**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 Orgcc -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 $ sudomkdirtst $ 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)***

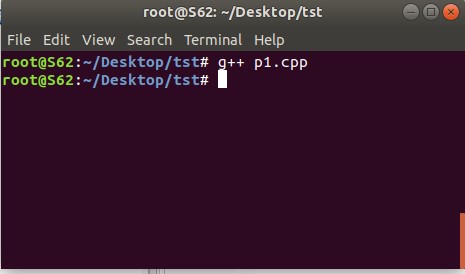
*(or)*

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

***1. $ sudo g++ p1.cpp***

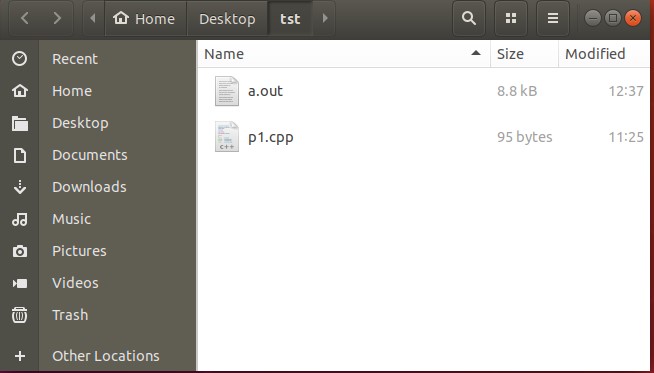
**To compile your c++ code, use:**

*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:

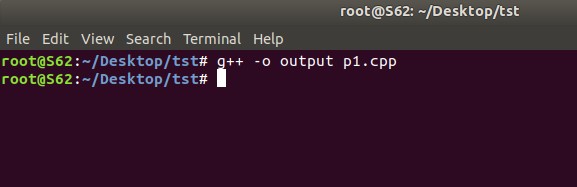
### ./a.out



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

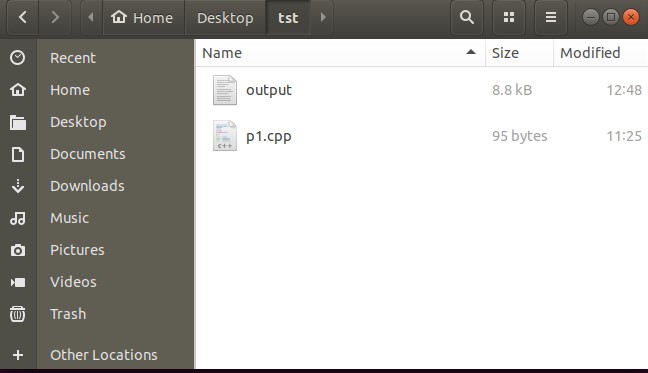
**To specify the name of the compiled output file, so that it is not named** a.out**, use** -o **withyour g++ command.**

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.

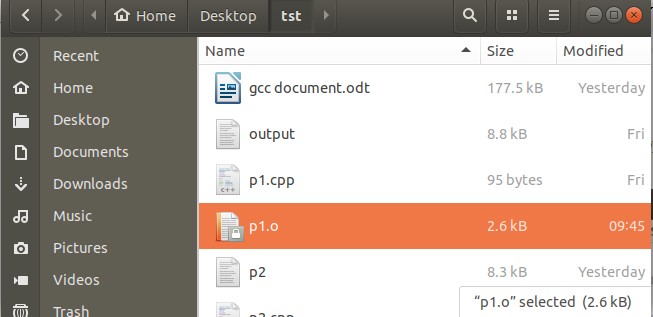
### ./output.out



**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.

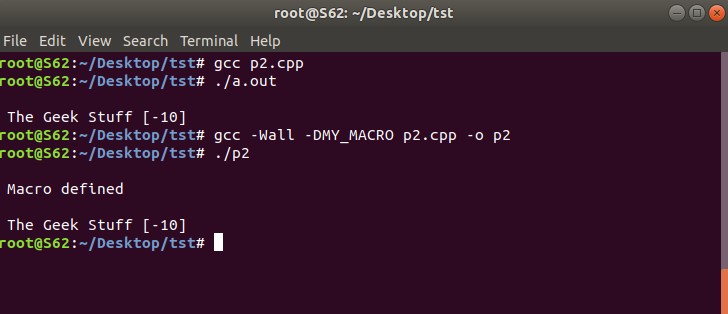
$ gcc -Wall -DMY\_MACRO main.c -o main

$ ./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 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

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<0xffffffee;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\_gproftest\_gprof.ctest\_gprof\_new.c

$ ./test\_gprof

Inside main()

Inside func1

Inside new\_func1()

Inside func2

$ ls

gmon.outtest\_gproftest\_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.

$ gproftest\_gprofgmon.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.outtest\_gproftest\_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]

1. 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]

1. 33.5 15.52 0.00 1 func2 [3]

-----------------------------------------------

15.50 0.00 1/1 func1 [2]

1. 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

1. func1 [1] main
2. 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\_gprofgmon.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\_gprofgmon.out> analysis.txt **3. Print only flat profile using -p**

In case only flat profile is required then :

$ gprof -p -b test\_gprofgmon.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\_gprofgmon.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\_gprofgmon.out> analysis.txt **6. Print only call graph information using -q** gprof -q -b test\_gprofgmon.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\_gprofgmon.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\_gprofgmon.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: read the size of first array

Step 4: read the size of second array

Step 5: read the elements of first array

Step 6: read the elements of second array

Step 7: display first array

Step 8: display second array

Step 9: sort first array

Step 10: sort second array

Step 11: print sorted first array

Step 12: print sorted second array

Step 13: merge first and second array

Step 14: print merged array

Step 15: sort merged array

Step 16: print merged and sorted array

Step 17 : END

**Procedure**

#include<stdio.h>

intmain()

{

int a[20],b[20],i,j,c,n1,n2,n;

printf("Enter the limit of 1st array:");

scanf("%d",&n1);

printf("\nEnter the elements of 1st array: ");

for(i=0;i<n1;i++)

{

scanf("%d",&a[i]);

}

printf("\nEnter the limit of 2nd array:");

scanf("%d",&n2);

printf("\nEnter the elements of 2nd array: ");

for(i=0;i<n2;i++)

{

scanf("%d",&b[i]);

}

for(i=0;i<n1;i++)

{

for(j=i+1;j<n1;j++)

{

if(a[i]>a[j])

{

c=a[i];

a[i]=a[j];

a[j]=c;

}

}

}

printf("Sorted array is: ");

for(i=0;i<n1;i++)

{

printf("%d ",a[i]);

}

for(i=0;i<n2;i++)

{

for(j=i+1;j<n2;j++)

{

if(b[i]>b[j])

{

c=b[i];

b[i]=b[j];

b[j]=c;

}

}

}

printf("\nSorted array is: ");

for(i=0;i<n2;i++)

{

printf("%d ",b[i]);

}

n=n1+n2;

int k=n1;

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

{

a[k]=b[i];

k++;

}

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

{

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

{

if(a[i]>a[j])

{

c=a[i];

a[i]=a[j];

a[j]=c;

}

}

}

printf("\nThe merged array after sorting : ");

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

{

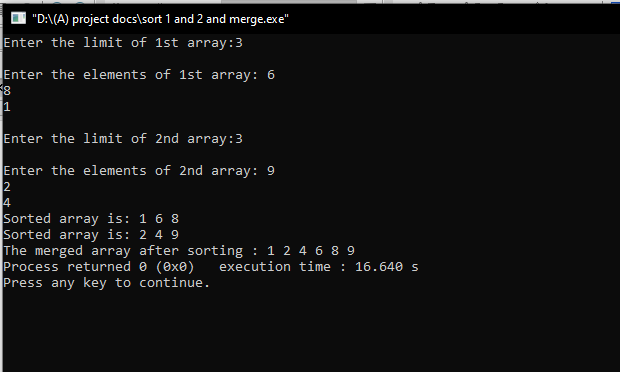
printf("%d ",a[i]);

}

return 0;

}

**Output Screenshot**



**Result**

The program was executed and the result was successfully obtained. Thus CO1was obtained.

**Experiment No.: 3**

**Aim**

Implementation of Singly Linked Stack

**CO1**

Use basic data structures and it’s operations,implementations.

**Algorithm**

Step 1:START

Step 2:declare,choice,value;

Step 3:use while loop to read choice

Step 4:select choice

* Push operations
* Pop operations
* Display operations

Step 5:stop

Push operation

Step1:allocate memory for the new node and name it as newnode

Step 2:setnewnode->data=value

Step 3:if top==NULL

* Set newnode->next=NULL
* Set Top=newnode
* Else
* Set newnode->next=top
* Set top=newnode

[end of if]

Step 4:END

Pop operation

Step1:if top=NULL

* Print(“underflow”);
* Go to step 5

[end of if]

Step 2:set temp=Top

Step 3:set top=temp->next

Step 4:free temp

Step 5:END

Display operation

Step 1:if top=NULL

* Print(“empty”)
* Go to step5

[end of if]

Step 2:set temp=top

Step 3:while(temp->next !=NULL)

* Print temp->data
* Set temp=temp->next
* End while

Step 4:print temp->data

Step 5:Stop

**Procedure**

#include<stdio.h>

#include<stdlib.h>

void push();

void pop();

void display();

struct node

{

int data;

struct node\*next;

};

struct node \*top;

void main()

{

int choice=0;

while(choice!=4)

{

printf("\nchoose one from following list\n");

printf("\n MAIN MENU\n");

printf("\n1.insert \n2.delete \n3.display");

printf("\nenter your choice ");

scanf("%d",&choice);

switch(choice)

{

case 1:

push();

break;

case 2:

pop();

break;

case 3:

display();

break;

default:

printf("choose a valid option");

}

}

}

void push()

{

struct node \*newnode;

int item;

newnode=(struct node\*)malloc(sizeof(struct node));

if (newnode==NULL)

{

printf("not able to insert");

}

else

{

printf("enter the value :");

scanf("%d",&item);

if(top==NULL)

{

newnode->data=item;

newnode->next=NULL;

top=newnode;

}

else{

newnode->data=item;

newnode->next=top;

top=newnode;

}

printf("item pushed");

}

}

void pop()

{

struct node \*newnode;

int item;

if(top==NULL)

{

printf("underflow");

}

else{

item=top->data;

newnode=top;

top=top->next;

free(newnode);

printf("item popped");

}

}

void display()

{

struct node \*newnode;

newnode = top;

if(newnode == NULL)

{

printf("the list is empty");

}

else

{

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

while (newnode!=NULL)

{

printf("\n%d",newnode->data);

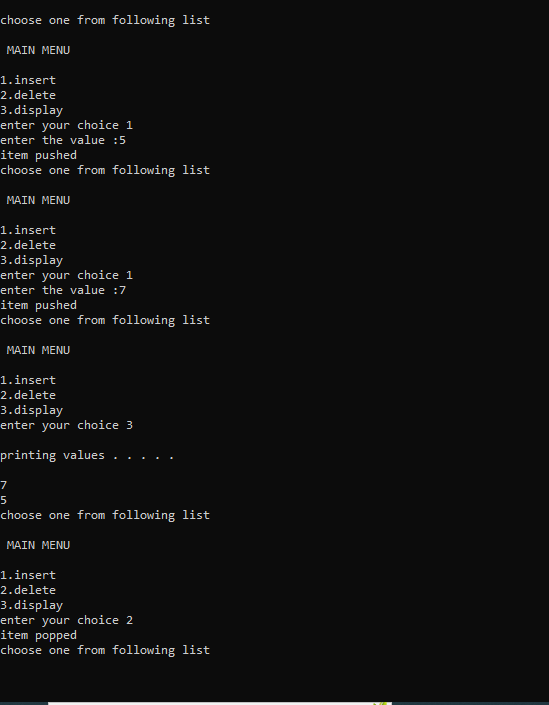
newnode = newnode -> next;

}

}

}

**Output Screenshot**



**Result**

The program was executed and the result was successfully obtained. Thus CO1was obtained.

**Experiment No.: 4**

**Aim**

Implementation of Circular Queue.

**CO1**

Use basic data structures and its operations, implementations.

**Algorithm**

ENQUEUE()

Step 1:START

Step 2:create a struct node type node

Step 3:insert the given data in the newnode data section and NULL in address section.

Step 4:If queue is empty then initialize front and rear from newnode

Step 5:queue is not empty then initialize rear next and rear from newnode

Step 6:newnode next initialize from front

Step 7:STOP

DEQUEUE()

Step1:START

Step 2:check if queue is empty or not

Step 3:if queue is empty then dequeuer is not possible

Step 4:else initialize temp from front

Step 5:if front is equal to the rear then initialize front and rear from NULL

Step 6:print data of temp and free temp memory.

Step 7:if there is more than one node in queue then make front next to front then initialize from front

Step 8:print temp and free temp

Step 9:STOP

DISPLAY()

step 1:START

step 2:check if there is some data in the queue or not.

Step 3:if the queue is empty print “no data in the queue”

Step 4:else define a node pointer and initialize it with front.

Step 5:print data of node pointer until the next of node pointer become NULL.

Step 6:STOP

**Procedure**

#include<stdio.h>

#include<stdlib.h>

struct node

{

int data;

struct node\* next;

};

struct node \*f = NULL;

struct node \*r = NULL;

void enqueue(int d)

{

struct node\* n;

n = (struct node\*)malloc(sizeof(struct node));

n->data = d;

n->next = NULL;

if((r==NULL)&&(f==NULL))

{

f = r = n;

r->next = f;

}

else

{

r->next = n;

r = n;

n->next = f;

}

}

void dequeue()

{

struct node\* t;

t = f;

if((f==NULL)&&(r==NULL))

printf("\nQueue is Empty");

else if(f == r){

f = r = NULL;

free(t);

}

else{

f = f->next;

r->next = f;

free(t);

}

}

void print(){

struct node\* t;

t = f;

if((f==NULL)&&(r==NULL))

printf("\nQueue is Empty");

else{

do{

printf("\n%d",t->data);

t = t->next;

}while(t != f);

}

}

Int main()

{

Int opt,n,i,data;

printf("Enter Your Choice:-");

do{

printf("\n\n1 for Insert the Data in Queue\n2 for show the Data in Queue \n3 for Delete the data from the Queue\n0 for Exit \n");

scanf("%d",&opt);

switch(opt){

case 1:

printf("\nEnter the number of data \n");

scanf("%d",&n);

printf("\nEnter your data\n");

i=0;

while(i<n){

scanf("%d",&data);

enqueue(data);

i++;

}

break;

case 2:

print();

break;

case 3:

dequeue();

break;

case 0:

break;

default:

printf("\nIncorrect Choice\n");

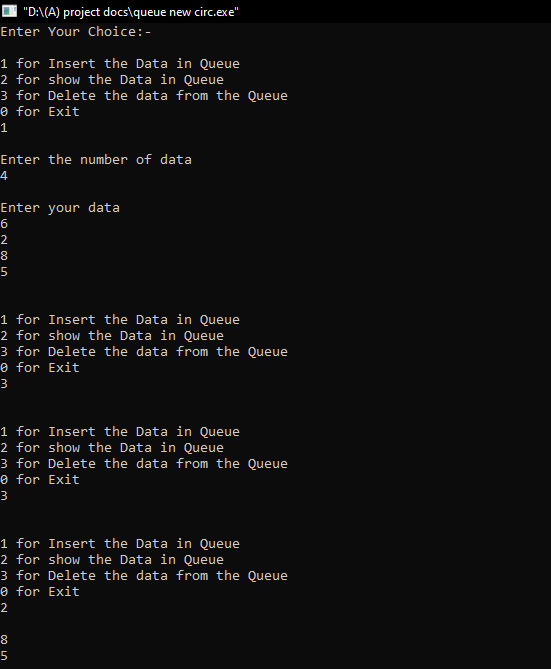
}

}while(opt!=0);

return 0;

}

**Output Screenshot**



**Result**

The program was executed and the result was successfully obtained. Thus CO1was obtained.

**Experiment No.: 5**

**Aim**

To implement Doubly Linked List.

**CO1**

Use basic data structures and it’s operations, implementations.

**Algorithm**

Step 1:START

Step 2:initialize node

Step 3:declare choice

Step 4:use while loop to read the choice from the user

Step 5:select choice

1. : insert at beginning
2. : insert at end
3. : insert at intermediate position
4. : delete at beginning
5. :delete from end
6. :delete from the intermediate position
7. :display

Step 6:STOP

Insertion at beginning

Step1:START

Step 2:create node and allocate the memory

Step 3:check whether newnode is NULL or not

3.1:If true print overflow

Step 4:accept the number from the user

Step 5:check whether the head is NULL or not

* Newnode->next=NULL
* Newnode->prev=NULL
* Head = Newnode

Step 6:else,

* Newnode ->data=item
* Newnode->prev=NULL
* Newnode ->next=head
* Head=newnode

Step 7:STOP

Insertion at End

Step1:START

Step 2:create node and allocate the memory

Step 3:check whether the newnode is NULL or not

Step 4: if TRUE print overflow

Step 5: else,

* Read the value from the user
* Newnode->data=item

Step 6:check if(head==NULL)

1. :print list is empty
2. :newnode->next=NULL
3. :newnode->prev=NULL
4. :head=newnode

Step 7:else,

1. :temp=head
2. :While(temp->next !=NULL)
3. :Temp=temp->next
4. :temp->next=newnode
5. :Newnode->prev=temp
6. :Newnode->next=NULL

Step 8:STOP

Insertion at intermediate

Step 1:NULL

Step 2:create node and allocate memory for node

Step 3:check whether the newnode is NULL or not if yes print OVERFLOW

Step 4:else,

1. :temp=head
2. :read the data from the user
3. :read the location to insert the node
4. :for(i=0;i<loc;i++)
5. :temp=temp->next
6. :If(temp==NULL)
7. :Print there is less elements
8. :Newnode->data=item
9. :Newnode->next=temp->next
10. Newnode ->prev=temp
11. Temp->next=newnode
12. Print node inserted

Step 5:Stop

Deletion at beginning

Step 1:START

Step 2:initialize newnode

Step 3:check whether head is NULL or not. If yes print Underflow

Step 4:else if(head->next==NULL)

* Head=NULL
* Free(head)
* Print node deleted

Step 5:else

* Newnode=head
* Head=head->next
* Head->prev=NULL
* Free(newnode)
* Printf(“node deleted”)

Step 6:STOP

Deletion at last

Step 1:START

Step 2:initialize newnode

Step3:check whether head is NULL or not if yes print underflow

Step 4:else if(head->next==NULL)

* Head=NULL
* Free(head)
* Print node deleted

Step 5:else,

* Newnode=head
* If(newnode->next !=NULL)
* Newnode=newnode->next
* Newnode->prev->next=NULL
* Free(newnode)
* Print node deleted

Step 6:STOP

Delete at intermediate

Step 1:START

Step 2:initialize newnode and temp

Step 3:read the data from the user which the user wants to delete

Step 4:set newnode=head

* While(newnode->data !=item)
* Newnode=newnode->next
* If(newnode->next==NULL)
* Print can’t delete

Step 5:else

* Temp=newnode->next
* Newnode->next=temp->next
* Temp->next->prev=newnode
* Free(temp)

Step 6:STOP

Display

Step 1:START

Step 2:initialize newnode

Step 3:newnode=head

Step 4:while newnode !=NULL

Step 5:print data

Step 6:STOP

**Procedure**

#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 main ()

{

int choice =0;

while(choice != 8)

{

printf("\nMain Menu\n");

printf("\nChoose one option from the following list ...\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 the node after the given data\n7.Display\n8.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:

display();

break;

case 8:

exit(0);

break;

default:

printf("Please enter valid choice..");

}

}

}

void insertion\_beginning()

{

struct node \*newnode;

int item;

newnode = (struct node \*)malloc(sizeof(struct node));

if(newnode == NULL)

{

printf("\nOVERFLOW");

}

else

{

printf("\nEnter Item value");

scanf("%d",&item);

if(head==NULL)

{

newnode->next = NULL;

newnode->prev=NULL;

newnode->data=item;

head=newnode;

}

else

{

newnode->data=item;

newnode->prev=NULL;

newnode->next = head;

head->prev=newnode;

head=newnode;

}

printf("\nNode inserted\n");

}

}

void insertion\_last()

{

struct node \*newnode,\*temp;

int item;

newnode = (struct node \*) malloc(sizeof(struct node));

if(newnode == NULL)

{

printf("\nOVERFLOW");

}

else

{

printf("\nEnter value");

scanf("%d",&item);

newnode->data=item;

if(head == NULL)

{

newnode->next = NULL;

newnode->prev = NULL;

head = newnode;

}

else

{

temp = head;

while(temp->next!=NULL)

{

temp = temp->next;

}

temp->next = newnode;

newnode ->prev=temp;

newnode->next = NULL;

}

}

printf("\nnode inserted\n");

}

void insertion\_specified()

{

struct node \*newnode,\*temp;

int item,loc,i;

newnode = (struct node \*)malloc(sizeof(struct node));

if(newnode == 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);

newnode->data = item;

newnode->next = temp->next;

newnode -> prev = temp;

temp->next = newnode;

temp->next->prev=newnode;

printf("\nnode inserted\n");

}

}

void deletion\_beginning()

{

struct node \*newnode;

if(head == NULL)

{

printf("\n UNDERFLOW");

}

else if(head->next == NULL)

{

head = NULL;

free(head);

printf("\nnode deleted\n");

}

else

{

newnode = head;

head = head -> next;

head -> prev = NULL;

free(newnode);

printf("\nnode deleted\n");

}

}

void deletion\_last()

{

struct node \*newnode;

if(head == NULL)

{

printf("\n UNDERFLOW");

}

else if(head->next == NULL)

{

head = NULL;

free(head);

printf("\nnode deleted\n");

}

else

{

newnode = head;

if(newnode->next != NULL)

{

newnode = newnode -> next;

}

newnode -> prev -> next = NULL;

free(newnode);

printf("\nnode deleted\n");

}

}

void deletion\_specified()

{

struct node \*newnode, \*temp;

int val;

printf("\n Enter the data after which the node is to be deleted : ");

scanf("%d", &val);

newnode = head;

while(newnode -> data != val)

newnode = newnode -> next;

if(newnode -> next == NULL)

{

printf("\nCan't delete\n");

}

else if(newnode -> next -> next == NULL)

{

newnode ->next = NULL;

}

else

{

temp = newnode -> next;

newnode -> next = temp -> next;

temp -> next -> prev = newnode;

free(temp);

printf("\nnode deleted\n");

}

}

void display()

{

struct node \*newnode;

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

newnode = head;

while(newnode != NULL)

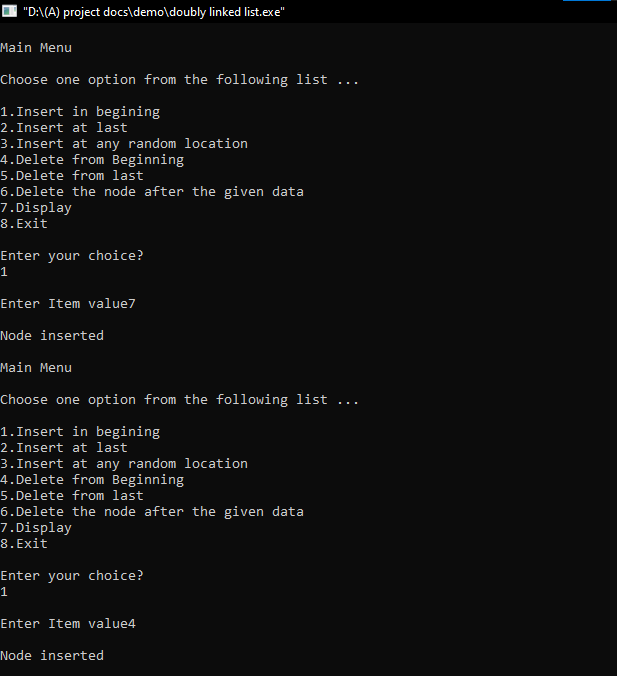
{

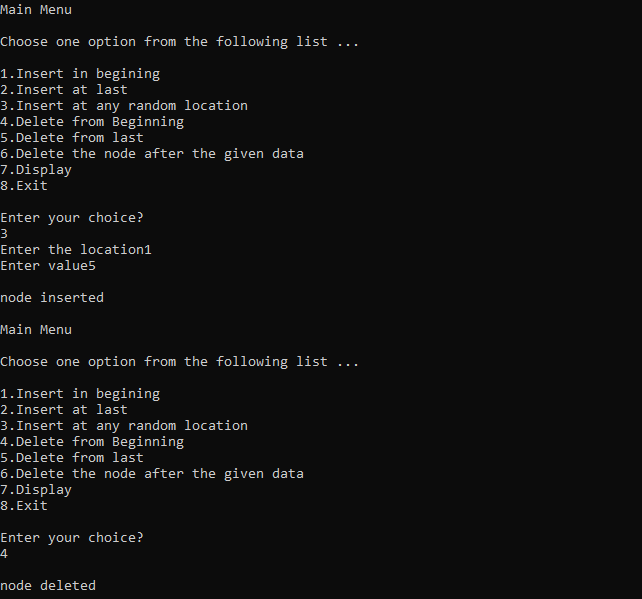
printf("%d\n",newnode->data);

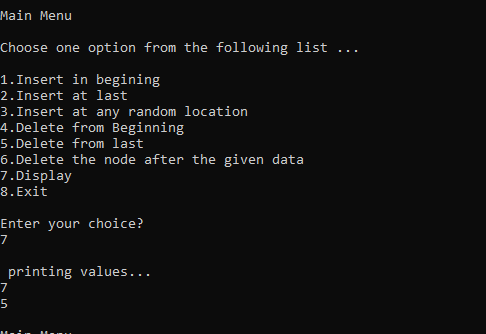
newnode=newnode->next;

}}

**Output Screenshot**







**Result**

The program was executed and the result was successfully obtained. Thus CO1was obtained.

**Experiment No.: 6**

**Aim**

Implementation of Binary search tree.

**CO3**

Understand the practical aspects of Advanced Tree Structures.

**Algorithm**

Insertion Operation in BST

Step 1 - Create a newNode with given value and set its left and right → NULL. Step 2 - Check whether tree is Empty.

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

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

Step 5 - If value is smaller than or equal to the node then node →left = newNode. If newNode is larger than the node then node →right = newNode.

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

Search Operation in BST

Step 1 - Read the search element from the user.

Step 2 - Compare the search element with the root → value.

Step 3 - If root → value == search element "Given node is found!!!" and exit

Step 4 - If root → value != search element, 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.

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 BST

{

int data;

struct BST\*left;

struct BST\*right;

};

typedef struct BST NODE;

NODE\*node;

NODE\*createtree(NODE\*node,int data)

{

if(node==NULL)

{

NODE\*temp;

temp=malloc(sizeof(NODE));

temp->data=data;

temp->left=NULL;

temp->right=NULL;

return temp;

}

if(data<(node->data))

{

node->left=createtree(node->left,data);

}

else if(data>node->data)

{

node->right=createtree(node->right,data);

}

return node;

}

NODE\*search(NODE\*node,int data)

{

if(node==NULL)

{

printf("\n element is not found");

}

else if(data<node->data)

{

node->left=search(node->left,data);

}

else if(data<node->data)

{

node->right=search(node->right,data);

}

else

{

printf("\n element found in %d",node->data);

return node;

}

}

void inorder(NODE\*node)

{

if(node!=NULL)

{

inorder(node->left);

printf("%d\t",node->data);

inorder(node->right);

}

}

void preorder(NODE\*node)

{

if(node!=NULL)

{

printf("%d\t",node->data);

preorder(node->left);

preorder(node->right);

}

}

void postorder(NODE\*node)

{

if(node!=NULL)

{

postorder(node->left);

postorder(node->right);

printf("%d\t",node->data);

}

}

NODE\*findMin(NODE\*node)

{

if(node==NULL)

{

return NULL;

}

if(node->left)

return findMin(node->left);

else

return node;

}

NODE \*del(NODE\*node,int data)

{

NODE\*temp;

if(node==NULL)

{

printf("\n element not found");

}

else if(data<node->data)

{

node->left=del(node->left,data);

}

else if(data>node->data)

{

node->right=del(node->left,data);

}

else

{

if(node->right&&node->left)

{

temp=findMin(node->right);

node->data=temp->data;

node->right=del(node->right,temp->data);

}

else

{

temp=node;

if(node->left==NULL)

{

node=node->right;

}

else if(node->right==NULL)

{

node=node->left;

}

free(temp);

}

}

return node;

}

int main()

{

int data,ch,i,n;

NODE\*root=NULL;

while(1)

{

printf("\n 1.Insertion in Binary Tree");

printf("\n 2.Search element in BST");

printf("\n 3.delete element in BST");

printf("\n 4.Inorder \n 5.preorder \n 6.postorder");

printf("\n 7.exit");

printf("\n enter your choice:");

scanf("%d",&ch);

switch(ch)

{

case 1:

printf("\n enter the value:");

scanf("%d",&n);

printf("\n enter the value to create a BST:");

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

{

scanf("%d",&data);

root=createtree(root,data);

}

break;

case 2:

printf("\n enter the element to search:");

scanf("%d",&data);

root=search(root,data);

break;

case 3:

printf("\n enter the element to delete:");

scanf("%d",&data);

root=del(root,data);

break;

case 4:

printf("Inorder Traversal:");

inorder(root);

break;

case 5:

printf("Preorder Traversal:");

preorder(root);

break;

case 6:

printf("Postorder Traversal:");

postorder(root);

break;

case 7:

exit(0);

default:

printf("\n wrong option");

break;

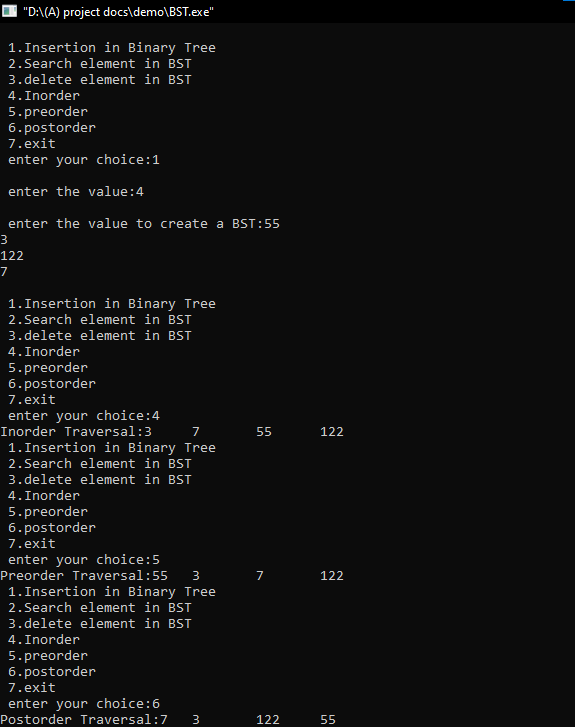
}

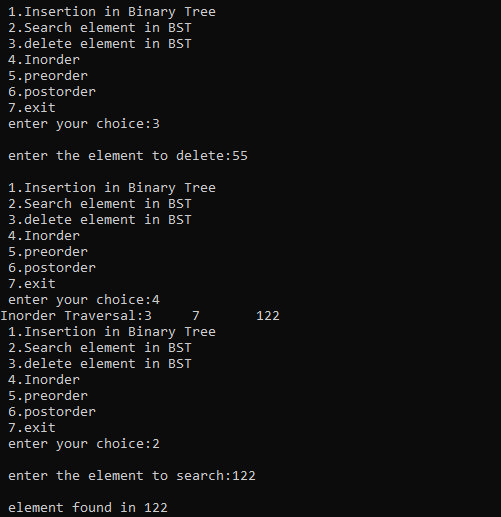
}

return 0;

}

**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 set and disjoint set data structure

**Algorithm**

Step 1. Start

Step 2. Create two character array

Step 3. Enter a bit string in array 1

Step 4. Enter another bit string in array2

Step 5. Display menu of operations

Step 6. If union():

* + For i = 0 to strlen(array): print(aray 1 [i] or array2 2[i] )

Step 7. If intersection():

* + For i = 0 to strlen(array): print(array 1 [i] and array2 2[i] )

Step 8. If set difference():

* + Declare an array3 for complementing array2
  + Store bitwise negation results on array2 in array3
  + For i = 0 to strlen(array): print(array 1[i] or array3 3[i] )

Step 9. Stop

**Procedure**

#include <stdio.h>

#include <stdlib.h>

int a[20],b[20],c[20],x;

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]);

}

}

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]);

}

}

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]);

}

}

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 \n"); 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 \n"); goto main2;

}

}

printf("\n SET MENU \n"); printf("1. Union\t 2.Intersection\n3.Difference\t 0.exit");

while(1){

printf("\n Enter your 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"); break;

case 2: if(m==n)

set\_intersection(a,b,m); else

printf("intersection perform only same size of array\n"); break;

case 3: if(m==n) set\_difference(a,b,m);

else

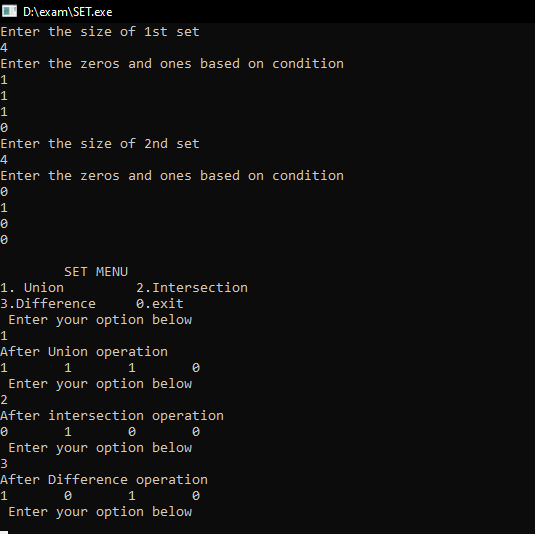
printf("difference perform only same size of array\n");

break; case 0:exit(1);

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

}}}

**Output Screenshot**



**Result**

The program was executed and the result was successfully obtained. Thus CO2was 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.

**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 \*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;

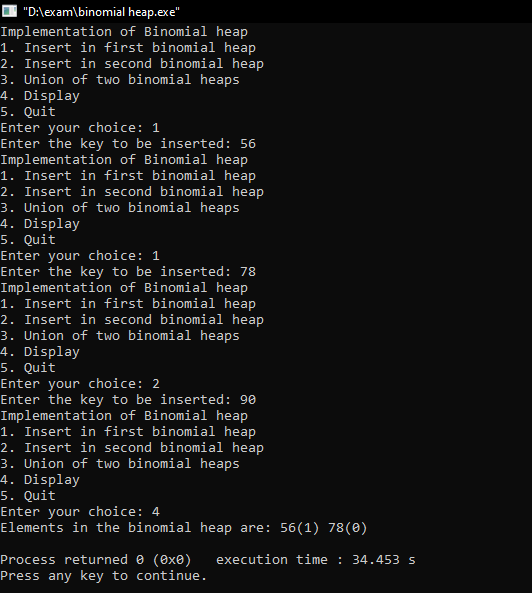
}

}

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

**Procedure**

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#define MAX 6

int vertex\_count =0;

struct vertex

{

char data;

bool visited;

};

struct vertex \*graph[MAX];

int adj\_matrix[MAX][MAX];

int stack[MAX]; int top = -1;

void push(int data)

{

stack[++top]=data;

}

int pop()

{

return stack[top--];

}

int peek()

{

return stack[top];

}

bool is\_stack\_empty()

{

return top == -1;

}

void add\_vertex(char data)

{

struct vertex \*new = (struct vertex\*)malloc(sizeof(struct vertex));

new->data = data;

new->visited = false;

graph[vertex\_count]=new;

vertex\_count++;

}

void add\_edge(int start,int end)

{

adj\_matrix[start][end]=1;

adj\_matrix[end][start]=1;

}

int adj\_vertex(int vertex\_get)

{

int i;

for(i=0;i<vertex\_count;i++)

{

if(adj\_matrix[vertex\_get][i] == 1 && graph[i]->visited == false)

{

return i;

}

}

return -1;

}

void display\_vertex(int pos)

{

printf("%c",graph[pos]->data);

}

void dfs()

{

int i;

int unvisited;

printf("\n---------------------------\n");

graph[0]->visited =true;

display\_vertex(0);

push(0);

while(!is\_stack\_empty())

{

int unvisited = adj\_vertex(peek());

if(unvisited == -1)

{

pop();

}

else

{

graph[unvisited]->visited = true;

display\_vertex(unvisited);

push(unvisited);

}

}

printf("\n---------------------------\n");

for(i=0;i<vertex\_count;i++)

{

graph[i]->visited = false;

}

}

void show()

{

int i;

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

for(i=0;i<vertex\_count;i++)

{

printf("Edge 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();

break;

case 4: exit(0);

break;

default: printf ("\nInvalid option try again !! ...\n");

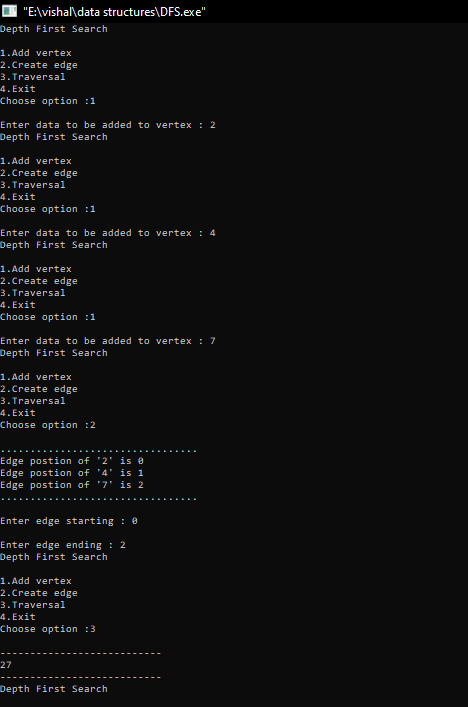
}

}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.

**Procedure**

#include<stdio.h>

#include<stdlib.h>

#include<stdbool.h>

#define MAX 10

int vertex\_count =0;

struct vertex{

char data;

bool visited;

};

struct vertex \*graph[MAX];

int adj\_matrix[MAX][MAX];

int queue[MAX];

int rear=-1;

int front=0;

int queue\_count=0;

void enqueue(int data){

queue[++rear]=data;

queue\_count++;

}

int dequeue(){

queue\_count--;

return queue[front++];

}

bool is\_queue\_empty(){

return queue\_count == 0;

}

void add\_vertex(char data){

struct vertex \*new =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();

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");

for(i=0;i<vertex\_count;i++){

printf("Edge postion of '%c' is %d\n",graph[i]->data,i);

}

printf(".................................");

}

int main(){

int opt;

char data;

int edge\_1,edge\_2;

int i, j;

int start;

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

for(j = 0; j < MAX; j++)

adj\_matrix[i][j] = 0;

do{

printf("1)Add vertex \n2)Create edge \n3)Traversal\n4)Exit \nChoose option : ");

scanf("%d",&opt);

switch(opt){

case 1:printf("Enter data to be added to vertex :");

scanf(" %c", &data);

add\_vertex(data);

break;

case 2: show();

printf("\nEnter edge starting\t: ");

scanf("%d",&edge\_1);

printf("Enter edge ending : \t");

scanf("%d",&edge\_2);

if(vertex\_count-1 < edge\_1 || vertex\_count-1 < edge\_2){

printf("There is no vertex !!\n");

}

else{

add\_edge(edge\_1,edge\_2);

}

break;

case 3: printf("Enter starting vertex position : \t");

scanf("%d",&start);

bfs(graph[start],start);

break;

default:

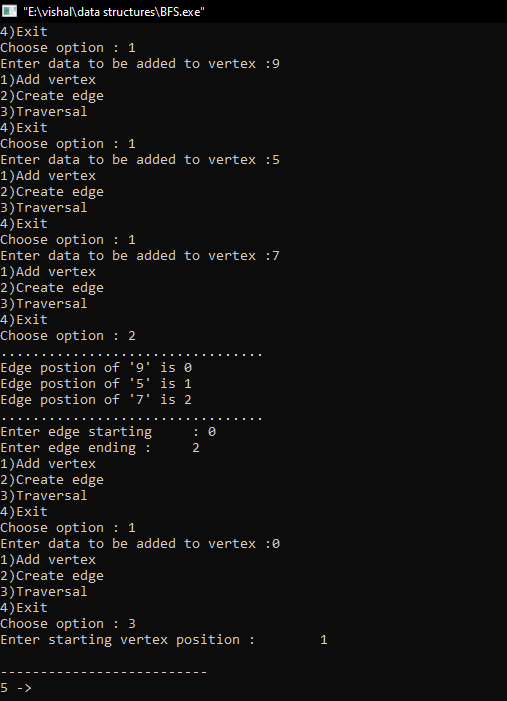
printf("Invalid option try again !! ...");

}

}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)

Step 1:Create a set mstSet that keeps track of vertices already included in MST.

Step 2 :Assign a key value to all vertices in the input graph. Initialize all key values as INFINITE. Assign key value as 0 for the first vertex so that it is picked first.

Step 3: While mstSet doesn't include all vertices

a) Pick a vertex u which is not there in mstSet and has minimum key value.

b) Include u to mstSet.

c) Update key value of all adjacent vertices of u.

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

**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("Implementation of Prims algorithm\n");

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;

}

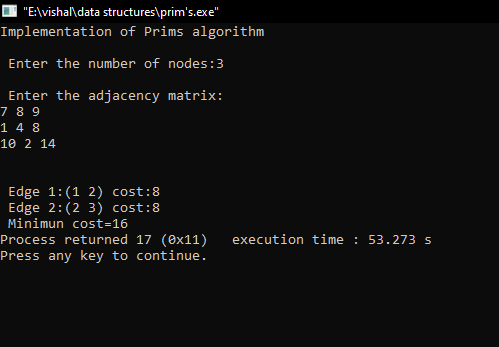
cost[a][b]=cost[b][a]=999;

}

printf("\n Minimun cost=%d",mincost);

}

**Output Screenshot**



**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):

Step1 : Sort all the edges in increasing order of their weight.

Step 2 : Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far.

If cycle is not formed, include this edge. Else, discard it.

Step 3 : Repeat step#2 until there are (V-1) edges in the spanning tree.

**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("Implementation of Prims algorithm\n");

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;

}

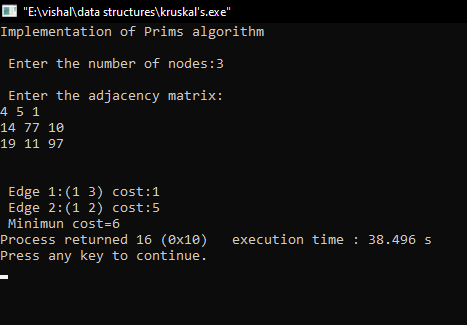
cost[a][b]=cost[b][a]=999;

}

printf("\n Minimun cost=%d",mincost);

}

**Output Screenshot**



**Result**

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