = node;

    while (current && current->left != NULL)
        current = current->left;
```

```
    return current;
}

struct node* deleteNode(struct node* root, int key)
{
    // base case
    if (root == NULL){
        printf("Item not found!");
        return root;
    }

    else if (key < root->key)
        root->left = deleteNode(root->left, key);

    else if (key > root->key)
        root->right = deleteNode(root->right, key);

    else {

        if (root->left == NULL) {
            struct node* temp = root->right;
            free(root);
            return temp;
        }

        else if (root->right == NULL) {
            struct node* temp = root->left;
            free(root);
            return temp;
        }
    }
}
```

```
    struct node* temp = minValueNode(root->right);

    root->key = temp->key;

    root->right = deleteNode(root->right, temp->key);
}
return root;
}

int iterativeSearch(struct node* node, int key)
{
    // Traverse untill root reaches to dead end
    while (node != NULL)
    {
        if(key == node->key){
            return 1;
        } // if the key is found return 1

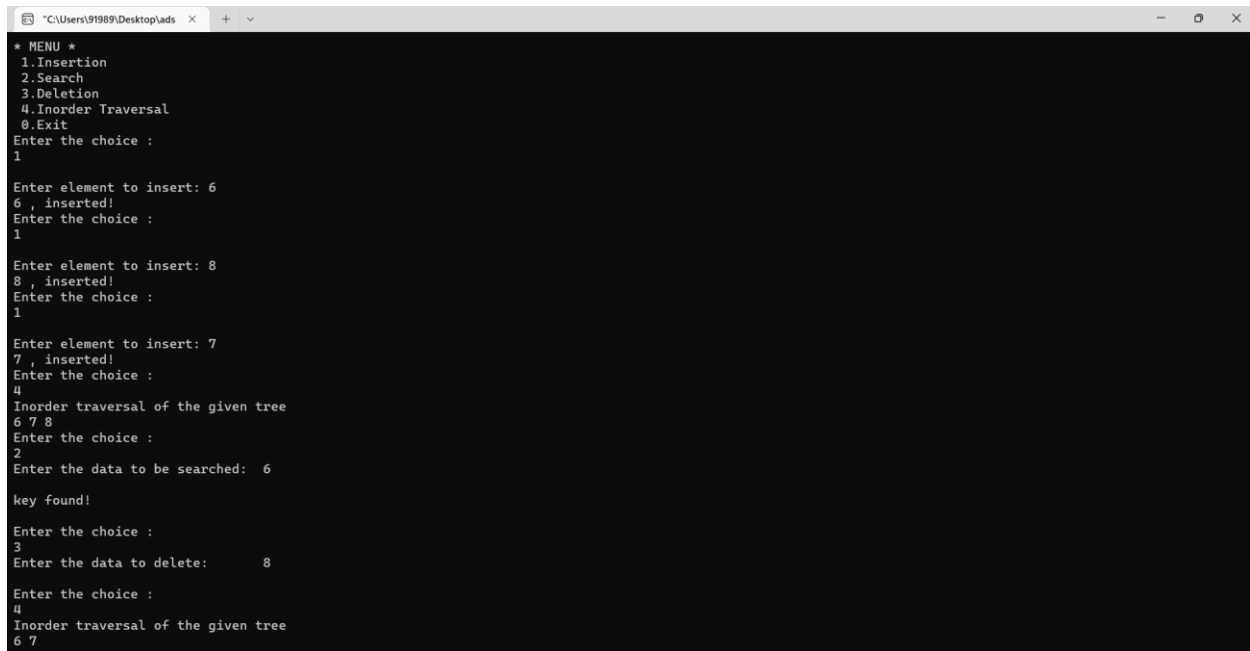
        // pass right subtree as new tree
        else if (key > node->key)
            node = node->right;

        // pass left subtree as new tree
        else
            node = node->left;
    }
    return 0;
}
```

```
int main(){
int ch,res,elt,item;
printf( "* MENU *");
printf("\n 1.Insertion ");
printf("\n 2.Search ");
printf("\n 3.Deletion ");
printf("\n 4.Inorder Traversal ");
printf("\n 0.Exit");
while(1)
{
    printf("\nEnter the choice : \n");
    scanf("%d", &ch);
    switch(ch){
        case 1:
            printf("\nEnter element to insert: ");
            scanf("%d",&elt);
            root = insert(root,elt);
            printf("%d , inserted! ",elt);
            break;
        case 2:
            printf("Enter the data to be searched:\t");
            scanf("%d",&item);
            res = iterativeSearch(root,item);
            if(res==1){
                printf("\nkey found!\n");
            }
            else{
                printf("\nkey not found!\n");
            }
        }
    }
```

```
    }  
    break;  
case 3:  
    if(root == NULL)  
        printf("empty binary search tree! \n");  
    else{  
        printf("Enter the data to delete:\t");  
        scanf("%d",&item);  
        root = deleteNode(root, item);  
    }  
    break;  
case 4:  
    if(root == NULL)  
        printf("empty binary search tree! \n");  
    else{  
        printf("Inorder traversal of the given tree \n");  
        inorder(root);  
    }  
    break;  
case 0:  
    printf("\nExited!\n");  
    exit(0);  
}  
}
```

Output Screenshot



```
* MENU *
1.Insertion
2.Search
3.Deletion
4.Inorder Traversal
0.Exit
Enter the choice :
1

Enter element to insert: 6
6 , inserted!
Enter the choice :
1

Enter element to insert: 8
8 , inserted!
Enter the choice :
1

Enter element to insert: 7
7 , inserted!
Enter the choice :
4
Inorder traversal of the given tree
6 7 8
Enter the choice :
2
Enter the data to be searched: 6
key found!

Enter the choice :
3
Enter the data to delete: 8

Enter the choice :
4
Inorder traversal of the given tree
6 7
```

Result

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

Experiment No.: 8**Aim:**

Implementation of Binomial Heap.

CO4:

Realise Modern Heap Structures for effectively solving advanced Computational problems.

Algorithm:**Union :**

Given binomial heaps H1 and H2

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, ptr 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.

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 - $O(\log n)$

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



```
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: 9
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: 6
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: 1
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: 1(1) 3(0) 4(2) 6(1) 9(0) 7(0)
```

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.

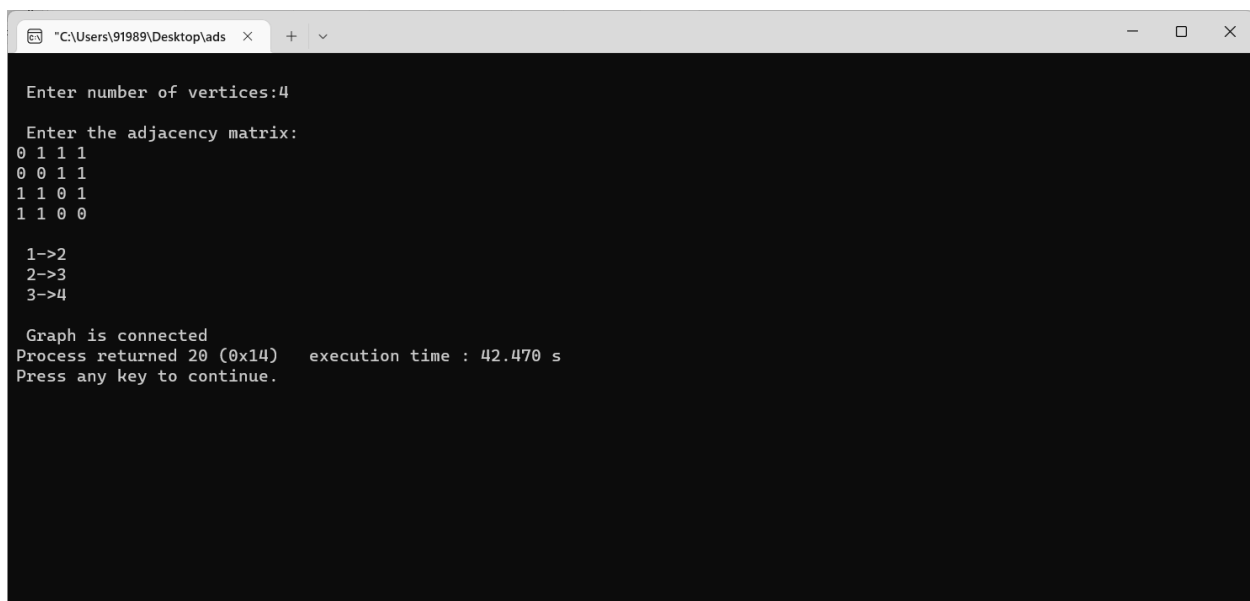
Step 7 - When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph

Procedure:

```
#include<stdio.h>
int a[20][20],reach[20],n;
void dfs(int v)
{
    int i;
    reach[v]=1;
    for (i=1;i<=n;i++)
        if(a[v][i] && !reach[i])
        {
            printf("\n %d->%d",v,i);
            dfs(i);
        }
}
void main()
{
    int i,j,count=0;
    printf("\n Enter number of vertices:");
```

```
scanf("%d",&n);
for (i=1;i<=n;i++)
{
    reach[i]=0;
    for (j=1;j<=n;j++)
        a[i][j]=0;
}
printf("\n Enter the adjacency matrix:\n");
for (i=1;i<=n;i++)
    for (j=1;j<=n;j++)
        scanf("%d",&a[i][j]);
dfs(1);
printf("\n");
for (i=1;i<=n;i++)
{
    if(reach[i])
        count++;
}
if(count==n)
    printf("\n Graph is connected");
else
    printf("\n Graph is not connected");
}
```

Output Screenshot



```
"C:\Users\91989\Desktop\ads" x + v
Enter number of vertices:4
Enter the adjacency matrix:
0 1 1 1
0 0 1 1
1 1 0 1
1 1 0 0

1->2
2->3
3->4

Graph is connected
Process returned 20 (0x14)    execution time : 42.470 s
Press any key to continue.
```

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.

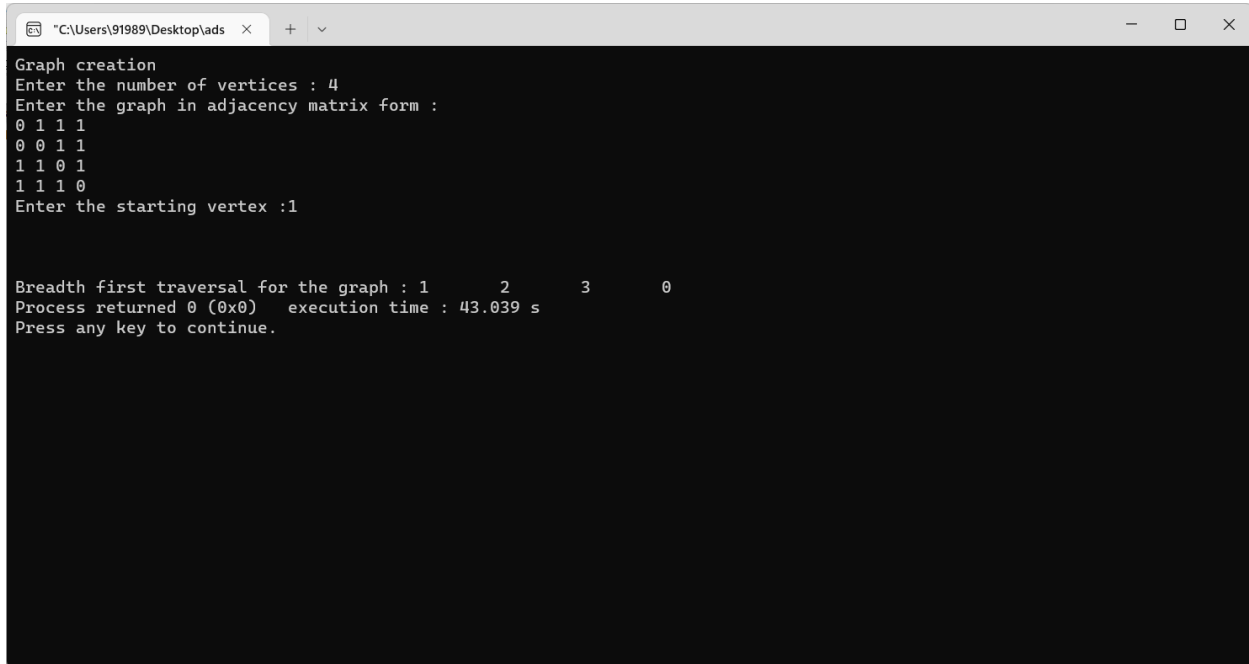
Step 6 - When queue becomes empty, then produce final spanning tree by removing unused edges from the graph

Procedure:

```
#include<stdio.h>
int a[20][20],v,f,i,j,q[20],front=0,rear=0,visited[20],m=0,s,k;
int main()
{
    printf("Graph creation\n");
    printf("Enter the number of vertices : ");
    scanf("%d",&v);
    printf("Enter the graph in adjacency matrix form :\n");
    for(i=0;i<v;i++)
    {
        for(j=0;j<v;j++)
        {
            scanf("%d",&a[i][j]);
        }
    }
    printf("Enter the starting vertex :");
    scanf("%d",&s);
```

```
i=s;
q[front]=i;
printf("\n\nBreadth first traversal for the graph : %d\t",q[front]);
visited[m]=i;
m=m+1;
j=0;
while(front<=rear)
{
    if(a[i][j]==1)
    {
        f=0;
        for(k=0;k<m;k++)
        {
            if(visited[k]==j)
            {
                f=1;
                break;
            }
        }
        if(f==0)
        {
            q[rear]=j;
            printf("%d\t",q[rear]);
            rear=rear+1;
            visited[m]=j;
            m=m+1;
        }
    }
    j=j+1;
    if(j==v)
    {
        i=q[front];
        front=front+1;
        j=0;
    }
}
```

Output Screenshot



```
Graph creation
Enter the number of vertices : 4
Enter the graph in adjacency matrix form :
0 1 1 1
0 0 1 1
1 1 0 1
1 1 1 0
Enter the starting vertex :1

Breadth first traversal for the graph : 1      2      3      0
Process returned 0 (0x0)   execution time : 43.039 s
Press any key to continue.
```

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<stdio.h>
#include<conio.h>
int a,b,u,v,n,i,j,ne=1;
int visited[10]= {0};
int min,mincost=0,cost[10][10];
void main()
{
    printf("\n Enter the number of nodes:");
    scanf("%d",&n);
    printf("\n Enter the adjacency matrix:\n");
    for (i=1;i<=n;i++)
        for (j=1;j<=n;j++) {
            scanf("%d",&cost[i][j]);
            if(cost[i][j]==0)
                cost[i][j]=999;
        }
}
```

```

    }
    visited[1]=1;
    printf("\n");
    while(ne<n) {
        for (i=1,min=999;i<=n;i++)
            for (j=1;j<=n;j++)
                if(cost[i][j]<min)
                    if(visited[i]!=0) {
                        min=cost[i][j];
                        a=u=i;
                        b=v=j;
                    }
        if(visited[u]==0 || visited[v]==0) {
            printf("\n Edge %d:(%d %d) cost:%d",ne++,a,b,min);
            mincost+=min;
            visited[b]=1;
        }
        cost[a][b]=cost[b][a]=999;
    }
    printf("\n Minimun cost=%d",mincost);
}

```

Output Screenshot

```

C:\Users\91989\Desktop\ads >
Enter the number of nodes:4

Enter the adjacency matrix:
0 1 0 1
1 0 1 1
1 1 0 0
0 0 1 0

Edge 1:(1 2) cost:1
Edge 2:(1 4) cost:1
Edge 3:(2 3) cost:1
Minimun cost=3
Process returned 16 (0x10)    execution time : 29.869 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:

```
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
```

Procedure:

```
#include<stdio.h>
#include<stdlib.h>
#define VAL 999
int i,j,k,a,b,u,v,n,ne=1;
int min,mincost=0,cost[9][9],parent[9];
int find(int i)
{
    while(parent[i])
        i=parent[i];
    return i;
}
int uni(int i,int j)
{
    if(i!=j)
    {
        parent[j]=i;
        return 1;
    }
    return 0;
}
int main()
```

```
{
    printf("Implementation of Kruskal's algorithm\n");
    printf("Enter the no. of vertices:");
    scanf("%d",&n);
    printf("Enter the cost adjacency matrix:\n");
    for(i=1;i<=n;i++)
    {
        for(j=1;j<=n;j++)
        {
            scanf("%d",&cost[i][j]);
            if(cost[i][j]==0)
                cost[i][j]=VAL;
        }
    }
    printf("The edges of Minimum Cost Spanning Tree are\n");
    while(ne < n)
    {
        for(i=1,min=VAL;i<=n;i++)
        {
            for(j=1;j <= n;j++)
            {
                if(cost[i][j] < min)
                {
                    min=cost[i][j];
                    a=u=i;
                    b=v=j;
                }
            }
        }
        u=find(u);
        v=find(v);
        if(uni(u,v))
        {
            printf("%d edge (%d,%d) =%d\n",ne++,a,b,min);
            mincost +=min;
        }
        cost[a][b]=cost[b][a]=999;
    }
    printf("\n\tMinimum cost = %d\n",mincost);
    return 0;
}
```

Output Screenshot

```
Implementation of Kruskal's algorithm
Enter the no. of vertices:3
Enter the cost adjacency matrix:
0 2 6
5 0 7
6 7 0
The edges of Minimum Cost Spanning Tree are
1 edge (1,2) =2
2 edge (1,3) =6

      Minimum cost = 8

Process returned 0 (0x0)   execution time : 11.712 s
Press any key to continue.
```

Result

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