

---

## Program no:01

**Aim:** Familiarization with gdb

Advanced use of gcc:Important options-o,-c,-D,

-1,-l,-g,-O,-save-temps,-pg

Important commands-break,run,next,print,display,help

Using gprof:Compile,Execute and Profile

### 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:

1. Start by updating the packages list:

```
sudo apt-get update
```

2. 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
```

---

- 
- 
3. 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:

1) ***\$ sudo g++ p1.cpp (p1 is the filename)***

*(or)*

2) ***\$ sudo g++ -o p1 p1.cpp***

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

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

*g++ p1.cpp*

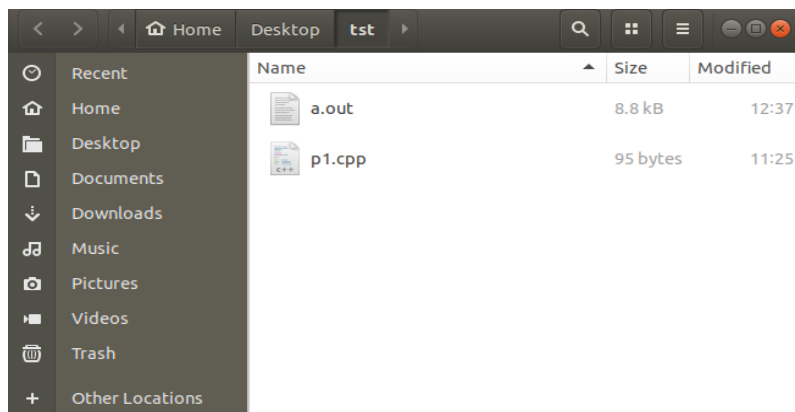
**p1.cpp** in the example is the name of the program to be compiled.

---

---

```
root@S62: ~/Desktop/tst
File Edit View Search Terminal Help
root@S62:~/Desktop/tst# g++ p1.cpp
root@S62:~/Desktop/tst#
```

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



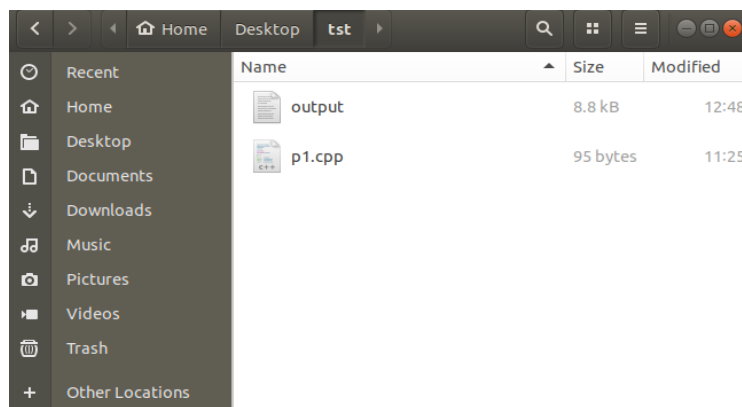
**`$ sudo g++ -o output p1.cpp`**

**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`

```
root@S62: ~/Desktop/tst
File Edit View Search Terminal Help
root@S62:~/Desktop/tst# g++ -o output p1.cpp
root@S62:~/Desktop/tst#
```

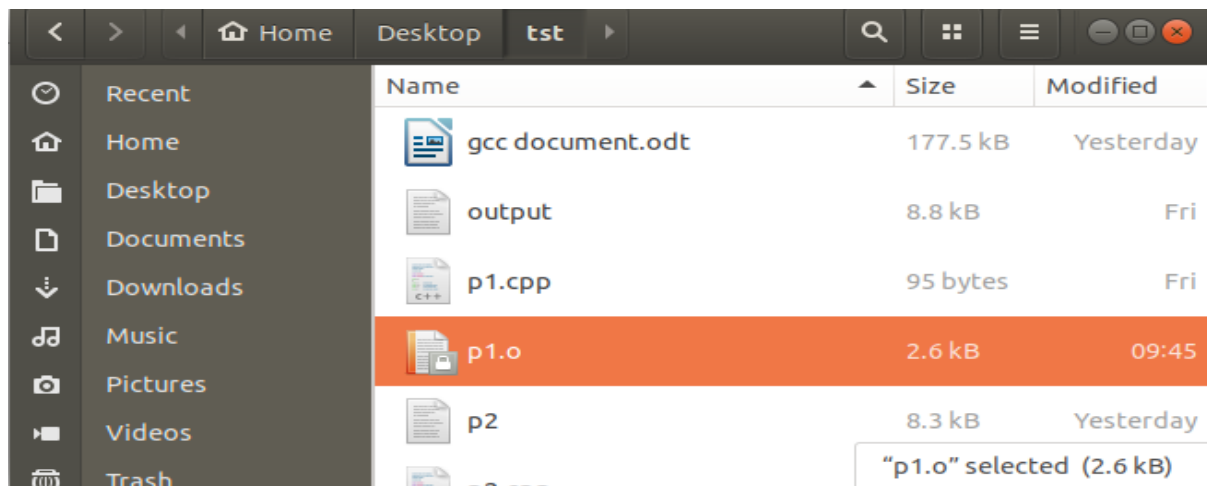
This will compile p1.cpp to the binary file named output, and you can type ./output to run the compiled code.



## 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.

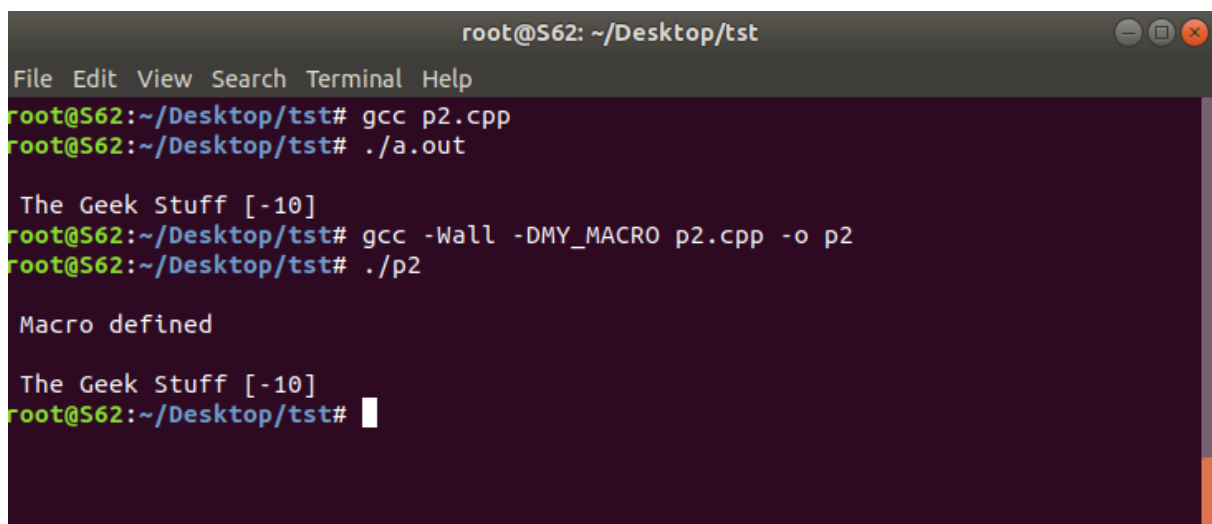
```
$ gcc -Wall -DMY_MACRO main.c -o main
```

```
$ ./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
```

→ *-save-temps*

Through this option, output at all the stages of compilation is stored in the current directory. Please note that this option produces the executable also.

For example :

```
$ gcc -save-temps p2.cpp
$ ls
a.out p2.c p2.i p2.o p2.s
```

---

---

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

→ *-pg*

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

### ***GDB Tutorial***

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

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

### **Compiling**

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

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

### **Invoking and Quitting GDB**

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

*gdb executable*

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

### **Commands**

#### **help**

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

---



---

## **file**

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

## **run**

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

```
run 2048 24 4
```

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

## **break**

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

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

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

## **delete**

delete will delete all breakpoints that you have set.

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

## **clear**

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

## **continue**

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

## **step**

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

---

---

**next**

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

**until**

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

**list**

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

**print**

`print expression` will print out the value of the expression, which could be just a variable name. To print out the first 25 (for example) values in an array called `list`, do `print list[0]@25`

**Gprof**

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

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

***How to use gprof***

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

---

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

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

```
//test_gprof.c
```

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

```
void new_func1(void);
```

```
void func1(void)
```

```
{
```

```
    printf("\n Inside func1 \n");  
    int i = 0;
```

```
    for(;i<0xffffffff;i++);  
    new_func1();
```

```
    return;
```

```
}
```

```
static void func2(void)
```

```
{
```

```
    printf("\n Inside func2 \n");  
    int i = 0;
```

```
    for(;i<0xfffffaa;i++);  
    return;
```

```
}
```

```
int main(void)
```

```
{
```

```
    printf("\n Inside main()\n");  
    int i = 0;
```

```
    for(;i<0xfffff;i++);  
    func1();  
    func2();
```

```
    return 0;
```

---

---

```
}
```

```
//test_gprof_new.c
```

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

```
void new_func1(void)
{
    printf("\n Inside new_func1()\n");
    int i = 0;

    for(;i<0xffffffff;i++);

    return;
}
```

### **Step-1 : Profiling enabled while compilation**

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

lets compile our code with ‘-pg’ option :

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

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

### **Step-2 : Execute the code**

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

```
$ ls
test_gprof test_gprof.c test_gprof_new.c
```

```
$ ./test_gprof
```

```
Inside main()
```

---

---

Inside func1

Inside new\_func1()

Inside func2

\$ ls

gmon.out test\_gprof test\_gprof.c test\_gprof\_new.c

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

### ***Step-3 : Run the gprof tool***

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

\$ gprof test\_gprof gmon.out > analysis.txt

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

\$ ls

analysis.txt gmon.out test\_gprof test\_gprof.c test\_gprof\_new.c

So we see that a file named 'analysis.txt' was generated. As produced above, all the profiling information is now present in 'analysis.txt'. Lets have a look at this text file :

Flat profile:

Each sample counts as 0.01 seconds.

%	cumulative	self		self	total	
time	seconds	seconds	calls	s/call	s/call	name
33.86	15.52	15.52	1	15.52	15.52	func2
33.82	31.02	15.50	1	15.50	15.50	new_func1
33.29	46.27	15.26	1	15.26	30.75	func1

---

---

---

0.07	46.30	0.03		main
------	-------	------	--	------

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

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

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

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

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

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

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

Call graph (explanation follows)

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

index % time self children called name

[1]	100.0	0.03	46.27		main [1]
	15.26	15.50	1/1		func1 [2]
	15.52	0.00	1/1		func2 [3]
-----					
	15.26	15.50	1/1		main [1]
[2]	66.4	15.26	15.50	1	func1 [2]
	15.50	0.00	1/1		new_func1 [4]
-----					
	15.52	0.00	1/1		main [1]
[3]	33.5	15.52	0.00	1	func2 [3]
-----					

---

---

---

	15.50	0.00	1/1	func1 [2]
[4]	33.5	15.50	0.00	1 new_func1 [4]

---

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

Each entry in this table consists of several lines. The line with the index number at the left hand margin lists the current function. The lines above it list the functions that called this function, and the lines below it list the functions this one called.

This line lists:

index A unique number given to each element of the table.

Index numbers are sorted numerically.

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

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

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

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

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

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

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

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

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

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

called This is the number of times this parent called the

---

---

function '/' the total number of times the function was called. Recursive calls to the function are not included in the number after the '/'.

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

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

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

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

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

called This is the number of times the function called this child '/' the total number of times the child was called. Recursive calls by the child are not listed in the number after the '/'.

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

If there are any cycles (circles) in the call graph, there is an entry for the cycle-as-a-whole. This entry shows who called the cycle (as parents) and the members of the cycle (as children.) The '+' recursive calls entry shows the number of function calls that were internal to the cycle, and the calls entry for each member shows, for that member, how many times it was called from other members of the cycle.

Index by function name

[2] func1 [1] main  
[3] func2 [4] new\_func1

---



---

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

1. Flat profile
2. Call graph

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

### **Customize gprof output using flags**

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

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

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

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

#### **2. Suppress verbose blurbs using -b**

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

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

#### **3. Print only flat profile using -p**

In case only flat profile is required then :

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

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

#### **4. Print information related to specific function in flat profile**

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

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

#### **5. Suppress flat profile in output using -P**

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

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

---

---

## **6. Print only call graph information using -q**

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

## **7. Print only specific function information in call graph.**

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

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

## **8. Suppress call graph using -Q**

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

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

---

---

## Program no:02

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

### Algorithm:

1. Start
2. Input the two sorted arrays
3. Find and add the size of the given arrays and declare a third array of the same size.
4. Loop through the index of the third array using for loop:
  - a) Check which array has the smaller element and fill the third array with the same.
  - b) If the first array has the smaller element (i.e.  $\text{arr1}[i] < \text{arr2}[j]$ ) then fill the third array with the element of the first array (i.e.  $\text{arr3}[k++] = \text{arr1}[i++]$ )  
otherwise fill with the element of the second array (i.e.  $\text{arr3}[k++] = \text{arr2}[j++]$ )
5. End loop
6. Output the third array.
7. Stop

### Source Code:

```
#include <stdio.h>

int main()
{
    int n1,n2,n3;

    printf("\nEnter the size of first array ");

    scanf("%d",&n1);

    printf("\nEnter the size of second array ");
```

---

---

```
scanf("%d",&n2);

n3=n1+n2;

printf("\nEnter the sorted array elements");

int a[n1],b[n2],c[n3];

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

{

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

    c[i]=a[i];

}

int k=n1;

printf("\nEnter the sorted array elements");

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

{

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

    c[k]=b[i];

    k++;

}

printf("\nThe merged array..\n");

for(int i=0;i<n3;i++)

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

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

for(int i=0;i<n3;i++)

{

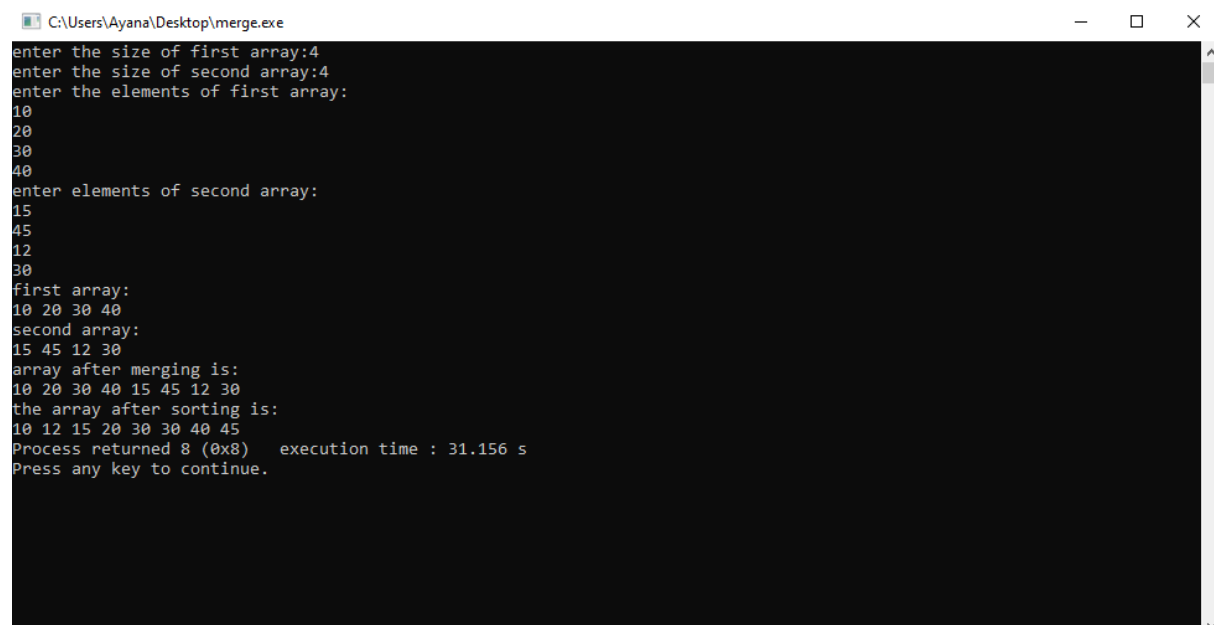
    int temp;

    for(int j=i+1; j<n3 ;j++)
```

---

```
{  
  
    if(c[i]>c[j])  
  
    {  
  
        temp=c[i];  
  
        c[i]=c[j];  
  
        c[j]=temp;  
  
    }  
  
}  
  
}  
  
for(int i=0 ; i<n3 ; i++)  
  
{  
  
    printf(" %d ",c[i]);  
  
}  
  
return 0;  
  
}
```

## Output:



```
C:\Users\Ayana\Desktop\merge.exe  
enter the size of first array:4  
enter the size of second array:4  
enter the elements of first array:  
10  
20  
30  
40  
enter elements of second array:  
15  
45  
12  
30  
first array:  
10 20 30 40  
second array:  
15 45 12 30  
array after merging is:  
10 20 30 40 15 45 12 30  
the array after sorting is:  
10 12 15 20 30 30 40 45  
Process returned 8 (0x8)   execution time : 31.156 s  
Press any key to continue.
```

---

## Program no:03

**Aim:** Singly linked stack

### Algorithm:

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

### Source Code:

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

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

---

---

---

```
struct Node
```

```
{
```

```
    int data;
```

```
    struct Node *next;
```

```
}*top = NULL;
```

```
void push(int);
```

```
void pop();
```

```
void display();
```

```
int main()
```

```
{
```

```
    int choice, value;
```

```
    printf("\n:: Stack using Linked List ::\n");
```

```
    while(1){
```

```
        printf("\n***** MENU *****\n");
```

```
        printf("1. Push\n2. Pop\n3. Display\n4. Exit\n");
```

```
        printf("Enter your choice: ");
```

```
        scanf("%d",&choice);
```

```
        switch(choice){
```

```
            case 1: printf("Enter the value to be insert: ");
```

```
                    scanf("%d", &value);
```

```
                    push(value);
```

```
                    break;
```

```
            case 2: pop(); break;
```

```
            case 3: display(); break;
```

```
            case 4: exit(0);
```

---

---

---

```
        default: printf("\nWrong selection!!! Please try again!!!\n");
    }
}
}

void push(int value)
{
    struct Node *newNode;
    newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    if(top == NULL)
        newNode->next = NULL;
    else
        newNode->next = top;
    top = newNode;
    printf("\nInsertion is Success!!!\n");
}

void pop()
{
    if(top == NULL)
        printf("\nStack is Empty!!!\n");
    else{
        struct Node *temp = top;
        printf("\nDeleted element: %d", temp->data);
        top = temp->next;
        free(temp);
    }
}
```

---



---

```
}  
  
void display()  
{  
    if(top == NULL)  
        printf("\nStack is Empty!!!\n");  
    else{  
        struct Node *temp = top;  
        while(temp->next != NULL){  
            printf("%d--->",temp->data);  
            temp = temp -> next;  
        }  
        printf("%d--->NULL",temp->data);  
    }  
}
```

**Output:**

---

C:\Users\Ayana\Desktop\stacklinkedlist.exe

:: Stack using Linked List ::

\*\*\*\*\* MENU \*\*\*\*\*

1. Push
2. Pop
3. Display
4. Exit

Enter your choice: 1

Enter the value to be insert: 20

Insertion is Success!!!

\*\*\*\*\* MENU \*\*\*\*\*

1. Push
2. Pop
3. Display
4. Exit

Enter your choice: 2

Deleted element: 20

\*\*\*\*\* MENU \*\*\*\*\*

1. Push
2. Pop
3. Display
4. Exit

Enter your choice: 3

Stack is Empty!!!

---

## Program no:04

**Aim:** Circular queue

### Algorithm:

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

### Source Code:

```
#include<stdio.h>

#include<stdlib.h>

void enqueue();

void dequeue();

void display();

struct node

{

int data;

struct node *next;

}*front=NULL;

struct node *rear=NULL;
```

---

---

```
struct node *newnode;

void create_node()

{

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

printf("\n enter a value for the node: ");

scanf("%d",&newnode->data);

}

void main()

{

int opt;

do

{

printf("\n SELECT A CHOICE\n");

printf("\n1. ENQUEUE\n");

printf("\n2. DEQUEUE\n");

printf("\n3. DISPLAY\n");

printf("\n4. EXIT\n");

scanf("%d",&opt);

switch(opt)

{

case 1: enqueue();

        break;

case 2: dequeue();

        break;

case 3: display();
```

---

---

```
        break;

case 4: exit(0);

default: printf("\n Invalid Choice\n");
}
}

while(opt!=4);
}

void enqueue()
{
    create_node();
    if(front==NULL && rear==NULL)
    {
        front=newnode;
        rear=newnode;
        newnode->next=front;
    }
    else
    {
        struct node *temp=front;
        while(temp->next!=front)
            temp=temp->next;
        newnode->next=temp->next;
        temp->next=newnode;
        rear=newnode;
```

---

---

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

---

---

```
}
```

```
display();
```

```
}
```

```
}
```

```
void display()
```

```
{
```

```
if(front==NULL)
```

```
printf("\n the CIRCULAR QUEUE IS empty...!!!\n");
```

```
else
```

```
{
```

```
struct node *temp=front;
```

```
while(temp->next!=front)
```

```
{
```

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

```
temp=temp->next;
```

```
}
```

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

```
}
```

---

## Output:

```
C:\Users\vineeth\Desktop\ADS_LAB\14_Dec_2021\Circular_Queue_USING_LL_14_12_2021.exe
3. DISPLAY
4. EXIT
1
enter a value for the node: 29
12    23    24    25    29
SELECT A CHOICE
1. ENQUEUE
2. DEQUEUE
3. DISPLAY
4. EXIT
1
enter a value for the node: 15
12    23    24    25    29    15
SELECT A CHOICE
1. ENQUEUE
2. DEQUEUE
3. DISPLAY
4. EXIT
```

```
C:\Users\vineeth\Desktop\ADS_LAB\14_Dec_2021\Circular_Queue_USING_LL_14_12_2021.exe
2. DEQUEUE
3. DISPLAY
4. EXIT
3
23    24    25    29    15
SELECT A CHOICE
1. ENQUEUE
2. DEQUEUE
3. DISPLAY
4. EXIT
2
the node 23 has been deleted
24    25    29    15
SELECT A CHOICE
1. ENQUEUE
2. DEQUEUE
3. DISPLAY
4. EXIT
```



---

## Program no:05

**Aim:** Doubly Linked List

### Algorithm:

- 1.START
  - 2.Display a menu of operations
  3. If choice is for insertion
    - 3.1. If beginning
      - 3.1.1.set previous node of newnode to NULL
      - 3.1.2.set next node of newnode to head
      - 3.1.3.set head to point to newnode
    - 3.2. If end
      - 3.2.1.set temp to point to the head (temp=head)
      - 3.2.2.Travel the doubly linked list till the next of temp is null
      - 3.2.3.insert newnode into temp->next
      - 3.2.4.set newnode->next as NULL
      - 3.2.5.set newnode->previous=temp
    - 3.3. If particular position
      - 3.3.1.set temp to point to the first node (head)
      - 3.3.2.travel till the desired position reach
      - 3.3.3.insert the newnode at next to the temp
      - 3.3.4.set previous pointer of newnode to temp
      - 3.3.5.set next pointer of newnode to the current next of temp
      - 3.3.6.set next pointer of temp to newnode
  4. If choice is for deletion
    - 4.1. If beginning
      - 4.1.1.make temp point to first node
      - 4.1.2 .set previous link of next link of temp to be NULL
      - 4.1.3. Set head pointing to the node next to temp;
      - 4.1.4..set NULL at next pointer of temp
    - 4.2. If end
      - 4.2.1.assign temp as newnode
      - 4.2.2.traverse the list till next of temp equal to
      - 4.2.3.set next pointer of previous link of temp to NULL
      - 4.2.4. set previous pointer of temp to NULL, and free temp
    - 4.3. If specific position
      - 4.3.1. Set temp pointing to the first node (head)
-

- 
- 
- 4.3.2. Read a logical position from which the node is to be removed
    - 4.3.3. Remove the desired node by traversing the list
    - 4.3.4. Set the next pointer of node residing before temp as the node after temp
    - 4.3.5. Set previous link of node after temp pointing to node before temp
  5. If choice is for searching an item in the list, traverse the list by checking that whether the item is matching with the data part of current node
    - 5.1. Assign temp = head
    - 5.2. while(temp->data!=item),traverse by temp=temp->next
    - 5.3. If data in temp is equal to item, print ‘ item found in list’
    - 5.4. Otherwise print “ item does not exist”
  6. If operation is for traversal or display
    - 6.1. Set temp as head
    - 6.2. Print the data contained in temp while temp reaches last node
  7. If user’s choice is none other than above , print ‘invalid choice’
  8. Continue steps 2 to 7 till the user input an option for exit
  9. STOP

**Source Code:**

```
#include<stdio.h>

#include<stdlib.h>

struct node {

    int data;

    struct node *prev;

    struct node *next;

}*head = NULL;
```

---

---

---

```
void insertAtFront();

void insertAtEnd();

void insertAtPosition();

void deleteBeg();

void deleteend();

void deleterandom();

void display();

int main()

{

    int choice;

    printf("\n 1.insert beginning\n 2.insert end\n 3.insert specific \n 4.delete front \n ");

    printf("5.delete end \n 6.delete specific \n 7.display\n 8.To exit\n");

    while (choice!=8)

    {

        printf("\nEnter Choice : ");

        scanf("%d", &choice);

        switch (choice)

        {

            case 1:

                insertAtFront();

                break;

            case 2:

                insertAtEnd();

                break;

            case 3:

                insertAtPosition();
```

---

---

```
        break;

    case 4:

        deleteBeg();

        break;

    case 5:

        deleteend();

        break;

    case 6:

        deleterandom();

        break;

    case 7:

        display();

        break;

    case 8:

        exit(1);

        break;

    default:

        printf("Enter a valid choice\n");

    }

}

void insertAtFront()

{

    int value;

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

    printf("\nEnter number to be inserted : ");
```

---

---

```
scanf("%d", &value);

if(head==NULL)

{
    newnode->data = value;
    newnode->next = NULL;
    newnode->prev=NULL;
    head = newnode;
}

else

{
    newnode->data = value;
    head->prev=newnode;
    newnode->next=head;
    newnode->prev=NULL;
    head=newnode;
}

printf("one node was inserted at beginning...\n");
display();
}

void insertAtEnd()

{
    int value;

    struct node *newnode = malloc(sizeof(struct node));
    printf("\nEnter number to be inserted : ");
    scanf("%d", &value);
    newnode->data = value;
```

---

---

```
    struct node *temp;

    temp = head;

    while (temp->next != NULL)

        {

            temp = temp->next;

        }

    newnode->prev=temp;

    temp->next=newnode;

    newnode->next=NULL;

    printf("one node was inserted at ending...\n");

    display();

}

void insertAtPosition()

{

    struct node*temp;

    temp=head;

    int item,pos;

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

    printf("enter the element:");

    scanf("%d",&item);

    printf("enter the location:");

    scanf("%d",&pos);

    newnode->data=item;

    if(head==NULL)

        {

            newnode->next=NULL;
```

---

---

```
newnode->prev=NULL;

head=newnode;

}

else

{

while(temp->data!=pos)

{

if(temp->next==NULL)

{

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

}

else

{

temp=temp->next;

}

}

newnode->prev=temp;

newnode->next=temp->next;

temp->next=newnode;

}

display();

}

void deleteBeg()

{

if(head==NULL)

{
```

---

---

```
    printf("deletion is not possible , linked list is empty");
}
else if(head->next==NULL)
{
    head=NULL;
    printf("one node was deleted");
}
else
{
    head->next->prev=NULL;
    head=head->next;
    printf("one node was deleted");
}
display();
}
void deleteend()
{
    if(head==NULL)
    {
        printf("deletion is not possible , linked list is empty");
    }
    else if(head->next==NULL)
    {
        head=NULL;
        printf("one node was deleted");
    }
}
```

---



---

```
else
{
    struct node *temp=head;
    while(temp->next!=NULL)
    {
        temp=temp->next;
    }
    temp->prev->next=NULL;
    temp->prev=NULL;
    free(temp);
    printf("one node was deleted");
}

display();
}

void deleterandom()
{
    int key;
    if(head==NULL)
    {
        printf("deletion is not possible , linked list is empty");
    }

    struct node *temp=head;
    printf("\n enter the value to delete:");
    scanf("%d",&key);
    if(head->data==key && head->next==NULL)
    {
```

---

---

```
head=NULL;

printf(" node was deleted");

}

else if(head->data==key && head->next!=NULL)

{

head->next->prev=NULL;

head=head->next;

printf("one node was deleted");

}

else

{

while(temp->data!=key && temp->next!=NULL)

{

temp=temp->next;

}

if(temp->next==NULL && temp->data!=key)

{

printf("Deletion not possible");

}

else if(temp->next==NULL && temp->data==key)

{

temp->prev->next=NULL;

temp->prev=NULL;

free(temp);

printf("one node was deleted");

}
```

---

---

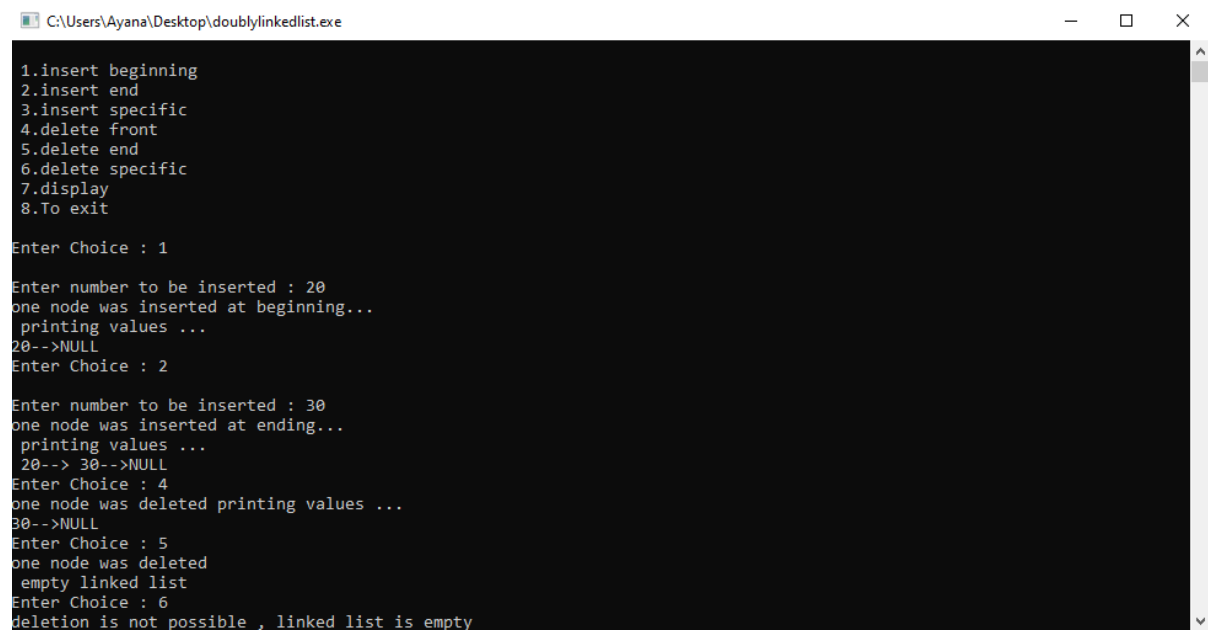
```
else
{
temp->prev->next=temp->next;
temp->next->prev=temp->prev;
free(temp);
printf("one node was deleted");
}
}
display();
}
void display()
{
struct node *temp;
temp=head;
if(head == NULL)
{
printf("\n empty linked list");
}
else
{
printf(" printing values ... \n");
while(temp -> next != NULL)
{
printf(" %d--> ", temp-> data);
temp = temp -> next;
}
}
```

---

---

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

## Output:



```
C:\Users\Ayana\Desktop\doublylinkedlist.exe  
1.insert beginning  
2.insert end  
3.insert specific  
4.delete front  
5.delete end  
6.delete specific  
7.display  
8.To exit  
Enter Choice : 1  
Enter number to be inserted : 20  
one node was inserted at beginning...  
printing values ...  
20-->NULL  
Enter Choice : 2  
Enter number to be inserted : 30  
one node was inserted at ending...  
printing values ...  
20--> 30-->NULL  
Enter Choice : 4  
one node was deleted printing values ...  
30-->NULL  
Enter Choice : 5  
one node was deleted  
empty linked list  
Enter Choice : 6  
deletion is not possible , linked list is empty
```

---

## Program no:06

**Aim:** Binary Search Tree

### Algorithm:

1. Start
  2. Define a structure for BST
  3. Display a menu of operations
  4. If inorder traversal
    - 4.1. Visit the left child
    - 4.2. Process the node currently accessed
    - 4.3. Visit the right child
  5. If preorder traversal
    - 5.1. Process the node currently accessed
    - 5.2. Visit the left child
    - 5.3. Visit the right child
  6. If postorder traversal
    - 6.1. Visit the left child
    - 6.2. Visit the right child
    - 6.3. Process the currently visited node
  7. If insertion operation
    - 7.1. Read a value in key
    - 7.2. If root is NULL, insert new node as root
    - 7.3. Else check if key less than root node
      - 7.3.1. Insert the newnode at left of root node
      - 7.3.2. Else insert newnode at the right of root node
  8. If search operation
    - 8.1. Read an item to be searched in item
    - 8.2. Check if item lesser or greater than the root
    - 8.3. If item lesser than root node value
      - 8.3.1. Perform recursive search on the left subtree
      - 8.3.2. Else perform recursive search on the right subtree
  9. If deletion operation
    - 9.1. Read a key to be deleted from the bst
    - 9.2. If key is lesser than the root node's value
      - 9.2.1. If the element to be deleted is parent node
        - 9.2.1.1. Replace it with inorder successor
        - 9.2.1.2. Else replace it with inorder predecessor
      - 9.2.2. If element to be deleted is leaf node, simply delete key
      - 9.2.3. Stop
-

---

**Source Code:**

```
#include <stdio.h>

#include <stdlib.h>

struct Node
{
    struct node *left;

    int data;

    struct node *right;
} *root=NULL;

void traversal();

struct Node *insertion(struct Node *p,int key)
{
    struct Node *t=NULL;

    if(p==NULL)
    {
        t=(struct Node *)malloc(sizeof(struct Node));

        t->data=key;

        t->left=t->right=NULL;

        return t;
    }

    if(key < p->data)
        p->left=insertion(p->left,key);

    else if(key > p->data)
        p->right=insertion(p->right,key);

    return p;

    traversal(root);
```

---

---

```
}

struct Node * Search(int key)
{
    struct Node *temp=root;
    while(temp!=NULL)
    {
        if(key==temp->data)
            return temp;
        else if(key<temp->data)
            temp=temp->left;
        else
            temp=temp->right;
    }
    return NULL;
}

void traversal(struct Node *root)
{
    if (root != NULL) {
        traversal(root->left);
        printf("%d \n", root->data);
        traversal(root->right);
    }
}

int Height(struct Node *p)
{
    int x,y;
```

---

---

```
if(p==NULL)return 0;

    x=Height(p->left);

    y=Height(p->right);

    return x>y?x+1:y+1;

}

struct Node *InPre(struct Node *p)

{

    while(p && p->right!=NULL)

        p=p->right;

    return p;

}

struct Node *InSucc(struct Node *p)

{

    while(p && p->left!=NULL)

        p=p->left;

    return p;

}

struct Node *Delete(struct Node *p,int key)

{

    struct Node *q;

    if(p==NULL)

        return NULL;

    if(p->left==NULL && p->right==NULL)

    {

        if(p==root)

            root=NULL;
```

---



---

```
        free(p);

        return NULL;

    }

    if(key < p->data)

        p->left=Delete(p->left,key);

    else if(key > p->data)

        p->right=Delete(p->right,key);

    else

    {

        if(Height(p->left)>Height(p->right))

        {

            q=InPre(p->left);

            p->data=q->data;

            p->left=Delete(p->left,q->data);

        }

        else

        {

            q=InSucc(p->right);

            p->data=q->data;

            p->right=Delete(p->right,q->data);

        }

    }

    return p;

    traversal(root);

}

int main()
```

---

---

```
{  
  
    int key,a;  
  
    struct Node *temp;  
  
    while(1){  
  
        printf("----Binary Tree Menu----\n");  
  
        printf(" 1. Insertion\n 2. Deletion\n 3. Traversal\n 4. Searching\n 5. Exit\n");  
  
        printf("\n Enter your choice:");  
  
        scanf("%d",&a);  
  
        switch(a){  
  
            case 1: printf("Enter the data : ");  
  
                scanf("%d",&key);  
  
                root=insertion(root,key);  
  
                break;  
  
            case 2: printf("Enter the element you want to delete\n");  
  
                scanf("%d",&key);  
  
                Delete(root,key);  
  
                break;  
  
            case 3: traversal(root);  
  
                break;  
  
            case 4: printf("Enter the element to search\n");  
  
                scanf("%d",&key);  
  
                temp=Search(key);  
  
                if(temp!=NULL)  
  
                    {  
  
                        printf("element %d is found\n",temp->data);  
  
                    }  
  
                }  
  
        }  
  
    }  
}
```

---

---

```
        else
        {
            printf("element is not found\n");
        }

        break;

    case 0: exit(1);

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

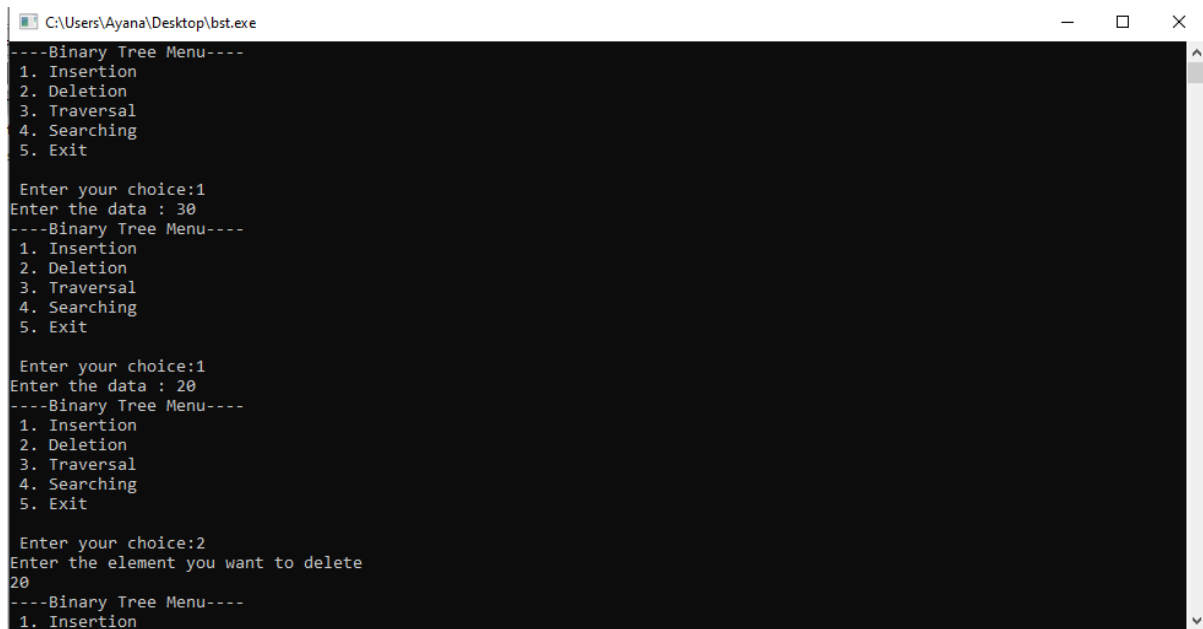
        break;

    }

}

}
```

## Output:



```
C:\Users\Ayana\Desktop\bst.exe
---Binary Tree Menu---
1. Insertion
2. Deletion
3. Traversal
4. Searching
5. Exit

Enter your choice:1
Enter the data : 30
---Binary Tree Menu---
1. Insertion
2. Deletion
3. Traversal
4. Searching
5. Exit

Enter your choice:1
Enter the data : 20
---Binary Tree Menu---
1. Insertion
2. Deletion
3. Traversal
4. Searching
5. Exit

Enter your choice:2
Enter the element you want to delete
20
---Binary Tree Menu---
1. Insertion
```

```
C:\Users\Ayana\Desktop\bst.exe
1. Insertion
2. Deletion
3. Traversal
4. Searching
5. Exit

Enter your choice:3
30
----Binary Tree Menu----
1. Insertion
2. Deletion
3. Traversal
4. Searching
5. Exit

Enter your choice:4
Enter the element to search
30
element 30 is found
----Binary Tree Menu----
1. Insertion
2. Deletion
3. Traversal
4. Searching
5. Exit

Enter your choice:
```

---

## Program no:07

**Aim:** Set data structure and set operations using Bit Strings

### Algorithm:

1. Start
2. Read the no: of elements in first set
3. Read the elements of first set
4. Read the no: of elements in second set
5. Read the elements of second set
6. Check  $k=0$  display resultant set is null
7. Otherwise display element of resultant set
8. Stop

### Source Code:

```
#include<stdio.h>

#include<stdlib.h>

void Union();
void intersection();
void difference();
void equal();
int main()
{
    int ch;

    printf("\n\t\t\t--MENU--\n");
```

---

---

```
printf("\t\t\t 1.union\n\t\t\t 2.Intersection\n\t\t\t 3.difference \n\t\t\t 4.equal \n\t\t\t 5.Exit\n");
```

```
while(ch=1)
```

```
{
```

```
printf("\nEnter your choice : ");
```

```
scanf("%d",&ch);
```

```
switch(ch)
```

```
{
```

```
case 1:
```

```
Union();
```

```
break;
```

```
case 2:
```

```
intersection();
```

```
break;
```

```
case 3:
```

```
difference();
```

```
break;
```

```
case 4:
```

```
equal();
```

```
break;
```

```
case 5:
```

```
exit(0);
```

```
break;
```

```
default:
```

```
printf("Enter a valid operation");
```

```
break;
```

---

---

```
}  
  
}  
  
}  
void Union()  
{  
    int li,lj,si[20],sj[20],s[50],i;  
    printf("Enter the size of first set:");  
    scanf("%d",&li);  
    printf("Enter the size of second set:");  
    scanf("%d",&lj);  
    printf("Enter the 1st set elements:");  
    for(i=0;i<li;i++)  
    {  
        scanf("%d",&si[i]);  
    }  
    printf("Enter the 2nd set elements:");  
    for(i=0;i<lj;i++)  
    {  
        scanf("%d",&sj[i]);  
    }  
    if(li!=lj)  
    {  
        printf("The cardinality of li != to cardinality of lj");  
        printf("Union not possible");  
    }  
    else
```

---

---

```
{  
    printf("Union = ");  
    for(i=0;i<li;i++)  
    {  
        s[i]=si[i] || sj[i];  
    }  
    for(i=0;i<li;i++)  
    {  
        printf("%d ",s[i]);  
    }  
}  
}  
  
void intersection()  
{  
    int li,lj,si[20],sj[20],s[50],i;  
    printf("Enter the size of first set:");  
    scanf("%d",&li);  
    printf("Enter the size of second set:");  
    scanf("%d",&lj);  
    printf("Enter the elements:");  
    for(i=0;i<li;i++)  
    {  
        scanf("%d",&si[i]);  
    }  
    printf("Enter the elements:");  
    for(i=0;i<lj;i++)
```

---



---

```
{
    scanf("%d",&sj[i]);
}
if(li!=lj)
{
    printf("The cardinality of li != to cardinality of lj");
    printf("Interseccion not possible");
}
else
{
    printf("Intersection = ");
    for(i=0;i<li;i++)
    {
        s[i]=si[i] && sj[i];
    }
    for(i=0;i<li;i++)
    {
        printf("%d",s[i]);
    }
}
}

void difference()
{
    int li,lj,si[20],sj[20],s[50],i;

    printf("Enter the size of first set:");

    scanf("%d",&li);
```

---

---

```
printf("Enter the size of second set:");

scanf("%d",&lj);

printf("Enter the elements:");

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

{

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

}

printf("Enter the elements:");

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

{

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

}

if(li!=lj)

{

    printf("The cardinality of li != to cardinality of lj");

    printf("Difference not possible");

}

else

{

    printf("Difference = ");

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

    {

        s[i]=si[i] - sj[i];

    }

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

    {
```

---

---

```
        printf("%d",s[i]);

    }

}

}

void equal()
{
    int li,lj,si[20],sj[20],i,x=0;

    printf("Enter the size of first set:");

    scanf("%d",&li);

    printf("Enter the size of second set:");

    scanf("%d",&lj);

    printf("Enter the elements:");

    for(i=0;i<li;i++)
    {
        scanf("%d",&si[i]);
    }

    printf("Enter the elements:");

    for(i=0;i<lj;i++)
    {
        scanf("%d",&sj[i]);
    }

    if(li!=lj)
    {
        printf("The cardinality of li != to cardinality of lj");

        printf("Equality not possible");

    }
}
```

---

---

```
else
{
for(i=0;i<li;i++)
{
    if(si[i]!=sj[i])
    {
        x++;
    }
}
if(x>0)
{
    printf("not equal set");
}
else
{
    printf("Equal set");
}
}
```

---

## Output:

```
C:\Users\Ayana\Desktop\bitvector.exe

--MENU--
1.union
2.Intersection
3.difference
4.equal
5.Exit

Enter your choice : 1
Enter the size of first set:4
Enter the size of second set:4
Enter the 1st set elements:4
5
2
1
Enter the 2nd set elements:5
2
3
1
Union = 1 1 1 1
Enter your choice : 2
Enter the size of first set:4
Enter the size of second set:4
Enter the elements:4
5
2
1
Enter the elements:5
2
3
```

```
C:\Users\Ayana\Desktop\bitvector.exe

Enter the elements:5
2
3
1
Intersection = 1111
Enter your choice : 3
Enter the size of first set:4
Enter the size of second set:4
Enter the elements:4
5
6
1
Enter the elements:2
3
1
4
Difference = 225-3
Enter your choice :
4
Enter the size of first set:4
Enter the size of second set:4
Enter the elements:5
2
1
2
Enter the elements:4
5
5
1
not equal set
```

---

## Program no:08

**Aim:** Graph Traversal techniques(DFS and BFS) and Topological Sorting

### Algorithm:

#### a)BFS:

1. Create a queue Q
2. Mark v as visited and put v into Q
3. while Q is non-empty
4. remove the head u of Q
5. mark and enqueue all (unvisited) neighbours of u

#### (b)DFS:

DFS(G, u)

    u.visited = true

    for each v  $\in$  G.Adj[u]

        if v.visited == false

            DFS(G,v)

init() {

    For each u  $\in$  G

        u.visited = false

    For each u  $\in$  G

        DFS(G, u)

}

#### (c)TOPOLOGICAL SORTING:

1. For each vertex  $U \in V$
-

- 
2. do indegree [U]<-0
  3. for each vertex  $U \in V$
  4. do for each  $V \in \text{Adj}[U]$
  5. do indegree[V]<-indegree[V]+1
  6.  $Q \leftarrow \emptyset$
  7. For each vertex  $U \in V$
  8. Do if indegree [U]=0
  9. Then EnQueue (Q,U)
  10. While  $Q \neq \emptyset$
  11. Do  $U \leftarrow \text{DEQUEUE}(Q)$
  12. Output U
  13. For each  $V \in \text{Adj} [U]$
  14. Do indegree [V]<- indegree[V]-1
  15. If indegree[V] = 0
  16. Then EnQueue(Q,V)
  17. Do if indegree[U]≠0
  18. Repeat there is a cycle.

### Source Code:

#### (a)BFS:

```
#include<stdio.h>

int a[20][20], q[20], visited[20], n, i, j, f = 0, r = -1;

void bfs(int v)
{
    for(i = 1; i <= n; i++)
        if(a[v][i] && !visited[i])
            q[++r] = i;
```

---

---

```
if(f <= r) {  
    visited[q[f]] = 1;  
    bfs(q[f++]);  
}  
}  
  
int main()  
{  
    int v;  
    printf("\n Enter the number of vertices:");  
    scanf("%d", &n);  
    for(i=1; i <= n; i++)  
    {  
        q[i] = 0;  
        visited[i] = 0;  
    }  
    printf("\n Enter graph data in matrix form:\n");  
    for(i=1; i<=n; i++)  
    {  
        for(j=1;j<=n;j++)  
        {  
            scanf("%d", &a[i][j]);  
        }  
    }  
    printf("\n Enter the starting vertex:");  
    scanf("%d", &v);
```

---



---

```
bfs(v);

printf("\n The node which are reachable are:\n");

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

{

if(visited[i])

printf("%d\t", i);

else

{

printf("\n Bfs is not possible. Not all nodes are reachable");

break;

}

}

}
```

**(b)DFS:**

```
#include<stdio.h>

int a[20][20],reach[20],n;

int 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);

        }

}
```

---

---

```
    }  
  
}  
  
int 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");  
  
return 0;  
  
}
```

### **(c) TOPOLOGICAL SORTING:**

```
#include <stdio.h>  
  
int main(){  
  
    int i,j,k,n,a[10][10],indeg[10],flag[10],count=0;  
  
    printf("Enter the no of vertices:\n");  
  
    scanf("%d",&n);  
  
    printf("Enter the adjacency matrix:\n");  
  
    for(i=0;i<n;i++){  
  
        printf("Enter row %d\n",i+1);  
  
        for(j=0;j<n;j++){  
  
            scanf("%d",&a[i][j]);  
  
        }  
  
        for(i=0;i<n;i++){  
  
indeg[i]=0;  
  
flag[i]=0;  
  
}  
  
for(i=0;i<n;i++){  
  
for(j=0;j<n;j++){  
  
indeg[i]=indeg[i]+a[j][i];  
  
printf("\nThe topological order is:");  
  
while(count<n){
```

---

---

```
for(k=0;k<n;k++){  
    if((indeg[k]==0) && (flag[k]==0)){  
        printf("%d ",(k+1));  
        flag [k]=1;  
    }  
    for(i=0;i<n;i++){  
        if(a[i][k]==1)  
            indeg[k]--;  
    }  
}  
count++;  
}  
Return  
}
```

## Output:

### BFS

```
Enter the number of vertices:3  
  
Enter graph data in matrix form:  
2 8 7  
5 4 9  
2 1 8  
  
Enter the starting vertex:1  
  
The node which are reachable are:  
1      2      3  
Process returned 3 (0x3)   execution time : 24.899 s  
Press any key to continue.
```

---

---

## DFS

```
Enter number of vertices:4

Enter the adjacency matrix:
12 18 11
77 13 25
44 52 62
7 10 34
6 15 80
5 32 16

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

Graph is connected
Process returned 0 (0x0)   execution time : 29.118 s
Press any key to continue.
```

## TOPOLOGICAL SORTING

```
Enter the no of vertices:
4
Enter the adjacency matrix:
Enter row 1
0 1 1 0
Enter row 2
0 1 0 0
Enter row 3
0 1 0 1
Enter row 4
0 0 1 1

The topological order is:1 2 3 4
Process returned 4 (0x4)   execution time : 61.440 s
Press any key to continue.
```

---

---

**Program no:09**

**Aim:** Prim's Algorithm for finding the minimum cost spanning tree  
Prim's Algorithm for finding the minimum cost spanning tree

**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  $\Pi[V] \leftarrow u$

$Key[V] \leftarrow w[u, v]$

**Source Code:**

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

```
#include<stdbool.h>
```

```
#define INF 9999999
```

```
#define V 5
```

```
int G[V][V] = {
```

```
{0, 9, 75, 0, 0},
```

```
{9, 0, 95, 19, 42},
```

```
{75, 95, 0, 51, 66},
```

---

```
{0, 19, 51, 0, 31},
{0, 42, 66, 31, 0}};

int main()
{
    int no_edge; // number of edge
    int selected[V];
    memset(selected, false, sizeof(selected));
    no_edge = 0;
    selected[0] = true;

    int x;
    int y;

    printf("Edge : Weight\n");
    while (no_edge < V - 1)
    {
        int min = INF;
        x = 0;
        y = 0;
        for (int i = 0; i < V; i++)
        {
            if (selected[i])
            {
                for (int j = 0; j < V; j++)
                {
                    if (!selected[j] && G[i][j])
                    {
```

---

```
if (min > G[i][j]) {  
    min = G[i][j];  
    x = i;  
    y = j;  
}  
}  
}  
}  
}  
}  
}  
printf("%d - %d : %d\n", x, y, G[x][y]);  
selected[y] = true;  
no_edge++;  
}  
return 0;  
}
```

### Output:

```
C:\Users\vineeth\Desktop\prims.exe  
Edge : Weight  
0 - 1 : 9  
1 - 3 : 19  
3 - 4 : 31  
3 - 2 : 51  
  
Process returned 0 (0x0)   execution time : 0.035 s  
Press any key to continue.
```



---

**Program no:10**

**Aim:** Kruskal's Algorithm

**Algorithm:**

**KRUSKAL(G):**

$A = \emptyset$

For each vertex  $v \in G.V$ :

MAKE-SET( $v$ )

For each edge  $(u, v) \in G.E$  ordered by increasing order by weight( $u, v$ ):

if FIND-SET( $u$ )  $\neq$  FIND-SET( $v$ ):

$A = A \cup \{(u, v)\}$

UNION( $u, v$ )

return  $A$

**Source Code:**

```
#include <stdio.h>

#define MAX 30

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

typedef struct edge_list
{
    edge data[MAX];
```

---

```
int n;

}

edge_list;

edge_list elist;

int Graph[MAX][MAX], n;

edge_list spanlist;

void kruskalAlgo();

int find(int belongs[], int vertexno);

void applyUnion(int belongs[], int c1, int c2);

void sort();

void print();

void kruskalAlgo()
{
    int belongs[MAX], i, j, cno1, cno2;

    elist.n = 0;

    for (i = 1; i < n; i++)
        for (j = 0; j < i; j++)
        {
            if (Graph[i][j] != 0)
            {
                elist.data[elist.n].u = i;

                elist.data[elist.n].v = j;

                elist.data[elist.n].w = Graph[i][j];

                elist.n++;
            }
        }
    }
```

---

```
}

sort();

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

    belongs[i] = i;

spanlist.n = 0;

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

{

    cno1 = find(belongs, elist.data[i].u);

    cno2 = find(belongs, elist.data[i].v);

    if (cno1 != cno2)

    {

        spanlist.data[spanlist.n] = elist.data[i];

        spanlist.n = spanlist.n + 1;

        applyUnion(belongs, cno1, cno2);

    }

}

int find(int belongs[], int vertexno)

{

    return (belongs[vertexno]);

}

void applyUnion(int belongs[], int c1, int c2) {

    int i;

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

        if (belongs[i] == c2)
```

---

```
belongs[i] = c1;

}

void sort() {

int i, j;

edge temp;

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

for (j = 0; j < elist.n - 1; j++)

if (elist.data[j].w > elist.data[j + 1].w) {

temp = elist.data[j];

elist.data[j] = elist.data[j + 1];

elist.data[j + 1] = temp;

}

}


void print()

{

int i, cost = 0;

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

{

printf("\n%d - %d : %d", spanlist.data[i].u, spanlist.data[i].v, spanlist.data[i].w);

cost = cost + spanlist.data[i].w;

}

printf("\nSpanning tree cost: %d", cost);

}
```

---

---

```
int main()

{

int i, j, total_cost;

n = 6;

Graph[0][0] = 0;

Graph[0][1] = 4;

Graph[0][2] = 4;

Graph[0][3] = 0;

Graph[0][4] = 0;

Graph[0][5] = 0;

Graph[0][6] = 0;

Graph[1][0] = 4;

Graph[1][1] = 0;

Graph[1][2] = 2;

Graph[1][3] = 0;

Graph[1][4] = 0;

Graph[1][5] = 0;

Graph[1][6] = 0;

Graph[2][0] = 4;

Graph[2][1] = 2;

Graph[2][2] = 0;

Graph[2][3] = 3;

Graph[2][4] = 4;

Graph[2][5] = 0;

Graph[2][6] = 0;
```

---

---

```
Graph[3][0] = 0;

Graph[3][1] = 0;

Graph[3][2] = 3;

Graph[3][3] = 0;

Graph[3][4] = 3;

Graph[3][5] = 0;

Graph[3][6] = 0;

Graph[4][0] = 0;

Graph[4][1] = 0;

Graph[4][2] = 4;

Graph[4][3] = 3;

Graph[4][4] = 0;

Graph[4][5] = 0;

Graph[4][6] = 0;

Graph[5][0] = 0;

Graph[5][1] = 0;

Graph[5][2] = 2;

Graph[5][3] = 0;

Graph[5][4] = 3;

Graph[5][5] = 0;

Graph[5][6] = 0;

kruskalAlgo();

print();

}
```

---

**Output:**

```
2 - 1 : 2
5 - 2 : 2
3 - 2 : 3
4 - 3 : 3
1 - 0 : 4
Spanning tree cost: 14
Process returned 23 (0x17)   execution time : 0.036 s
Press any key to continue.
```