AMIRAJ

COLLEGE OF ENGINEERING & TECHNOLOGY

LABORATORY MANUAL

DATA STRUCTURE

SUBJECT CODE: 3130702

COMPUTER SCIENCE AND ENGINEERING DEPARTMENT

B.E. 3rd SEMESTER

NAME:	
ENROLLMENT NO:	
BATCH NO:	
YEAR:	

Amiraj College of Engineering and Technology,

Nr.Tata Nano Plant, Khoraj, Sanand, Ahmedabad.

AMIRAJ
COLLEGE OF ENGINEERING & TECHNOLOGY

Amiraj College of Engineering and Technology,

Nr.Tata Nano Plant, Khoraj, Sanand, Ahmedabad.

CERTIFICATE

This is to certify that Mr. / Ms. ____ Of class____________________ Enrolment No ___________________________has Satisfactorily completed the course in ____________________________________as by the Gujarat Technological University for ____ Year (B.E.) semester___ of Computer Science and Engineering in the Academic year__________.

Date of Submission:-

Faculty Name and Signature
Subject Teacher

Head of Department
Computer

AMIRAJ
COLLEGE OF ENGINEERING & TECHNOLOGY

COMPUTER SCIENCE AND ENGINEERING DEPARTMENT

B.E. 3RD SEMESTER

SUBJECT: DATA STRUCTURE

SUBJECT CODE: 3130702

List Of Experiments

S. NO.	Title	Date of Performance	Date of submission	Sign	Remark
1.	Introduction to pointers. Call by Value and Call by reference.				
2.	Implement a program for stack that performs following operations using array. (a) PUSH (b) POP (C) EXIT (D) DISPLAY				
3.	Write a program to implement QUEUE using arrays that performs following operations (a) INSERT (b) DELETE (c) DISPLAY				
4.	Write a program to implement				

	Circular Queue using arrays that performs following operations. (a) INSERT (b) DELETE (c) DISPLAY				
5.	"Write a menu driven program to implement following operations on the singly linked list. " (a) Insert a node at the front of the linked list. (b) Insert a node at the end of the linked list. (c) Insert a node at specific place. (d) Delete a first node of the linked list. (e) Delete a specified node. (f) Delete a last node of the linked list.				
6.	"Write a program to implement following operations on the doubly linked list. " (a) Insert a node at the front of the linked list. (b) Insert a node at the end of the linked list. (c) Delete a middle node of the linked list.				
7.	Write a program which create binary search tree. Implement recursive and non-recursive tree traversing methods inorder, preorder and post-order traversal.				

8.	Write a program to implement Quick Sort Write a program to implement Merge Sort				
9.	Write a program to implement Bubble Sort				
10.	Write a program to implement Binary Search.				

PRACTICAL – 1

OBJECTIVE : Introduction to pointers. Call by Value and Call by reference.

Introduction to Dynamic Memory Allocation

What are Pointers?

A **pointer** is a variable whose value is the address of another variable, i.e., direct address of the memory location. Like any variable or constant, you must declare a pointer before using it to store any variable address. The general form of a pointer variable declaration is −

```
type *var-name;
```

Here, **type** is the pointer's base type; it must be a valid C data type and **var-name** is the name of the pointer variable. The asterisk * used to declare a pointer is the same asterisk used for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer. Take a look at some of the valid pointer declarations −

```
int    *ip;    /* pointer to an integer */
double *dp;    /* pointer to a double */
float  *fp;    /* pointer to a float */
char   *ch     /* pointer to a character */
```

The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

How to Use Pointers?

There are a few important operations, which we will do with the help of pointers very frequently. **(a)** We define a pointer variable, **(b)** assign the address of a variable to a pointer and **(c)** finally access the value at the address available in the pointer variable. This is done by using unary operator ***** that returns the value of the variable located at the address specified by its operand. The following example makes use of these operations −

```
#include<stdio.h>


int main (){

intvar=20;/* actual variable declaration */

int*ip;/* pointer variable declaration */

   ip =&var;/* store address of var in pointer variable*/

   printf("Address of var variable: %x\n",&var);

   printf("Address stored in ip variable: %x\n", ip );

   printf("Value of *ip variable: %d\n",*ip );

return0;

}
```

When the above code is compiled and executed, it produces the following result −

```
Address of var variable: bffd8b3c
Address stored in ip variable: bffd8b3c
Value of *ip variable: 20
```

NULL Pointers

It is always a good practice to assign a NULL value to a pointer variable in case you do not have an exact address to be assigned. This is done at the time of variable declaration. A pointer that is assigned NULL is called a **null** pointer.

The NULL pointer is a constant with a value of zero defined in several standard libraries. Consider the following program −

```
#include<stdio.h>

int main ()

{

int*ptr = NULL;

   printf("The value of ptr is : %x\n", ptr  );

return0;

}
```

When the above code is compiled and executed, it produces the following result −

```
The value of ptr is 0
```

In most of the operating systems, programs are not permitted to access memory at address 0 because that memory is reserved by the operating system. However, the memory address 0 has special significance; it signals that the pointer is not intended to point to an accessible memory location. But by convention, if a pointer contains the null (zero) value, it is assumed to point to nothing.

To check for a null pointer, you can use an 'if' statement as follows −

```
if(ptr)     /* succeeds if p is not null */
if(!ptr)    /* succeeds if p is null */
```

The **call by value method** of passing arguments to a function copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.

By default, C programming uses *call by value* to pass arguments. In general, it means the code within a function cannot alter the arguments used to call the function.

The **call by reference method** of passing arguments to a function copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. It means the changes made to the parameter affect the passed argument.

To pass a value by reference, argument pointers are passed to the functions just like any other value. So accordingly you need to declare the function parameters as pointer types as in the following function **swap()**, which exchanges the values of the two integer variables pointed to, by their arguments.

Dynamic Memory Allocation

Although C inherently does not have any technique to allocate memory dynamically, there are 4 library functions defined under <stdlib.h> for dynamic memory allocation.

Function	Use of Function
malloc HYPERLINK "https://www.programiz.com/c-programming/c-dynamic-memory-allocation"()	Allocates requested size of bytes and returns a pointer first byte of allocated space
calloc HYPERLINK "https://www.programiz.com/c-programming/c-dynamic-memory-allocation"()	Allocates space for an array elements, initializes to zero and then returns a pointer to memory
free()	deallocate the previously allocated space
realloc HYPERLINK "https://www.programiz.com/c-programming/c-dynamic-memory-allocation"()	Change the size of previously allocated space

C malloc()

The name malloc stands for "memory allocation".

The function malloc() reserves a block of memory of specified size and return a pointer of type void which can be casted into pointer of any form.

Syntax of malloc()

```
ptr = (cast-type*) malloc(byte-size)
```

Here, ptr is pointer of cast-type. The malloc() function returns a pointer to an area of memory with size of byte size. If the space is insufficient, allocation fails and returns NULL pointer.

```
ptr = (int*) malloc(100 * sizeof(int));
```

This statement will allocate either 200 or 400 according to size of int 2 or 4 bytes respectively and the pointer points to the address of first byte of memory.

C calloc()

The name calloc stands for "contiguous allocation".

The only difference between malloc() and calloc() is that, malloc() allocates single block of memory whereas calloc() allocates multiple blocks of memory each of same size and sets all bytes to zero.

Syntax of calloc()

```
ptr = (cast-type*)calloc(n, element-size);
```

This statement will allocate contiguous space in memory for an array of n elements. For example:

```
ptr = (float*) calloc(25, sizeof(float));
```

This statement allocates contiguous space in memory for an array of 25 elements each of size of float, i.e, 4 bytes.

C free()

Dynamically allocated memory created with either calloc() or malloc() doesn't get freed on its own. You must explicitly use free() to release the space.

syntax of free()

```
free(ptr);
```

C realloc()

If the previously allocated memory is insufficient or more than required, you can change the previously allocated memory size using realloc().

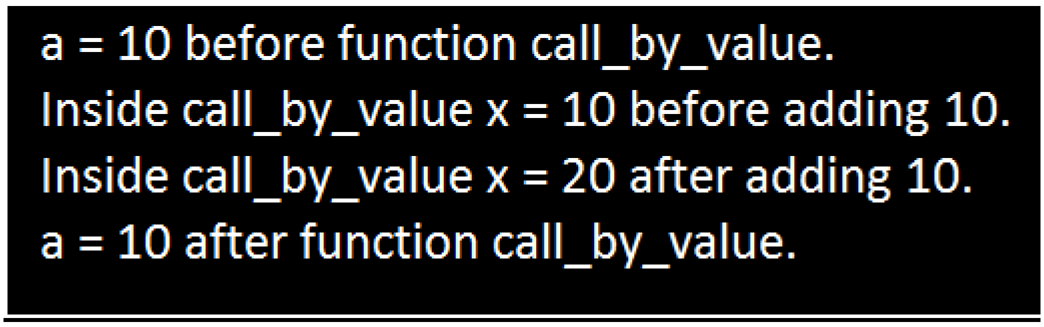
Syntax of realloc()

```
ptr = realloc(ptr, newsize);
```

PROGRAM: Attach a c program of call by value and call by reference.

```
#include <stdio.h>

void call_by_value(int x) {
        printf("Inside call_by_value x = %d before adding 10.\n", x);
        x += 10;
        printf("Inside call_by_value x = %d after adding 10.\n", x);
}

int main() {
        int a=10;

        printf("a = %d before function call_by_value.\n", a);
        call_by_value(a);
        printf("a = %d after function call_by_value.\n", a);
        return 0;
}
```

RESULT:

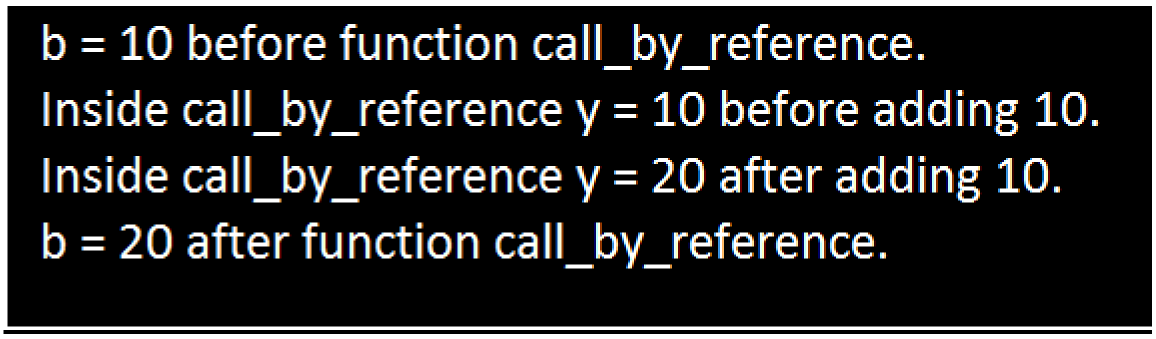


```
a = 10 before function call_by_value.
Inside call_by_value x = 10 before adding 10.
Inside call_by_value x = 20 after adding 10.
a = 10 after function call_by_value.
```

PROGRAM: Attach a c program of call by reference

```
#include <stdio.h>

void call_by_reference(int *y) {
        printf("Inside call_by_reference y = %d before adding 10.\n", *y);
        (*y) += 10;
        printf("Inside call_by_reference y = %d after adding 10.\n", *y);
}

int main() {
        int b=10;

        printf("b = %d before function call_by_reference.\n", b);
        call_by_reference(&b);
        printf("b = %d after function call_by_reference.\n", b);

        return 0;
}
```

RESULT:



```
b = 10 before function call_by_reference.
Inside call_by_reference y = 10 before adding 10.
Inside call_by_reference y = 20 after adding 10.
b = 20 after function call_by_reference.
```

PRACTICAL – 2

OBJECTIVE : Implement a program for stack that performs following operations using array. (a) PUSH (b) POP (C) EXIT (D) DISPLAY

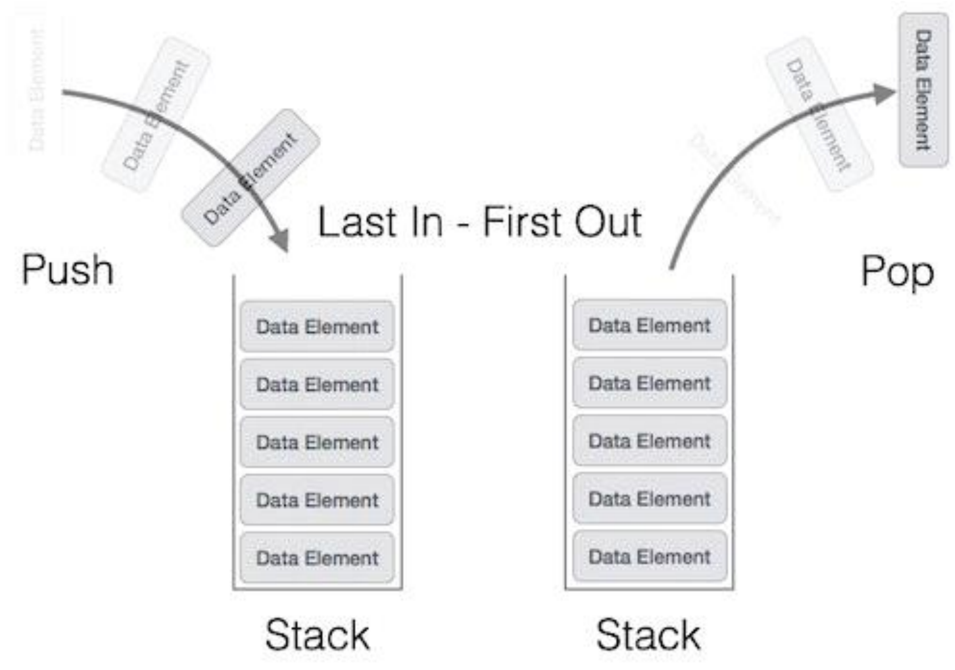
A stack is an Abstract Data Type (ADT), commonly used in most programming languages. It is named stack as it behaves like a real-world stack, for example – a deck of cards or a pile of plates, etc.

A real-world stack allows operations at one end only. For example, we can place or remove a card or plate from the top of the stack only. Likewise, Stack ADT allows all data operations at one end only. At any given time, we can only access the top element of a stack.

This feature makes it LIFO data structure. LIFO stands for Last-in-first-out. Here, the element which is placed (inserted or added) last, is accessed first. In stack terminology, insertion operation is called **PUSH** operation and removal operation is called **POP** operation.

Stack Representation

The following diagram depicts a stack and its operations −



A stack can be implemented by means of Array, Structure, Pointer, and Linked List. Stack can either be a fixed size one or it may have a sense of dynamic resizing. Here, we are going to implement stack using arrays, which makes it a fixed size stack implementation.

Basic Operations

Stack operations may involve initializing the stack, using it and then de-initializing it. Apart from these basic stuffs, a stack is used for the following two primary operations −

- **push()** − Pushing (storing) an element on the stack.
- **pop()** − Removing (accessing) an element from the stack.

When data is PUSHed onto stack.

To use a stack efficiently, we need to check the status of stack as well. For the same purpose, the following functionality is added to stacks −

- **peek()** − get the top data element of the stack, without removing it.
- **isFull()** − check if stack is full.
- **isEmpty()** − check if stack is empty.

At all times, we maintain a pointer to the last PUSHed data on the stack. As this pointer always represents the top of the stack, hence named **top**. The **top**pointer provides top value of the stack without actually removing it.

First we should learn about procedures to support stack functions −

peek()

Algorithm of peek() function −

```
begin procedure peek
   return stack[top]
end procedure
```

Implementation of peek() function in C programming language −

Example

```
int peek(){

return stack[top];

}
```

isfull()

Algorithm of isfull() function −

```
begin procedure isfull



if top equals to MAXSIZE

returntrue

else

returnfalse

   endif



end procedure
```

Implementation of isfull() function in C programming language −

Example

```
bool isfull(){

if(top == MAXSIZE)

returntrue;

else

returnfalse;

}
```

isempty()

Algorithm of isempty() function −

```
begin procedure isempty



if top less than 1

returntrue

else

returnfalse

   endif



end procedure
```

Implementation of isempty() function in C programming language is slightly different. We initialize top at -1, as the index in array starts from 0. So we check if the top is below zero or -1 to determine if the stack is empty. Here's the code −

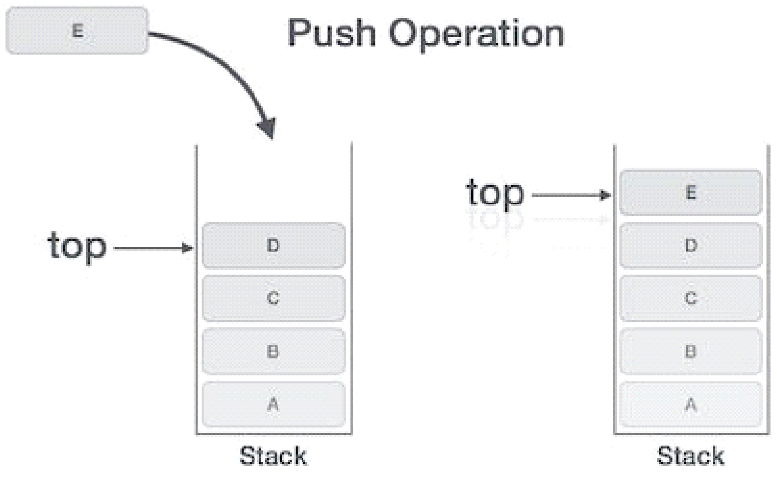
Example

```
bool isempty(){

if(top ==-1)
returntrue;
else
returnfalse;
}
```

Push Operation

The process of putting a new data element onto stack is known as a Push Operation. Push operation involves a series of steps −

- **Step 1** − Checks if the stack is full.
- **Step 2** − If the stack is full, produces an error and exit.
- **Step 3** − If the stack is not full, increments **top** to point next empty space.
- **Step 4** − Adds data element to the stack location, where top is pointing.
- **Step 5** − Returns success.



If the linked list is used to implement the stack, then in step 3, we need to allocate space dynamically.

Algorithm for PUSH Operation

A simple algorithm for Push operation can be derived as follows −

```
begin procedure push: stack, data


if stack is full

returnnull

   endif


   top ← top +1

   stack[top]← data


end procedure
```

Implementation of this algorithm in C, is very easy. See the following code −

Example

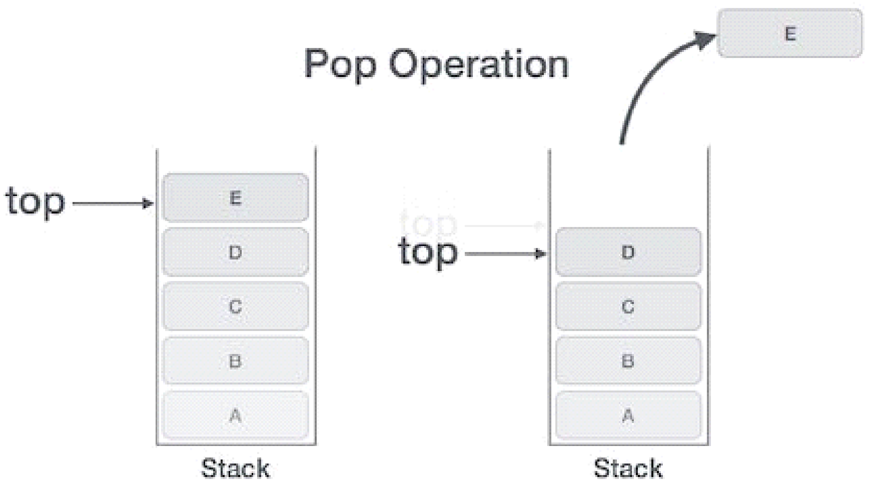
```
void push(int data){

if(!isFull()){

      top = top +1;

      stack[top]= data;

}else{

      printf("Could not insert data, Stack is full.\n");

}

}
```

Pop Operation

Accessing the content while removing it from the stack, is known as a Pop Operation. In an array implementation of pop() operation, the data element is not actually removed, instead **top** is decremented to a lower position in the stack to point to the next value. But in linked-list implementation, pop() actually removes data element and de allocates memory space.

A Pop operation may involve the following steps −

- **Step 1** − Checks if the stack is empty.
- **Step 2** − If the stack is empty, produces an error and exit.
- **Step 3** − If the stack is not empty, accesses the data element at which **top** is pointing.
- **Step 4** − Decreases the value of top by 1.
- **Step 5** − Returns success.



Algorithm for Pop Operation

A simple algorithm for Pop operation can be derived as follows −

```
begin procedure pop: stack
```

```
if stack is empty
returnnull
   endif

   data ← stack[top]
   top ← top -1
return data

end procedure
```

Implementation of this algorithm in C, is as follows −

Example

```
int pop(int data){

if(!isempty()){
      data = stack[top];
      top = top -1;
return data;
}else{
      printf("Could not retrieve data, Stack is empty.\n");
}
}
```

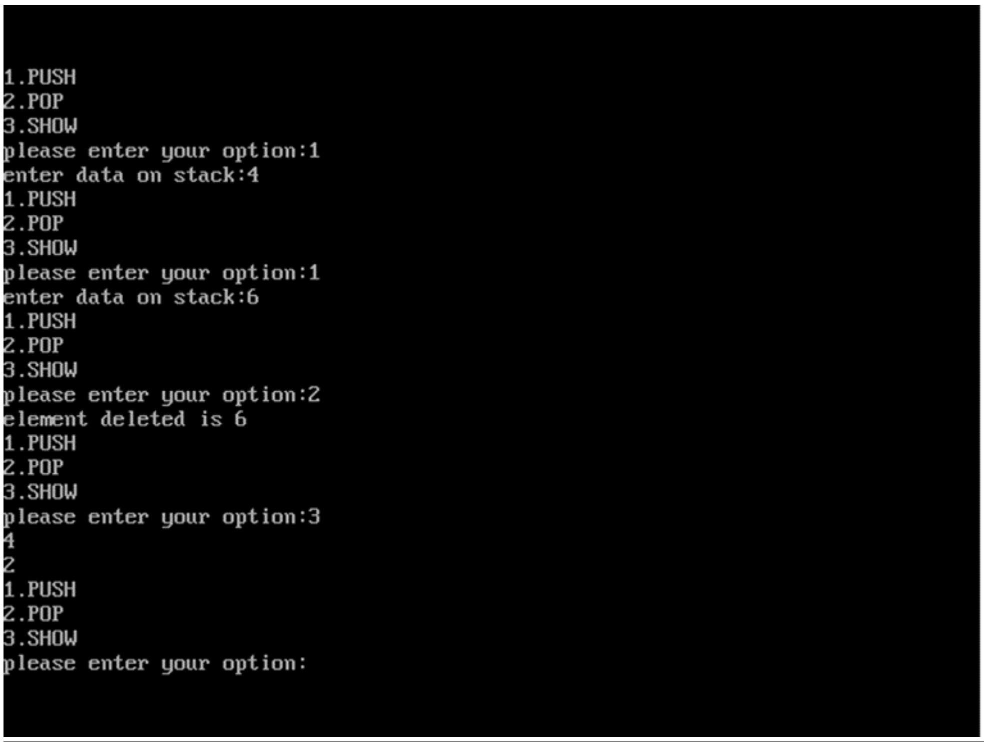
PROGRAM: Attach a c program which perform following operations push, pop, display stack .

```
#include<stdio.h>

int stack[100],choice,n,top,x,i;
void push(void);
void pop(void);
void display(void);
int main()
{
   //clrscr();
   top=-1;
   printf("\n Enter the size of STACK[MAX=100]:");
   scanf("%d",&n);
   printf("\n\t STACK OPERATIONS USING ARRAY");
   printf("\n\t--------------------------------");
   printf("\n\t 1.PUSH\n\t 2.POP\n\t 3.DISPLAY\n\t 4.EXIT");
   do
   {
      printf("\n Enter the Choice:");
      scanf("%d",&choice);
      switch(choice)
      {
         case 1:
         {
            push();
            break;
         }
         case 2:
         {
            pop();
            break;
         }
```

```
            case 3:
            {
                display();
                break;
            }
            case 4:
            {
                printf("\n\t EXIT POINT ");
                break;
            }
            default:
            {
                printf ("\n\t Please Enter a Valid Choice(1/2/3/4)");
            }

        }
    }
    while(choice!=4);
    return 0;
}
void push()
{
    if(top>=n-1)
    {
        printf("\n\tSTACK is over flow");

    }
    else
    {
        printf(" Enter a value to be pushed:");
        scanf("%d",&x);
        top++;
        stack[top]=x;
    }
```

```
}
void pop()
{
   if(top<=-1)
   {
      printf("\n\t Stack is under flow");
   }
   else
   {
      printf("\n\t The popped elements is %d",stack[top]);
      top--;
   }
}
void display()
{
   if(top>=0)
   {
      printf("\n The elements in STACK \n");
      for(i=top; i>=0; i--)
         printf("\n%d",stack[i]);
      printf("\n Press Next Choice");
   }
   else
   {
      printf("\n The STACK is empty");
   }

}
```

RESULT:



```
1.PUSH
2.POP
3.SHOW
please enter your option:1
enter data on stack:4
1.PUSH
2.POP
3.SHOW
please enter your option:1
enter data on stack:6
1.PUSH
2.POP
3.SHOW
please enter your option:2
element deleted is 6
1.PUSH
2.POP
3.SHOW
please enter your option:3
4
2
1.PUSH
2.POP
3.SHOW
please enter your option:
```

PRACTICAL – 3

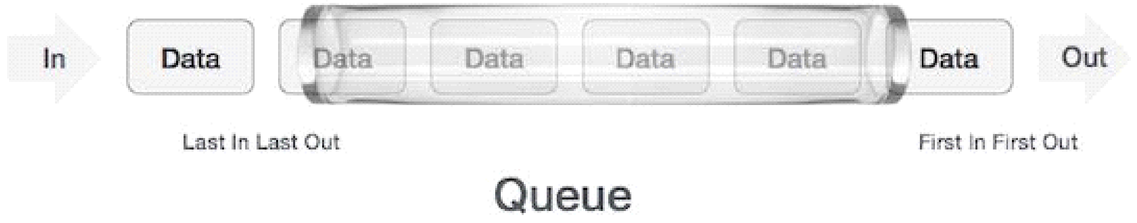
OBJECTIVE : Write a program to implement QUEUE using arrays that performs following operations (a) INSERT (b) DELETE (c) DISPLAY.

Queue is an abstract data structure, somewhat similar to Stacks. Unlike stacks, a queue is open at both its ends. One end is always used to insert data (enqueue) and the other is used to remove data (dequeue). Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first.

A real-world example of queue can be a single-lane one-way road, where the vehicle enters first, exits first. More real-world examples can be seen as queues at the ticket windows and bus-stops.

Queue Representation

As we now understand that in queue, we access both ends for different reasons. The following diagram given below tries to explain queue representation as data structure −



As in stacks, a queue can also be implemented using Arrays, Linked-lists, Pointers and Structures. For the sake of simplicity, we shall implement queues using one-dimensional array.

Basic Operations

Queue operations may involve initializing or defining the queue, utilizing it, and then completely erasing it from the memory. Here we shall try to understand the basic operations associated with queues −

- **enqueue()** − add (store) an item to the queue.
- **dequeue()** − remove (access) an item from the queue.

Few more functions are required to make the above-mentioned queue operation efficient. These are −

- **peek()** − Gets the element at the front of the queue without removing it.
- **isfull()** − Checks if the queue is full.
- **isempty()** − Checks if the queue is empty.

In queue, we always dequeue (or access) data, pointed by **front** pointer and while enqueing (or storing) data in the queue we take help of **rear** pointer.

Let's first learn about supportive functions of a queue −

peek()

This function helps to see the data at the **front** of the queue. The algorithm of peek() function is as follows −

Algorithm

```
begin procedure peek
   return queue[front]
end procedure
```

Implementation of peek() function in C programming language −

Example

```
int peek(){

return queue[front];

}
```

isfull()

As we are using single dimension array to implement queue, we just check for the rear pointer to reach at MAXSIZE to determine that the queue is full. In case we maintain the queue in a circular linked-list, the algorithm will differ. Algorithm of isfull() function −

Algorithm

```
begin procedure isfull


if rear equals to MAXSIZE
returntrue
else
returnfalse
   endif
end procedure
```

Implementation of isfull() function in C programming language −

Example

```
bool isfull(){
if(rear == MAXSIZE -1)
returntrue;
else
returnfalse;
}
```

isempty()

Algorithm of isempty() function −

Algorithm

```
begin procedure isempty

if front is less than MIN  OR front is greater than rear

returntrue

else

returnfalse

   endif



end procedure
```

If the value of **front** is less than MIN or 0, it tells that the queue is not yet initialized, hence empty.

Here's the C programming code −

Example

```
bool isempty(){

if(front <0|| front > rear)

returntrue;

else

returnfalse;

}
```

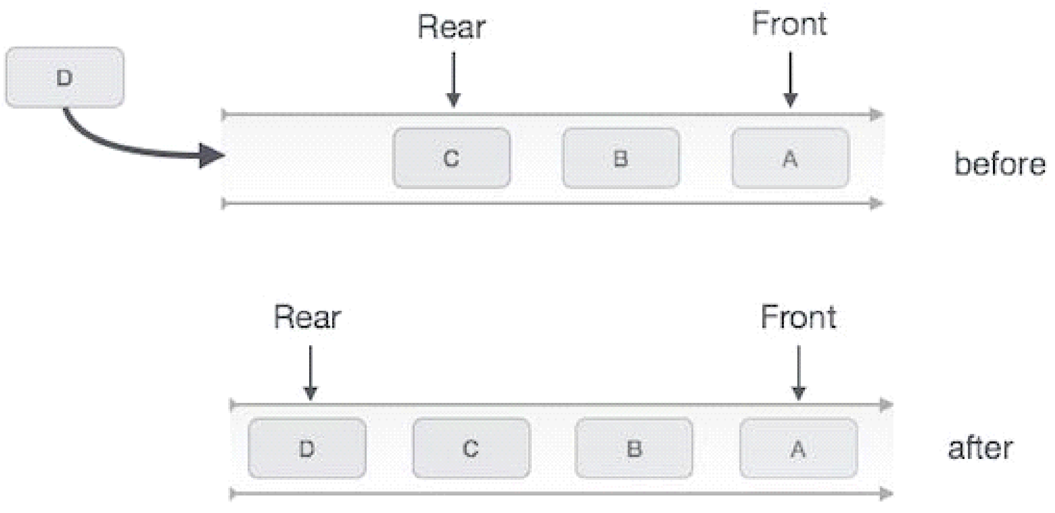
Enqueue Operation

Queues maintain two data pointers, **front** and **rear**. Therefore, its operations are comparatively difficult to implement than that of stacks.

The following steps should be taken to enqueue (insert) data into a queue −

- **Step 1** − Check if the queue is full.
- **Step 2** − If the queue is full, produce overflow error and exit.

- **Step 3** − If the queue is not full, increment **rear** pointer to point the next empty space.
- **Step 4** − Add data element to the queue location, where the rear is pointing.
- **Step 5** − return success.



Queue Enqueue

Sometimes, we also check to see if a queue is initialized or not, to handle any unforeseen situations.

Algorithm for enqueue operation

```
procedure enqueue(data)
if queue is full
return overflow
  endif
  rear ← rear +1
  queue[rear]← data
returntrue
end procedure
```

Implementation of enqueue() in C programming language −

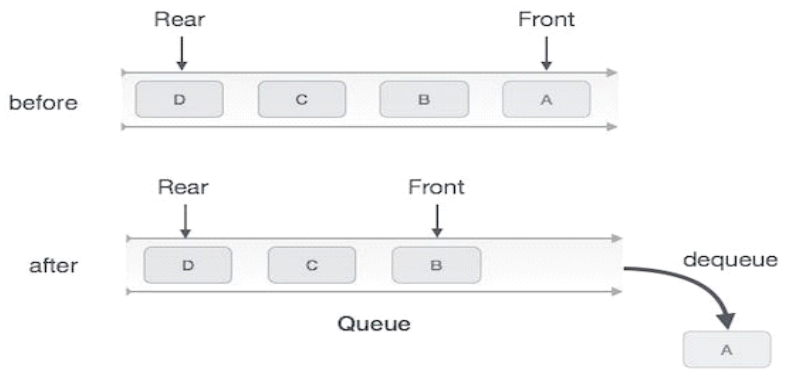
Example

```
int enqueue(int data)
if(isfull())
return0;
   rear = rear +1;
   queue[rear]= data;
return1;
end procedure
```

Dequeue Operation

Accessing data from the queue is a process of two tasks − access the data where **front** is pointing and remove the data after access. The following steps are taken to perform **dequeue** operation −

- **Step 1** − Check if the queue is empty.
- **Step 2** − If the queue is empty, produce underflow error and exit.
- **Step 3** − If the queue is not empty, access the data where **front** is pointing.
- **Step 4** − Increment **front** pointer to point to the next available data element.
- **Step 5** − Return success.



Queue Dequeue

Algorithm for dequeue operation

```
procedure dequeue
```

```
if queue is empty
return underflow
endif
  data = queue[front]
  front ← front +1
returntrue
end procedure
```

Implementation of dequeue() in C programming language −

Example

```
int dequeue(){
if(isempty())
return0;
int data = queue[front];
  front = front +1;
return data;
}
```

PROGRAM: Attach a c program which perform following operations insertion, deletion and display queue .

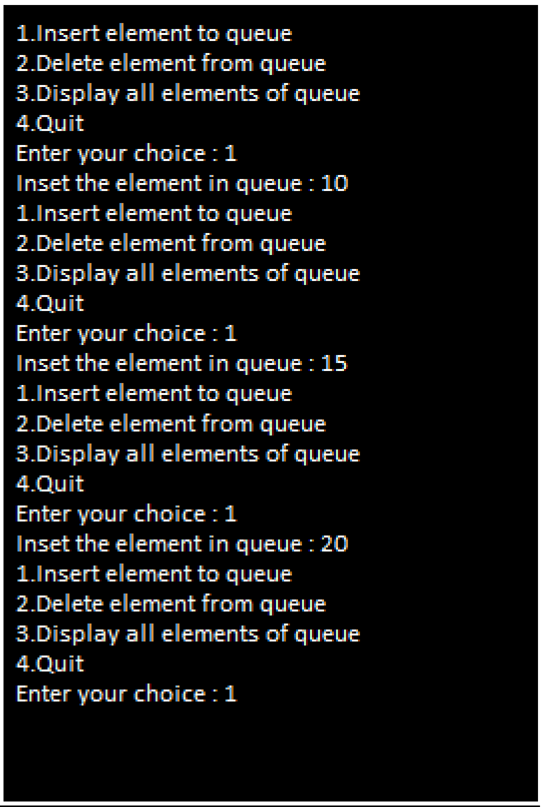
```
#include <stdio.h>
#define MAX 50
int queue_array[MAX];
int rear =-1;
int front =-1;
```

```
main()

{

int choice;

while(1)

{

printf("1.Insert element to queue \n");

printf("2.Delete element from queue \n");

printf("3.Display all elements of queue \n");

printf("4.Quit \n");

printf("Enter your choice : ");

scanf("%d",&choice);

switch(choice)

{

case1:

        insert();

break;

case2:

        delete();

break;

case3:

        display();

break;

case4:
```

```
exit(1);

default:

printf("Wrong choice \n");

}

}

}

insert()

{

int add_item;

if(rear == MAX -1)

printf("Queue Overflow \n");

else

{

if(front ==-1)


        front =0;

printf("Inset the element in queue : ");

scanf("%d",&add_item);

        rear = rear +1;

        queue_array[rear]= add_item;

}

}

delete()
```

```
{
if(front ==-1|| front > rear)
{
printf("Queue Underflow \n");
return;
}
else
{
printf("Element deleted from queue is : %d\n", queue_array[front]);
        front = front +1;
}
}
display()
{
int i;
if(front ==-1)
printf("Queue is empty \n");
else
{
printf("Queue is : \n");
for(i = front; i <= rear; i++)
printf("%d ", queue_array[i]);
printf("\n");
}

}
```

RESULT:



```
1.Insert element to queue
2.Delete element from queue
3.Display all elements of queue
4.Quit
Enter your choice : 1
Inset the element in queue : 10
1.Insert element to queue
2.Delete element from queue
3.Display all elements of queue
4.Quit
Enter your choice : 1
Inset the element in queue : 15
1.Insert element to queue
2.Delete element from queue
3.Display all elements of queue
4.Quit
Enter your choice : 1
Inset the element in queue : 20
1.Insert element to queue
2.Delete element from queue
3.Display all elements of queue
4.Quit
Enter your choice : 1
```

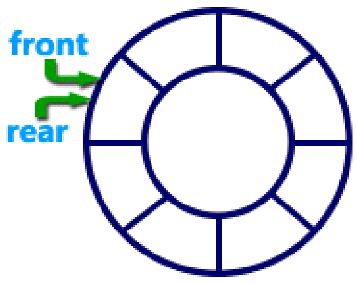
PRACTICAL – 4

OBJECTIVE : Write a program to implement Circular Queue using arrays that performs following operations. (a) INSERT (b) DELETE (c) DISPLAY

A Circular Queue can be defined as follows...

Circular Queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle.

Graphical representation of a circular queue is as follows...



Implementation of Circular Queue

To implement a circular queue data structure using array, we first perform the following steps before we implement actual operations.

- **Step 1:** Include all the **header files** which are used in the program and define a constant **'SIZE'** with specific value.
- **Step 2:** Declare all **user defined functions** used in circular queue implementation.
- **Step 3:** Create a one dimensional array with above defined SIZE (**int cQueue[SIZE]**)
- **Step 4:** Define two integer variables **'front'** and '**rear**' and initialize both with **'-1'**. (**int front = -1, rear = -1**)
- **Step 5:** Implement main method by displaying menu of operations list and make suitable function calls to perform operation selected by the user on circular queue.

enQueue(value) - Inserting value into the Circular Queue

In a circular queue, enQueue() is a function which is used to insert an element into the circular queue. In a circular queue, the new element is always inserted at **rear** position. The enQueue() function takes one integer value as parameter and inserts that value into the circular queue. We can use the following steps to insert an element into the circular queue...

- **Step 1:** Check whether **queue** is **FULL**. (**(rear == SIZE-1 && front == 0) || (front == rear+1)**)
- **Step 2:** If it is **FULL**, then display **"Queue is FULL!!! Insertion is not possible!!!"** and terminate the function.
- **Step 3:** If it is **NOT FULL**, then check **rear == SIZE - 1 && front != 0** if it is **TRUE**, then set **rear = -1**.
- **Step 4:** Increment **rear** value by one (**rear++**), set **queue[rear]** = **value** and check '**front == -1**' if it is **TRUE**, then set **front = 0**.

deQueue() - Deleting a value from the Circular Queue

In a circular queue, deQueue() is a function used to delete an element from the circular queue. In a circular queue, the element is always deleted from **front** position. The deQueue() function doesn't take any value as parameter. We can use the following steps to delete an element from the circular queue...

- **Step 1:** Check whether **queue** is **EMPTY**. (**front == -1 && rear == -1**)
- **Step 2:** If it is **EMPTY**, then display **"Queue is EMPTY!!! Deletion is not possible!!!"** and terminate the function.
- **Step 3:** If it is **NOT EMPTY**, then display **queue[front]** as deleted element and increment the **front** value by one (**front ++**). Then check whether **front == SIZE**, if it is **TRUE**, then set **front = 0**. Then check whether both **front - 1** and **rear** are equal (**front -1** == **rear**), if it **TRUE**, then set both **front** and **rear** to '**-1**' (**front** = **rear** = **-1**).

display() - Displays the elements of a Circular Queue

We can use the following steps to display the elements of a circular queue...

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

- **Step 2:** If it is **EMPTY**, then display **"Queue is EMPTY!!!"** and terminate the function.
- **Step 3:** If it is **NOT EMPTY**, then define an integer variable '**i**' and set '**i** = **front**'.
- **Step 4:** Check whether '**front <= rear**', if it is **TRUE**, then display '**queue[i]**' value and increment '**i**' value by one (**i++**). Repeat the same until '**i <= rear**' becomes **FALSE**.
- **Step 5:** If '**front <= rear**' is **FALSE**, then display '**queue[i]**' value and increment '**i**' value by one (**i++**). Repeat the same until'**i <= SIZE - 1**' becomes **FALSE**.
- **Step 6:** Set **i** to **0**.
- **Step 7:** Again display '**cQueue[i]**' value and increment **i** value by one (**i++**). Repeat the same until '**i <= rear**' becomes **FALSE**.

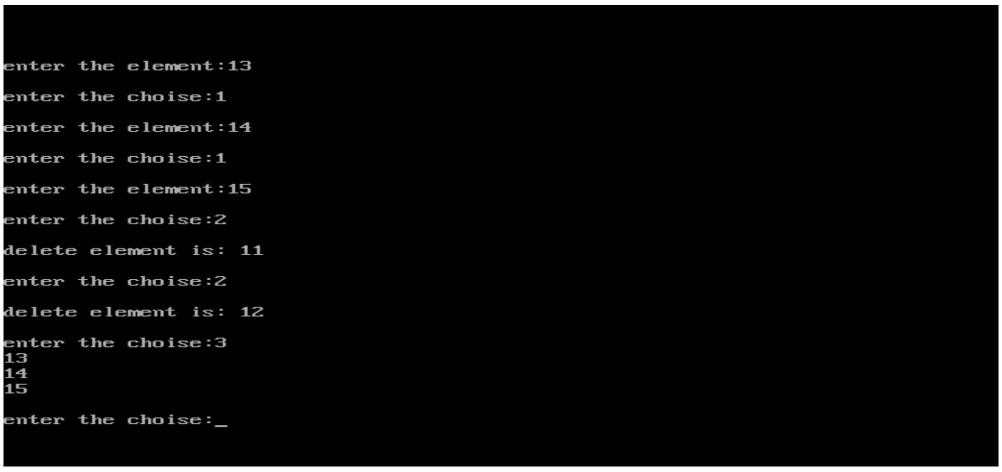
PROGRAM: Attach a c program which perform following operations insertion, deletion and display queue .

```
#include<stdio.h>
#define max 3
int q[10],front=0,rear=-1;
void main()
{
int ch;
void insert();
void delet();
void display();
clrscr();
printf("\nCircular Queue operations\n");
printf("1.insert\n2.delete\n3.display\n4.exit\n");
while(1)
{
        printf("Enter your choice:");
        scanf("%d",&ch);
        switch(ch)
        {
        case 1: insert();
                break;
        case 2: delet();
                break;
        case 3:display();
                break;
        case 4:exit();
        default:printf("Invalid option\n");
        }
    }
    }

    void insert()
    {
    int x;
    if((front==0&&rear==max-1)||(front>0&&rear==front-1))
        printf("Queue is overflow\n");
    else
    {
        printf("Enter element to be insert:");
        scanf("%d",&x);
        if(rear==max-1&&front>0)
        {
                rear=0;
                q[rear]=x;
        }
        else
        {
                if((front==0&&rear==-1)||(rear!=front-1))
                q[++rear]=x;
        }
```

```
}
}
void  delet()
{
int a;
if((front==0)&&(rear==-1))
{
        printf("Queue is underflow\n");
        getch();
        exit();
}
if(front==rear)
{
        a=q[front];
        rear=-1;
        front=0;
}
else
        if(front==max-1)
        {
                a=q[front];
                front=0;
        }
        else a=q[front++];
        printf("Deleted element is:%d\n",a);
}

void display()
{
int i,j;
if(front==0&&rear==-1)
{
        printf("Queue is underflow\n");
        getch();
```

```
            exit();
    }
    if(front>rear)
    {
            for(i=0;i<=rear;i++)
                    printf("\t%d",q[i]);
            for(j=front;j<=max-1;j++)
                    printf("\t%d",q[j]);
            printf("\nrear is at %d\n",q[rear]);
            printf("\nfront is at %d\n",q[front]);
    }
    else
    {
            for(i=front;i<=rear;i++)
            {
                    printf("\t%d",q[i]);
            }
            printf("\nrear is at %d\n",q[rear]);
            printf("\nfront is at %d\n",q[front]);
    }
    printf("\n");
    }
getch();
```

RESULT:



```
enter the element:13
enter the choise:1
enter the element:14
enter the choise:1
enter the element:15
enter the choise:2
delete element is: 11
enter the choise:2
delete element is: 12
enter the choise:3
13
14
15

enter the choise:_
```

PRACTICAL – 5

OBJECTIVE :Write a menu driven program to implement following operations on the singly linked list.

(a) Insert a node at the front of the linked list.

(b) Insert a node at the end of the linked list.

(c) Insert a node at specific place.

(d) Delete a first node of the linked list.

(e) Delete a specified node.

(f) Delete a last node of the linked list.

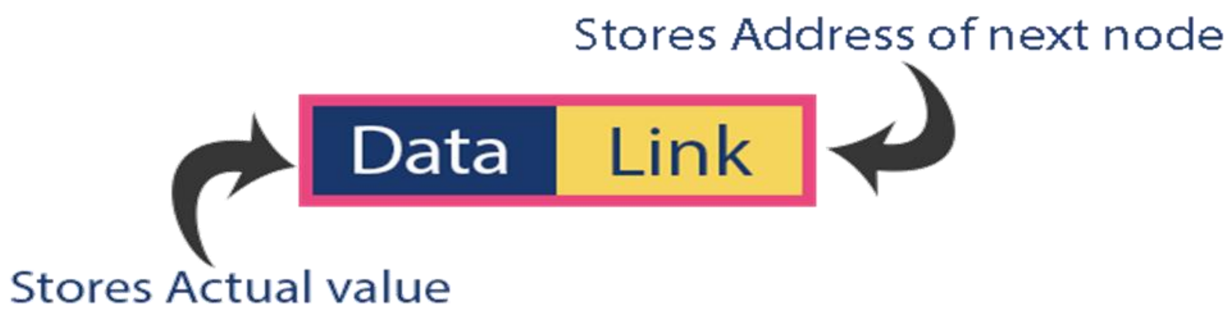
Simply a list is a sequence of data, and linked list is a sequence of data linked with each other.

The formal definition of a single linked list is as follows...

"Single linked list is a sequence of elements in which every element has link to its next element in the sequence."

In any single linked list, the individual element is called as "Node". Every "Node" contains two fields, data and next. The data field is used to store actual value of that node and next field is used to store the address of the next node in the sequence.

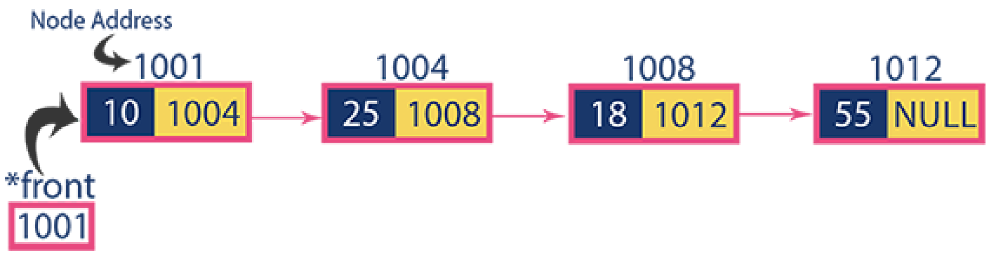
The graphical representation of a node in a single linked list is as follows...



NOTE :-

- In a single linked list, the address of the first node is always stored in a reference node known as "front" (Some times it is also known as "head").
- Always next part (reference part) of the last node must be NULL.

Example



In a single linked list, the insertion operation can be performed in three ways. They are as follows...

1. Inserting At Beginning of the list
2. Inserting At End of the list
3. Inserting At Specific location in the list

Inserting At Beginning of the list

We can use the following steps to insert a new node at beginning of the single linked list...

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

Inserting At End of the list

We can use the following steps to insert a new node at end of the single linked list...

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

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

We can use the following steps to insert a new node after a node in the single linked list...

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

In a single linked list, the deletion operation can be performed in three ways. They are as follows...

1. Deleting from Beginning of the list
2. Deleting from End of the list
3. Deleting a Specific Node

Deleting from Beginning of the list

We can use the following steps to delete a node from beginning of the single linked list...

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

Deleting from End of the list

We can use the following steps to delete a node from end of the single linked list...

- **Step 1:** Check whether list is **Empty** (**head** == **NULL**)
- **Step 2:** If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
- **Step 3:** If it is **Not Empty** then, define two Node pointers **'temp1'** and '**temp2'** and initialize '**temp1**' with **head**.
- **Step 4:** Check whether list has only one Node (**temp1 → next** == **NULL**)
- **Step 5:** If it is **TRUE**. Then, set **head** = **NULL** and delete **temp1**. And terminate the function. (Setting **Empty** list condition)
- **Step 6:** If it is **FALSE**. Then, set '**temp2 = temp1** ' and move **temp1** to its next node. Repeat the same until it reaches to the last node in the list. (until **temp1 → next** == **NULL**)
- **Step 7:** Finally, Set **temp2 → next** = **NULL** and delete **temp1**.

Deleting a Specific Node from the list

We can use the following steps to delete a specific node from the single linked list...

- **Step 1:** Check whether list is **Empty** (**head** == **NULL**)
- **Step 2:** If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.

- **Step 3:** If it is **Not Empty** then, define two Node pointers **'temp1'** and '**temp2**' and initialize '**temp1**' with **head**.
- **Step 4:** Keep moving the **temp1** until it reaches to the exact node to be deleted or to the last node. And every time set '**temp2 = temp1**' before moving the '**temp1**' to its next node.
- **Step 5:** If it is reached to the last node then display **'Given node not found in the list! Deletion not possible!!!'**. And terminate the function.
- **Step 6:** If it is reached to the exact node which we want to delete, then check whether list is having only one node or not
- **Step 7:** If list has only one node and that is the node to be deleted, then set **head** = **NULL** and delete **temp1** (**free(temp1)**).
- **Step 8:** If list contains multiple nodes, then check whether **temp1** is the first node in the list (**temp1 == head**).
- **Step 9:** If **temp1** is the first node then move the **head** to the next node (**head = head → next**) and delete **temp1**.
- **Step 10:** If **temp1** is not first node then check whether it is last node in the list (**temp1 → next == NULL**).
- **Step 11:** If **temp1** is last node then set **temp2 → next** = **NULL** and delete **temp1** (**free(temp1)**).
- **Step 12:** If **temp1** is not first node and not last node then set **temp2 → next** = **temp1 → next** and delete **temp1** (**free(temp1)**).

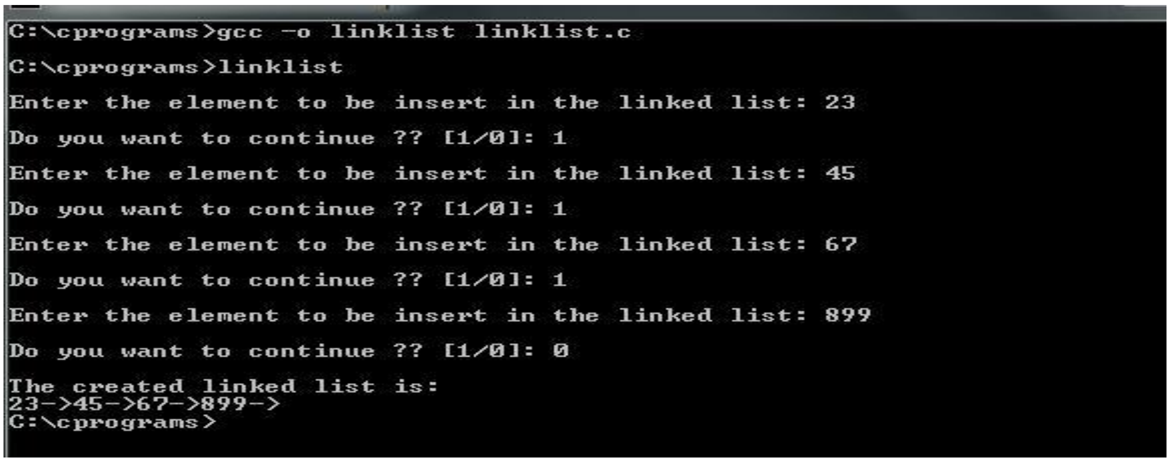
PROGRAM: Attach a c program of singly linked list.

```
#include<stdio.h>

#include<conio.h>

#include<alloc.h>

struct node

{
```

```
int data;

struct node *next;

}*start=NULL;



void creat()

{

char ch;

do

 {

struct node *new_node,*current;



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



  printf("nEnter the data : ");

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

  new_node->next=NULL;



if(start==NULL)

  {

  start=new_node;
  current=new_node;

  }

else

  {

  current->next=new_node;

  current=new_node;

  }


 printf("nDo you want to creat another : ");

 ch=getche();

 }while(ch!='n');

}


void display()

{

struct node *new_node;

 printf("The Linked List : n");

 new_node=start;

while(new_node!=NULL)

   {
```

```
   printf("%d--->",new_node->data);

   new_node=new_node->next;

   }

  printf("NULL");

}

void main()

{

create();

display();

}
```

RESULT:



```
C:\cprograms>gcc -o linklist linklist.c
C:\cprograms>linklist
Enter the element to be insert in the linked list: 23
Do you want to continue ?? [1/0]: 1
Enter the element to be insert in the linked list: 45
Do you want to continue ?? [1/0]: 1
Enter the element to be insert in the linked list: 67
Do you want to continue ?? [1/0]: 1
Enter the element to be insert in the linked list: 899
Do you want to continue ?? [1/0]: 0
The created linked list is:
23->45->67->899->
C:\cprograms>
```

PRACTICAL – 6

OBJECTIVE: Write a program to implement following operations on the doubly linked list.

(a) Insert a node at the front of the linked list.

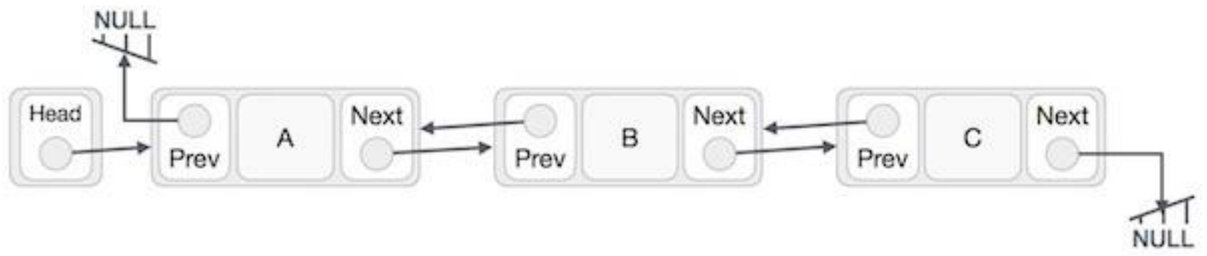
(b) Insert a node at the end of the linked list.

(c) Delete a middle node of the linked list.

Doubly Linked List is a variation of Linked list in which navigation is possible in both ways, either forward and backward easily as compared to Single Linked List. Following are the important terms to understand the concept of doubly linked list.

- **Link** − Each link of a linked list can store a data called an element.
- **Next** − Each link of a linked list contains a link to the next link called Next.
- **Prev** − Each link of a linked list contains a link to the previous link called Prev.
- **LinkedList** − A Linked List contains the connection link to the first link called First and to the last link called Last.

Doubly Linked List Representation



As per the above illustration, following are the important points to be considered.

- Doubly Linked List contains a link element called first and last.
- Each link carries a data field(s) and two link fields called next and prev.
- Each link is linked with its next link using its next link.
- Each link is linked with its previous link using its previous link.
- The last link carries a link as null to mark the end of the list.

In a double linked list, the insertion operation can be performed in three ways as follows...

Inserting At Beginning of the list

We can use the following steps to insert a new node at beginning of the double linked list...

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

Inserting At End of the list

We can use the following steps to insert a new node at end of the double linked list...

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

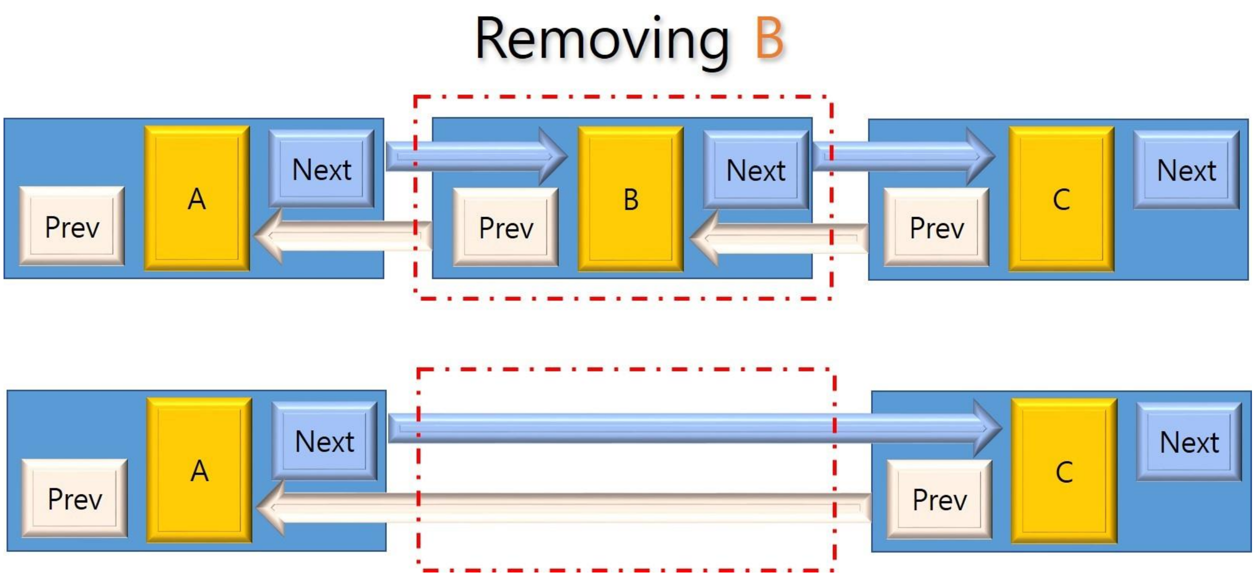
Removing from the middle of the list

This is probably the most complex case out of the three, because we now have to work with three nodes:

1. Previous node.
2. Next node.

3. Node to remove.

Removing a node from the linked list is not as complicated as inserting an item into the middle of the list. If you got that, this should be a cakewalk.



We need to update the following pointers.

- **Next node** pointer of **A**. This should be updated to point to **C**.
- **Prev node** pointer of **C**. This should be updated to point to **A**.

Lastly, we **destroy B** by setting it to null.

Pseudo Code

Now that we have reviewed all the cases, let's take a look at the pseudo code for the entire remove operation.

<u>Remove operation</u>

```
begin remove(T dataToRemove):
currentNode = head
while current node is not null:
currentData = currentNode.data
```

```
if  currentData equals dataToRemove:
call removeNode(currentNode)
end if;
currentNode = currentNode.next
end while;
end remove;
```

removeNode operation

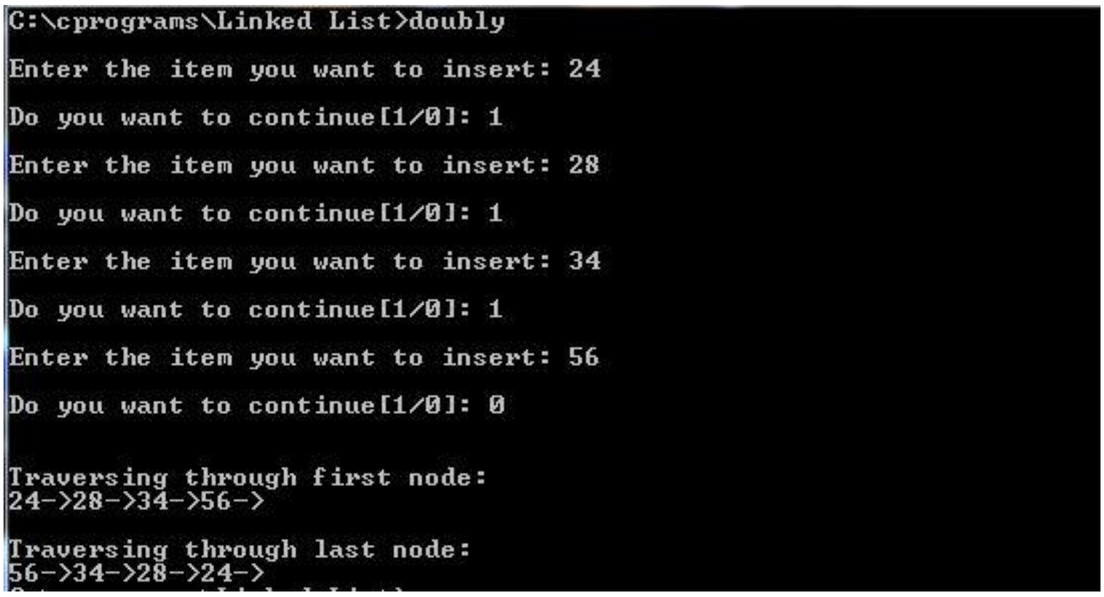
```
begin removeNode(nodeToRemove):
prevNode = nodeToRemove.prev
nextNode = nodeToRemove.next
// Head node does not have a previous node
if prevNode is null:
head = null
head = nextNode
head.prev = null
else if nextNode is null:
// Tail does not have a next node
tail = null
tail = prevNode
tail.next = null
else:
// Is somewhere in the middle of the linked list
// Set current node to null
nodeToRemove = null
// connect the previous and next nodes together
prevNode.next = nextNode
nextNode.prev = prevNode
end if else statement;
decrement size
```

```
end removeNode;
```

PROGRAM: Attach a c program of doubly linked list.

```
#include <stdio.h>
#include <conio.h>
#include <alloc.h>
struct linklist
{
    struct linklist *prev;
    int num;
    struct linklist *next;
};
typedef struct linklist node;
void create(node *);
void display(node *);
void main()
{
    node *head;
    clrscr();
    head=(node *) malloc(sizeof(node));
    head->prev=NULL;

    create(head);

    printf("\n");

    display(head);

    getch();

}

void create(node *list)

{

    char conf='y';

    int i;

    printf("\nENTER A NUMBER: ");

    scanf("%d",&list->num);

    list->next->prev=list;

    printf("\nWANT TO CONTINUE[Y/N]: ");

    conf=getche();

    printf("\n");

    if(conf=='n' || conf=='N')

            list->next=NULL;

    else

    {
            list->next=(node *) malloc(sizeof(node));

            create(list->next);

    }

}

void display(node *disp)

{

printf("THE ADDRESS-PREVIOUS ADDRESS-VALUE-NEXT ADDRESS OF THE LINK
LIST ARE:\n");

    while(disp!=NULL)

    {

            printf("%u-%u-%u-%u\n",disp,disp->prev,disp->num,disp->next);

            disp=disp->next;

    }

}
```

RESULT:



```
C:\cprograms\Linked List>doubly

Enter the item you want to insert: 24

Do you want to continue[1/0]: 1

Enter the item you want to insert: 28

Do you want to continue[1/0]: 1

Enter the item you want to insert: 34

Do you want to continue[1/0]: 1

Enter the item you want to insert: 56

Do you want to continue[1/0]: 0


Traversing through first node:
24->28->34->56->

Traversing through last node:
56->34->28->24->
```

PRACTICAL – 7

OBJECTIVE: Write a program which create binary search tree. Implement recursive and non-recursive tree traversing methods inorder, preorder and post-order traversal.

PROGRAM: Attach a c program of BST

```
#include<stdio.h>

#include<conio.h>

#include<stdio.h>

struct tree {

        int info;

        struct tree *left;

        struct tree *right;

};

struct tree *insert(struct tree *,int);

void inorder(struct tree *);

void postorder(struct tree *);

void preorder(struct tree *);

struct tree *delet(struct tree *,int);

struct tree *search(struct tree *);

int main(void) {

        struct tree *root;

        int choice, item,item_no;

        root = NULL;

        clrscr();
```

```
/* rear  = NULL;*/

do {

        do {

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

                printf("\n\t 2. Delete from Binary Tree ");

                printf("\n\t 3. Inorder traversal of Binary tree");

                printf("\n\t 4. Postorder traversal of Binary tree");

                printf("\n\t 5. Preorder traversal of Binary tree");

                printf("\n\t 6. Search and replace ");

                printf("\n\t 7. Exit ");

                printf("\n\t Enter choice : ");

                scanf(" %d",&choice);

                if(choice<1 || choice>7)

                    printf("\n Invalid choice - try again");

        }

        while (choice<1 || choice>7);

        switch(choice) {

                case 1:

                         printf("\n Enter new element: ");

                scanf("%d", &item);

                root= insert(root,item);

                printf("\n root is %d",root->info);

                printf("\n Inorder traversal of binary tree is : ");

                inorder(root);
```

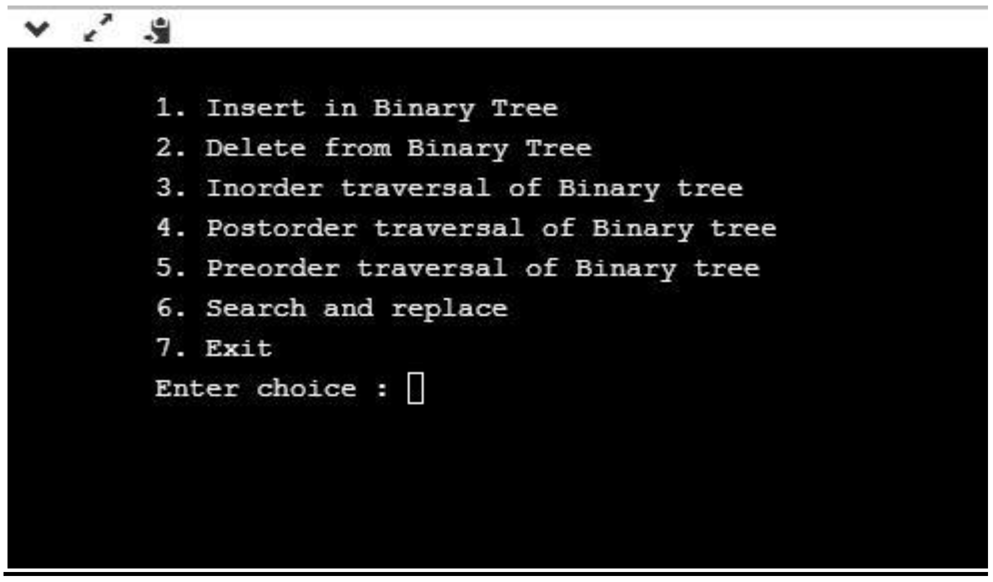
```
break;

case 2:

        printf("\n Enter the element to be deleted : ");

scanf(" %d",&item_no);

root=delet(root,item_no);

inorder(root);

break;

case 3:

        printf("\n Inorder traversal of binary tree is : ");

inorder(root);

break;

case 4:

        printf("\n Postorder traversal of binary tree is : ");

postorder(root);

break;

case 5:

        printf("\n Preorder traversal of binary tree is : ");

preorder(root);

break;

case 6:

        printf("\n Search and replace operation in binary tree ");

root=search(root);

break;

default:
```

```
                        printf("\n End of program ");
            }
            /* end of switch */
        }
        while(choice !=7);
        return(0);
}
struct tree *insert(struct tree *root, int x) {
        if(!root) {
            root=(struct tree*)malloc(sizeof(struct tree));
            root->info = x;
            root->left = NULL;
            root->right = NULL;
            return(root);
        }
        if(root->info > x)
          root->left = insert(root->left,x); else {
            if(root->info < x)
                root->right = insert(root->right,x);
        }
        return(root);
}
void inorder(struct tree *root) {
        if(root != NULL) {
```

```
		inorder(root->left);

		printf(" %d",root->info);

		inorder(root->right);

	}

	return;

}

void postorder(struct tree *root) {

	if(root != NULL) {

		postorder(root->left);

		postorder(root->right);

		printf(" %d",root->info);

	}

	return;

}

void preorder(struct tree *root) {

	if(root != NULL) {

		printf(" %d",root->info);

		preorder(root->left);

		preorder(root->right);

	}

	return;

}

/* FUNCTION TO DELETE A NODE FROM A BINARY TREE */

struct tree *delet(struct tree *ptr,int x) {
```

```
struct tree *p1,*p2;
if(!ptr) {
        printf("\n Node not found ");
        return(ptr);
} else {
        if(ptr->info < x) {
                ptr->right = delet(ptr->right,x);
                /*return(ptr);*/
        } else if (ptr->info >x) {
                ptr->left=delet(ptr->left,x);
                return ptr;
        } else
        /* no. 2 else */ {
                if(ptr->info == x)
                /* no. 2 if */ {
                        if(ptr->left == ptr->right)
                        /*i.e., a leaf node*/ {
                                free(ptr);
                                return(NULL);
                        } else if(ptr->left==NULL)
                        /* a right subtree */ {
                                p1=ptr->right;
                                free(ptr);
                                return p1;
                        } else if(ptr->right==NULL)

                        /* a left subtree */ {

                                p1=ptr->left;

                                free(ptr);

                                return p1;

                        } else {

                                p1=ptr->right;

                                p2=ptr->right;

                                while(p1->left != NULL)

                                        p1=p1->left;

                                p1->left=ptr->left;

                                free(ptr);

                                return p2;

                        }

                }

                /*end of no. 2 if */

        }

        /* end of no. 2 else */

        /* check which path to search for a given no. */

    }

    return(ptr);

}

/* function to search and replace an element in the binary tree */

struct tree *search(struct tree *root) {
    int no,i,ino;

    struct tree *ptr;

    ptr=root;

    printf("\n Enter the element to be searched :");

    scanf(" %d",&no);

    fflush(stdin);

    while(ptr) {

        if(no>ptr->info)

            ptr=ptr->right; else if(no<ptr->info)

            ptr=ptr->left; else

            break;

    }

    if(ptr) {

        printf("\n Element %d which was searched is found and is = %d",no,ptr-
>info);

        printf("\n Do you want replace it, press 1 for yes : ");

        scanf(" %d",&i);

        if(i==1) {

                printf("\n Enter new element :");

                scanf(" %d",&ino);

                ptr->info=ino;

        } else

            printf("\n\t It's okay");

    } else

      printf("\n Element %d does not exist in the binary tree",no);
    return(root);
}
```

RESULT:



```
1. Insert in Binary Tree
2. Delete from Binary Tree
3. Inorder traversal of Binary tree
4. Postorder traversal of Binary tree
5. Preorder traversal of Binary tree
6. Search and replace
7. Exit
Enter choice :
```

PRACTICAL – 8

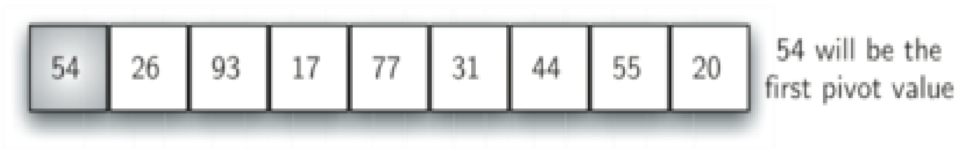
OBJECTIVE: Write a program to implement Quick Sort. Write a program to implement Merge Sort

Quick Sort

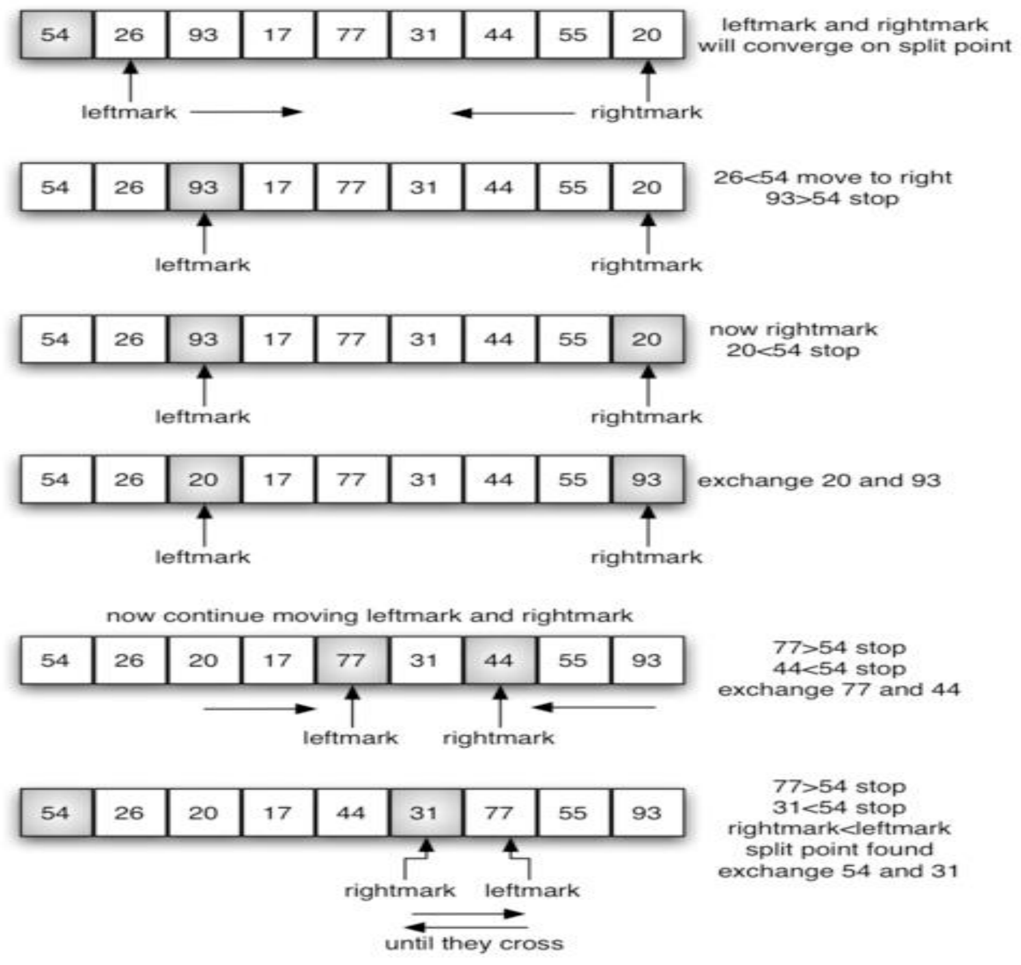
The **quick sort** uses divide and conquer to gain the same advantages as the merge sort, while not using additional storage. As a trade-off, however, it is possible that the list may not be divided in half. When this happens, we will see that performance is diminished.

A quick sort first selects a value, which is called the **pivot value**. Although there are many different ways to choose the pivot value, we will simply use the first item in the list. The role of the pivot value is to assist with splitting the list. The actual position where the pivot value belongs in the final sorted list, commonly called the **split point**, will be used to divide the list for subsequent calls to the quick sort.

Figure 12 shows that 54 will serve as our first pivot value. Since we have looked at this example a few times already, we know that 54 will eventually end up in the position currently holding 31. The **partition**process will happen next. It will find the split point and at the same time move other items to the appropriate side of the list, either less than or greater than the pivot value.

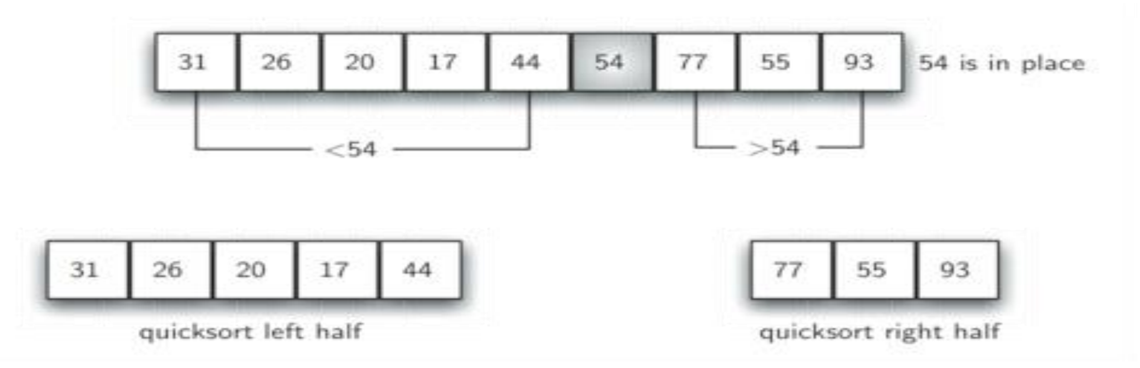


Partitioning begins by locating two position markers—let's call them leftmark and rightmark—at the beginning and end of the remaining items in the list (positions 1 and 8 in Figure 13). The goal of the partition process is to move items that are on the wrong side with respect to the pivot value while also converging on the split point. Figure 13 shows this process as we locate the position of 54.



We begin by incrementing leftmark until we locate a value that is greater than the pivot value. We then decrement rightmark until we find a value that is less than the pivot value. At this point we have discovered two items that are out of place with respect to the eventual split point. For our example, this occurs at 93 and 20. Now we can exchange these two items and then repeat the process again.

At the point where rightmark becomes less than leftmark, we stop. The position of rightmark is now the split point. The pivot value can be exchanged with the contents of the split point and the pivot value is now in place (Figure 14). In addition, all the items to the left of the split point are less than the pivot value, and all the items to the right of the split point are greater than the pivot value. The list can now be divided at the split point and the quick sort can be invoked recursively on the two halves.



The quickSort function shown in ActiveCode 1 invokes a recursive function, quickSortHelper. quickSortHelper begins with the same base case as the merge sort. If the length of the list is less than or equal to one, it is already sorted. If it is greater, then it can be partitioned and recursively sorted. The partition function implements the process described earlier.

Quick Sort Pivot Algorithm

Based on our understanding of partitioning in quick sort, we will now try to write an algorithm for it, which is as follows.

Step 1 − Choose the highest index value has pivot
Step 2 − Take two variables to point left and right of the list excluding pivot
Step 3 − left points to the low index
Step 4 − right points to the high
Step 5 − while value at left is less than pivot move right
Step 6 − while value at right is greater than pivot move left
Step 7 − if both step 5 and step 6 does not match swap left and right
Step 8 − if left ≥ right, the point where they met is new pivot

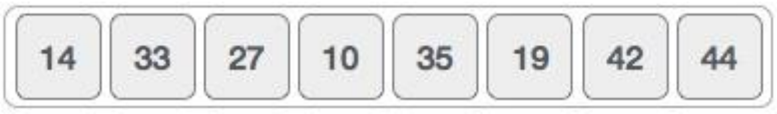
Merge Sort

Merge sort is a sorting technique based on divide and conquer technique. With worst-case time complexity being O(n log n), it is one of the most respected algorithms.

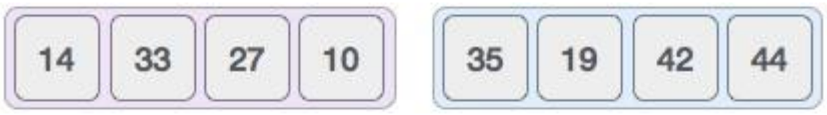
Merge sort first divides the array into equal halves and then combines them in a sorted manner.

How Merge Sort Works?

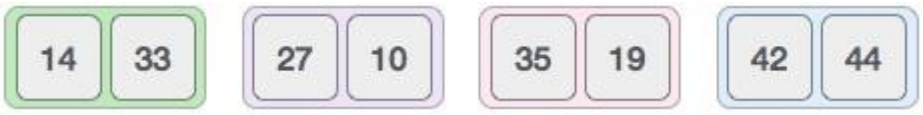
To understand merge sort, we take an unsorted array as the following −



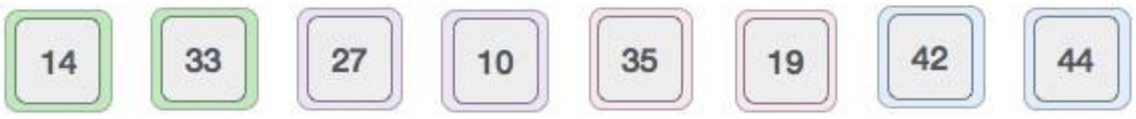
We know that merge sort first divides the whole array iteratively into equal halves unless the atomic values are achieved. We see here that an array of 8 items is divided into two arrays of size 4.



This does not change the sequence of appearance of items in the original. Now we divide these two arrays into halves.

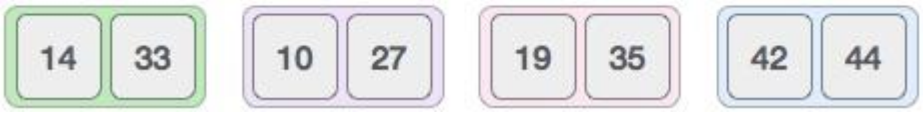


We further divide these arrays and we achieve atomic value which can no more be divided.

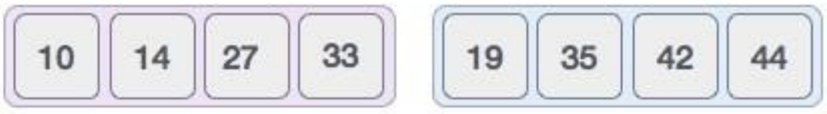


Now, we combine them in exactly the same manner as they were broken down. Please note the color codes given to these lists.

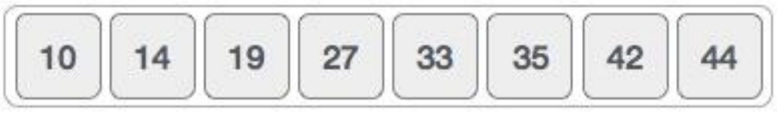
We first compare the element for each list and then combine them into another list in a sorted manner. We see that 14 and 33 are in sorted positions. We compare 27 and 10 and in the target list of 2 values we put 10 first, followed by 27. We change the order of 19 and 35 whereas 42 and 44 are placed sequentially.



In the next iteration of the combining phase, we compare lists of two data values, and merge them into a list of found data values placing all in a sorted order.



After the final merging, the list should look like this −



Now we should learn some programming aspects of merge sorting.

Algorithm

Merge sort keeps on dividing the list into equal halves until it can no more be divided. By definition, if it is only one element in the list, it is sorted. Then, merge sort combines the smaller sorted lists keeping the new list sorted too.

Step 1 − if it is only one element in the list it is already sorted, return.
Step 2 − divide the list recursively into two halves until it can no more be divided.
Step 3 − merge the smaller lists into new list in sorted order.

PROGRAM: Attach a c program of quick sort and merge sort.

QUICK SORT:

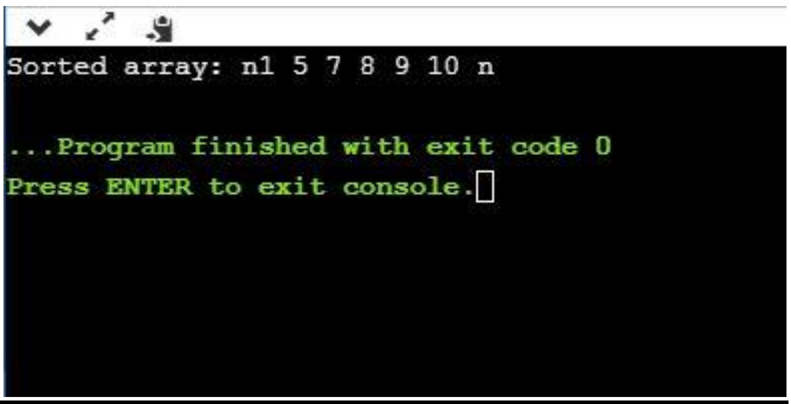
```
#include<stdio.h>
```

```
// A utility function to swap two elements
void swap(int* a, int* b)
{
    int t = *a;
    *a = *b;
    *b = t;
}

/* This function takes last element as pivot, places
   the pivot element at its correct position in sorted
    array, and places all smaller (smaller than pivot)
   to left of pivot and all greater elements to right
   of pivot */
int partition (int arr[], int low, int high)
{
    int pivot = arr[high];    // pivot
    int i = (low - 1);  // Index of smaller element

    for (int j = low; j <= high- 1; j++)
    {
        // If current element is smaller than or
        // equal to pivot
        if (arr[j] <= pivot)
        {
            i++;    // increment index of smaller element
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

/* The main function that implements QuickSort
 arr[] --> Array to be sorted,
  low  --> Starting index,
  high  --> Ending index */
void quickSort(int arr[], int low, int high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[p] is now
           at right place */
        int pi = partition(arr, low, high);

        // Separately sort elements before
        // partition and after partition
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

/* Function to print an array */
void printArray(int arr[], int size)
{
    int i;
    for (i=0; i < size; i++)
        printf("%d ", arr[i]);
    printf("n");
}

// Driver program to test above functions
int main()
{
    int arr[] = {10, 7, 8, 9, 1, 5};
    int n = sizeof(arr)/sizeof(arr[0]);
    quickSort(arr, 0, n-1);
    printf("Sorted array: n");
    printArray(arr, n);
    return 0;
}
```

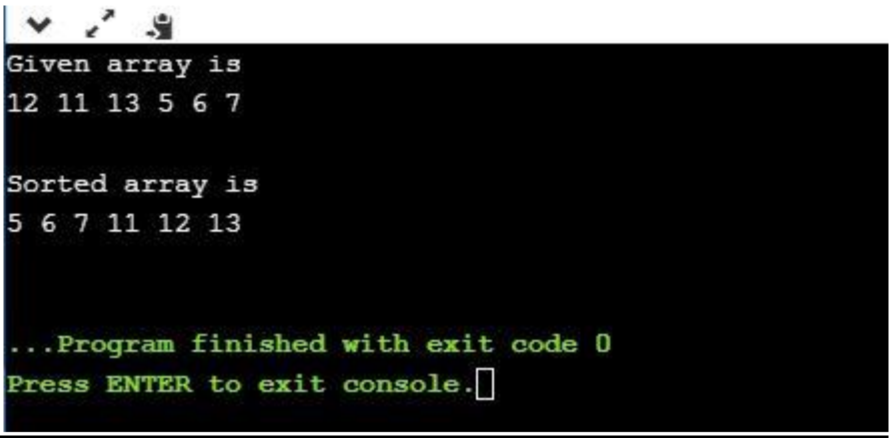
MERGE SORT:

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

// Merges two subarrays of arr[].
// First subarray is arr[l..m]
// Second subarray is arr[m+1..r]
void merge(int arr[], int l, int m, int r)
{
    int i, j, k;
    int n1 = m - l + 1;
    int n2 =  r - m;

    /* create temp arrays */
    int L[n1], R[n2];

    /* Copy data to temp arrays L[] and R[] */
    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1+ j];

    /* Merge the temp arrays back into arr[l..r]*/
    i = 0; // Initial index of first subarray
    j = 0; // Initial index of second subarray
    k = l; // Initial index of merged subarray
    while (i < n1 && j < n2)
    {
        if (L[i] <= R[j])
        {
            arr[k] = L[i];
            i++;
        }
        else
        {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    /* Copy the remaining elements of L[], if there
       are any */
    while (i < n1)
    {
        arr[k] = L[i];
        i++;
        k++;
    }

    /* Copy the remaining elements of R[], if there
       are any */
    while (j < n2)
    {
        arr[k] = R[j];
        j++;
        k++;
    }
}

/* l is for left index and r is right index of the
   sub-array of arr to be sorted */
void mergeSort(int arr[], int l, int r)
{
    if (l < r)
    {
        // Same as (l+r)/2, but avoids overflow for
        // large l and h
        int m = l+(r-l)/2;

        // Sort first and second halves
        mergeSort(arr, l, m);
        mergeSort(arr, m+1, r);

        merge(arr, l, m, r);
    }
}

/* UTILITY FUNCTIONS */
/* Function to print an array */
void printArray(int A[], int size)
{
    int i;
    for (i=0; i < size; i++)
        printf("%d ", A[i]);
    printf("\n");
}

/* Driver program to test above functions */
int main()
{
    int arr[] = {12, 11, 13, 5, 6, 7};
    int arr_size = sizeof(arr)/sizeof(arr[0]);

    printf("Given array is \n");
    printArray(arr, arr_size);

    mergeSort(arr, 0, arr_size - 1);
    printf("\nSorted array is \n");
    printArray(arr, arr_size);
    return 0;
}
```

RESULT:



```
Sorted array: n1 5 7 8 9 10 n

...Program finished with exit code 0
Press ENTER to exit console.
```



```
Given array is
12 11 13 5 6 7

Sorted array is
5 6 7 11 12 13


...Program finished with exit code 0
Press ENTER to exit console.
```

PRACTICAL – 9

OBJECTIVE: Write a program to implement Bubble Sort

Bubble sort algorithm starts by comparing the first two elements of an array and swapping if necessary, i.e., if you want to sort the elements of array in ascending order and if the first element is greater than second then, you need to swap the elements but, if the first element is smaller than second, you mustn't swap the element. Then, again second and third elements are compared and swapped if it is necessary and this process go on until last and second last element is compared and swapped. This completes the first step of bubble sort.

If there are n elements to be sorted then, the process mentioned above should be repeated n-1 times to get required result. But, for better performance, in second step, last and second last elements are not compared because, the proper element is automatically placed at last after first step. Similarly, in third step, last and second last and second last and third last elements are not compared and so on.

Example

Unsorted list:

5	2	1	4	3	7	6

<u>1st iteration:</u>

5 > 2 swap

2	5	1	4	3	7	6

5 > 1 swap

2	1	5	4	3	7	6

5 > 4 swap

2	1	4	5	3	7	6

5 > 3 swap

2	1	4	3	5	7	6

5 < 7 no swap

2	1	4	3	5	7	6

7 > 6 swap

2	1	4	3	5	6	7

2nd iteration:

2 > 1 swap

1	2	4	3	5	6	7

2 < 4 no swap

1	2	4	3	5	6	7

4 > 3 swap

1	2	3	4	5	6	7

4 < 5 no swap

1	2	3	4	5	6	7

5 < 6 no swap

1	2	3	4	5	6	7

There is no change in 3rd, 4th, 5th and 6th iteration.

Finally,

the sorted list is

1	2	3	4	5	6	7

This is the simplest technique among all sorting algorithms.

Algorithm: Sequential-Bubble-Sort (A)

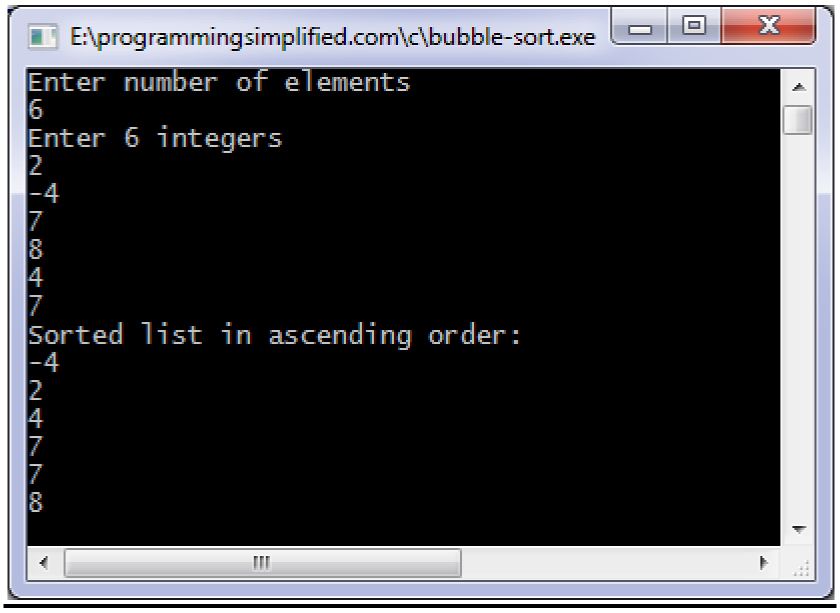
```
fori← 1 to length [A] do
for j ← length [A] down-to i +1 do
  if A[A] < A[j - 1] then
    Exchange A[j] ↔ A[j-1]
```

PROGRAM: Attach a c program of bubble sort.

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

```
void swap(int *xp, int *yp)
{
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}
void bubbleSort(int arr[], int n)
{
   int i, j;
   for (i = 0; i < n-1; i++)
       for (j = 0; j < n-i-1; j++)
           if (arr[j] > arr[j+1])
              swap(&arr[j], &arr[j+1]);
}
void printArray(int arr[], int size)
{
    int i;
    for (i=0; i < size; i++)
        printf("%d ", arr[i]);
    printf("n");
}
int main()
{
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr)/sizeof(arr[0]);
    bubbleSort(arr, n);
    printf("Sorted array: \n");

    printArray(arr, n);

    return 0;

}
```

RESULT:



E:\programmingsimplified.com\c\bubble-sort.exe

```
Enter number of elements
6
Enter 6 integers
2
-4
7
8
4
7
Sorted list in ascending order:
-4
2
4
7
7
8
```

PRACTICAL – 10

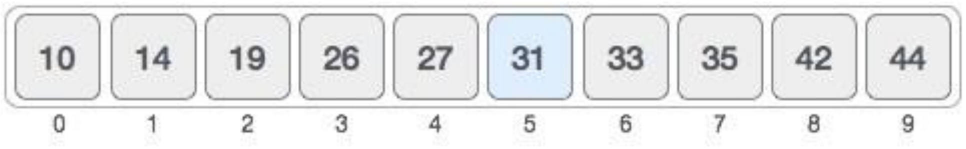
OBJECTIVE: Write a program to implement Binary Search.

Binary search is a fast search algorithm with run-time complexity of O*logn*. This search algorithm works on the principle of divide and conquer. For this algorithm to work properly the data collection should be in sorted form.

Binary search search a particular item by comparing the middle most item of the collection. If match occurs then index of item is returned. If middle item is greater than item then item is searched in sub-array to the right of the middle item other wise item is search in sub-array to the left of the middle item. This process continues on sub-array as well until the size of subarray reduces to zero.

How binary search works?

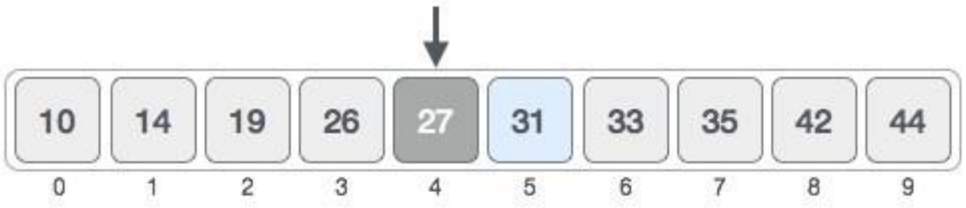
For a binary search to work, it is mandatory for the target array to be sorted. We shall learn the process of binary search with anpictorial example. The below given is our sorted array and assume that we need to search location of value 31 using binary search.



First, we shall determine the half of the array by using this formula −

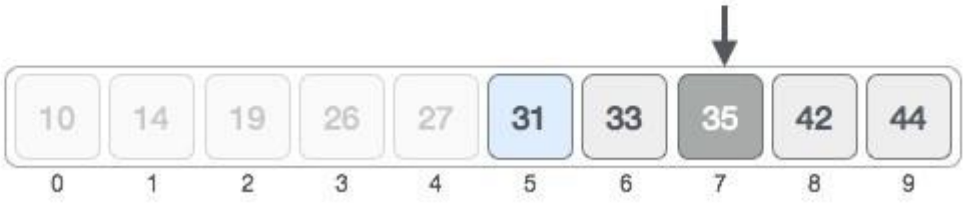
```
mid = low + (high - low) / 2
```

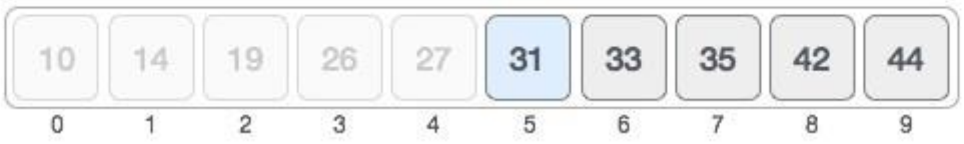
Here it is, 0 + 9 − 0 / 2 = 4 *integervalueof*4.5. So 4 is the mid of array.



Now we compare the value stored at location 4, with the value being searched i.e. 31. We find that value at location 4 is 27, which is not a match. Because value is greater than 27 and we have a sorted array so we also know that target value must be in upper portion of the array.

Our new mid is 7 now. We compare the value stored at location 7 with our target value 31.

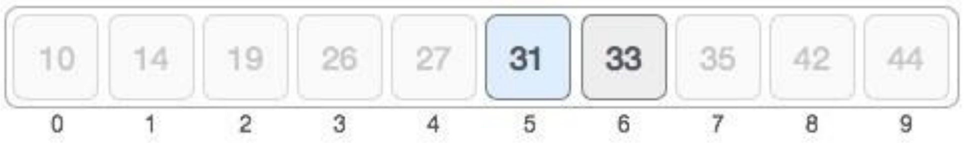




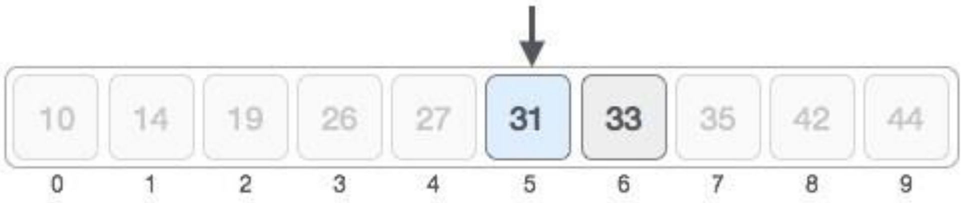
We change our low to mid + 1 and find the new mid value again.

```
low = mid + 1
mid = low + (high - low) / 2
```

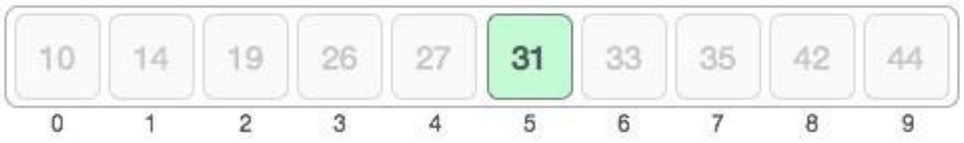
The value stored at location 7 is not a match, rather it is less that what we are looking for. So the value must be in lower part from this location.



So we calculate the mid again. This time it is 5.



We compare the value stored ad location 5 with our target value. We find that it is a match.



We conclude that the target value 31 is stored at location 5.

Binary search halves the searchable items and thus reduces the count of comparisons to be made to very less numbers.

Pseudocode

The pseudocode of binary search algorithm should look like this −

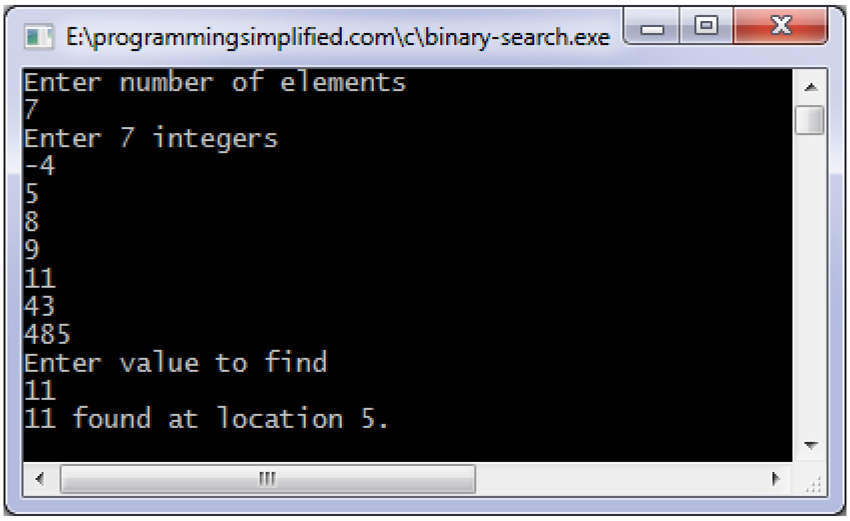
```
Procedure binary_search    A ←
sorted array    n ← size of array
x ← value ot be searched

  Set lowerBound = 1    Set
upperBound = n

  while x not found
      if    upperBound    <    lowerBound
EXIT: x does not exists.


    set midPoint = lowerBound + ( upperBound - lowerBound ) / 2


    if A[midPoint] < x          set lowerBound =
midPoint + 1


    if A[midPoint] > x
      set upperBound = midPoint - 1

    if A[midPoint] = x
      EXIT: x found at location midPoint

  end while
  end
procedure
```

PROGRAM: Attach a c program of BINARY SEARCH.

```
#include<stdio.h>

void main()

{

   int c,first,last,middle,n,search,a[100];

   printf("enter the number of elements");

   scanf("%d",&n);

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

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

   printf("enter the value to find");

   scanf("%d",&search);

   first=0;

   last=n-1;

   middle=(first+last)/2;

   while(first<=last)

   {

      if (a[middle]<search)

      first+middle+1;

      else if(a[middle]==search)

      printf("%d found at %dlocation",search,middle+1);

      break;
```

```
    }

    getch();

}
```

RESULT:



```
Enter number of elements
7
Enter 7 integers
-4
5
8
9
11
43
485
Enter value to find
11
11 found at location 5.
```