**UNIT-1**

**Introduction to Data Structure**

 Computers are data processing machines. Traditionally this data used to be database records or text files. But as processing power and storage capability of computers are increasing, and cost of computers is decreasing, computers are being used to process different types of data, like images, voice, music, movies and also biological data.What ever may be the data, the efficiency of processing the data depends on, how well the data is organized. We use different types of data structures to organize the data so that efficient algorithms can be used to process the data. Knowledge of various data structures and algorithms is essential for the software engineers.Following are the typical requirements from the data structures. The data structures should be able to support these requirements well. All the requirements may not be able to support by a single datastructure (or single data organization) that's why we need different data structures, to suit various requirements.

Ability to add data dynamically (run time) and efficiently

Ability to search for a given data

Ability to insert the data at the required position

Ability to delete the data

Ability to sort the data

Ability to access the data items around a particular data item

**Data structures are classified into following**

* Linear Data structures
* Non-Linear Data structures

Linear Data structures

Arrays

Structures

Unions

Bit-fields

Enumeration

Linked Lists

Stacks

Queues

Non-Linear Data structures

Trees

Graphs

**Data structures can also be classified into following**

* Static Data structures
* Dynamic Data structures.

Static Data structures

Arrays.

Structures.

Unions.

Bit-fields.

Enumeration.

Dynamic Data structures

Linked Lists.

Stacks using arrays and linked lists.

Queues using arrays and linked lists.

Trees using arrays and linked lists.

Graphs using arrays and linked lists.

Arrays and linked lists are the basic building blocks. All the other data structures will be built using

either Arrays or Linked lists or both of them. So one should be very comfortable with these basic

building blocks.

### What is Time Complexity?

Time complexity of an algorithm signifies the total time required by the program to run till its completion.

The time complexity of algorithms is most commonly expressed using the **big O notation**. It's an asymptotic notation to represent the time complexity. We will study about it in detail in the next tutorial.

Time Complexity is most commonly estimated by counting the number of elementary steps performed by any algorithm to finish execution. Like in the example above, for the first code the loop will run n number of times, so the time complexity will be n atleast and as the value of n will increase the time taken will also increase. While for the second code, time complexity is constant, because it will never be dependent on the value of n, it will always give the result in 1 step.

And since the algorithm's performance may vary with different types of input data, hence for an algorithm we usually use the **worst-case Time complexity** of an algorithm because that is the maximum time taken for any input size.

Time Complexity of Algorithms

For any defined problem, there can be N number of solution. This is true in general. If I have a problem and I discuss about the problem with all of my friends, they will all suggest me different solutions. And I am the one who has to decide which solution is the best based on the circumstances.

Similarly for any problem which must be solved using a program, there can be infinite number of solutions. Let's take a simple example to understand this. Below we have two different algorithms to find square of a number(for some time, forget that square of any number n is n\*n):

One solution to this problem can be, running a loop for n times, starting with the number n and adding n to it, every time.

/\*

 we have to calculate the square of n

\*/

for i=1 to n

 do n = n + n

// when the loop ends n will hold its square

return n

# Space Complexity

Whenever a solution to a problem is written some memory is required to complete. For any algorithm memory may be used for the following:

1. Variables (This include the constant values, temporary values)
2. Program Instruction
3. Execution

***Space complexity****is the amount of memory used by the algorithm (including the input values to the algorithm) to execute and produce the result.*

Sometime **Auxiliary Space** is confused with Space Complexity. But Auxiliary Space is the extra space or the temporary space used by the algorithm during it's execution.

**Space Complexity** = **Auxiliary Space + Input space**

**Memory Usage while Execution**

While executing, algorithm uses memory space for three reasons:

1. **Instruction Space**

It's the amount of memory used to save the compiled version of instructions.

1. **Environmental Stack**

Sometimes an algorithm(function) may be called inside another algorithm(function). In such a situation, the current variables are pushed onto the system stack, where they wait for further execution and then the call to the inside algorithm(function) is made.

For example, If a function A() calls function B() inside it, then all th variables of the function A() will get stored on the system stack temporarily, while the function B() is called and executed inside the funciton A()

.

1. **Data Space**

Amount of space used by the variables and constants.

But while calculating the **Space Complexity** of any algorithm, we usually consider only **Data Space** and we neglect the **Instruction Space** and **Environmental Stack**.

**Arrays**

 Arrays are used to store multiple elements of same type. Each element of this array will be typically a structure. Arrays can be implemented using either static (compile time) allocation or dynamic (runtime) allocation. In the case of static allocation array will be defined with required size. In the case of dynamic allocation, memory for the array will be allocated during run time using malloc () function. The address of this allocated memory will be stored in a pointer variable. Then onwards this pointer can be used as array.

**Operations on arrays**

Following are the typical operations on arrays. One needs to get familiar with these operations as much as possible.

Adding an element to the end of array

Adding an element at the beginning of array

Inserting an element in a sorted array

Deleting an element at the given index

linear search and binary search on arrays

Sorting techniques

 An array is a collection of variables that are of similar data types and are referred by a common name. A specific element in an array is accessed by a particular index of that array. Arrays in Java work differently as compared to C++. The main topic of our discussion is the difference between One-dimension and Two-Dimension array. A one-dimensional array is a list of variables with the same datatype, whereas the two-Dimensional array is ‘array of arrays’ having similar data types. C++ do not have bound checking on arrays whereas, Java have strict bound checking on arrays.

**One-Dimensional (1D) Array**

One-Dimensional or Single-Dimensional array is considered as the” list of variables of similar data types”, and each variable can be distinctly accessed by specifying its index in square brackets preceded by the name of that array.

In C++, the declaration of an array variable with size is enough to allocate space for them in memory. In Java, this is achieved in two steps. First, you must declare a variable of the desired type. Second, you must allocate the memory to hold the array using ‘new’ and assign it to declared array variable. Hence, arrays are dynamically allocated .

// Program to take 5 values from the user and store them in an array

// Print the elements stored in the array

#include <stdio.h>

int main() {

 int values[5];

 printf("Enter 5 integers: ");

 // taking input and storing it in an array

 for(int i = 0; i < 5; ++i) {

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

 }

 printf("Displaying integers: ");

 // printing elements of an array

 for(int i = 0; i < 5; ++i) {

 printf("%d\n", values[i]);

 }

 return 0;

}

### Definition of Two-Dimensional Array (2-D array)

Both C++ and Java support multidimensional array. One of the simplest forms of a multidimensional array is a, two-dimensional array or 2-D array. A two-Dimensional array can be considered as ‘array of arrays’  or ‘array of one-dimensional arrays’. To declare the two-dimensional array variable, we have to specify the array name followed by two square brackets where the second index is the second set of square brackets.

A two-dimensional array is stored in the form of the row-column matrix, where the first index indicate the row and second index indicates the column. The second or the rightmost index of an array changes very fastly as compared to first or left-most index while accessing the elements of an array.

In C++, the two-dimensional array is declared as;

//declaration in C++

type variable\_name[size1][size2];

\*\*\*\*\*\*// C program to store temperature of two cities of a week and display it.

 #include <stdio.h>

 const int CITY = 2;

 const int WEEK = 7;

 int main()

{

 int temperature[CITY][WEEK];

 // Using nested loop to store values in a 2d array

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

 {

 for (int j = 0; j < WEEK; ++j)

 {

 printf("City %d, Day %d: ", i + 1, j + 1);

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

 }

 }

 printf("\nDisplaying values: \n\n");

 // Using nested loop to display vlues of a 2d array

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

 {

 for (int j = 0; j < WEEK; ++j)

 {

 printf("City %d, Day %d = %d\n", i + 1, j + 1, temperature[i][j]);

 }

 }

 return 0;

}

**Multi-dimensional array**

A **multi-dimensional array** is an array that has more than one dimension. It is an array of arrays; an array that has multiple levels. The simplest multi-dimensional array is the **2D array**, or two-dimensional array. It's technically an array of arrays, as you will see in the code. A 2D array is also called a **matrix**, or a table of rows and columns.

Declaring a multi-dimensional array is similar to the one-dimensional arrays. For a 2D array, we need to tell C that we have 2 dimensions.



// C Program to store and print 12 values entered by the user

#include <stdio.h>

int main()

{

 int test[2][3][2];

 printf("Enter 12 values: \n");

 for (int i = 0; i < 2; ++i)

 {

 for (int j = 0; j < 3; ++j)

 {

 for (int k = 0; k < 2; ++k)

 {

 scanf("%d", &test[i][j][k]);

 }

 }

 }

 // Printing values with proper index.

 printf("\nDisplaying values:\n");

 for (int i = 0; i < 2; ++i)

 {

 for (int j = 0; j < 3; ++j)

 {

 for (int k = 0; k < 2; ++k)

 {

 printf("test[%d][%d][%d] = %d\n", i, j, k, test[i][j][k]);

 }

 }

 }

 return 0;

}

***STRUCTURE***

Arrays allow to define type of variables that can hold several data items of the same kind. Similarly **structure** is another user defined data type available in C that allows to combine data items of different kinds.

Structures are used to represent a record. Suppose you want to keep track of your books in a library. You might want to track the following attributes about each book −

* Title
* Author
* Subject
* Book ID

## Defining a Structure

To define a structure, you must use the **struct** statement. The struct statement defines a new data type, with more than one member. The format of the struct statement is as follows −

struct [structure tag] {

 member definition;

 member definition;

 ...

 member definition;

} [one or more structure variables];

The **structure tag** is optional and each member definition is a normal variable definition, such as int i; or float f; or any other valid variable definition. At the end of the structure's definition, before the final semicolon, you can specify one or more structure variables but it is optional. Here is the way you would declare the Book structure −

struct Books {

 char title[50];

 char author[50];

 char subject[100];

 int book\_id;

} book;

## Accessing Structure Members

To access any member of a structure, we use the **member access operator (.)**. The member access operator is coded as a period between the structure variable name and the structure member that we wish to access. You would use the keyword **struct** to define variables of structure type. The following example shows how to use a structure in a program –

#include <stdio.h>

#include <string.h>

struct Books {

 char title[50];

 char author[50];

 char subject[100];

 int book\_id;

};

int main( ) {

 struct Books Book1; /\* Declare Book1 of type Book \*/

 struct Books Book2; /\* Declare Book2 of type Book \*/

 /\* book 1 specification \*/

 strcpy( Book1.title, "C Programming");

 strcpy( Book1.author, "Nuha Ali");

 strcpy( Book1.subject, "C Programming Tutorial");

 Book1.book\_id = 6495407;

 /\* book 2 specification \*/

 strcpy( Book2.title, "Telecom Billing");

 strcpy( Book2.author, "Zara Ali");

 strcpy( Book2.subject, "Telecom Billing Tutorial");

 Book2.book\_id = 6495700;

 /\* print Book1 info \*/

 printf( "Book 1 title : %s\n", Book1.title);

 printf( "Book 1 author : %s\n", Book1.author);

 printf( "Book 1 subject : %s\n", Book1.subject);

 printf( "Book 1 book\_id : %d\n", Book1.book\_id);

 /\* print Book2 info \*/

 printf( "Book 2 title : %s\n", Book2.title);

 printf( "Book 2 author : %s\n", Book2.author);

 printf( "Book 2 subject : %s\n", Book2.subject);

 printf( "Book 2 book\_id : %d\n", Book2.book\_id);

 return 0;

}

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

Book 1 title : C Programming

Book 1 author : Nuha Ali

Book 1 subject : C Programming Tutorial

Book 1 book\_id : 6495407

Book 2 title : Telecom Billing

Book 2 author : Zara Ali

Book 2 subject : Telecom Billing Tutorial

Book 2 book\_id : 6495700

**Pointers**

 Pointers in C are easy and fun to learn. Some C programming tasks are performed more easily with pointers, and other tasks, such as dynamic memory allocation, cannot be performed without using pointers. So it becomes necessary to learn pointers to become a perfect C programmer. Let's start learning them in simple and easy steps.

As you know, every variable is a memory location and every memory location has its address defined which can be accessed using ampersand (&) operator, which denotes an address in memory. Consider the following example, which prints the address of the variables defined −

#include <stdio.h>

int main () {

 int var1;

 char var2[10];

 printf("Address of var1 variable: %x\n", &var1 );

 printf("Address of var2 variable: %x\n", &var2 );

 return 0;

}

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

Address of var1 variable: bff5a400

Address of var2 variable: bff5a3f6

## 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 () {

 int var = 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 );

 /\* address stored in pointer variable \*/

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

 /\* access the value using the pointer \*/

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

 return 0;

}

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

# ****Memory Allocations in Data Structures****

**Memory allocation** is the process of setting aside sections of memory in a program to be used to store variables, and instances of structures and classes.

There are two types of memory allocations possible in C:

1. Compile-time or Static allocation.
2. Run-time or Dynamic allocation (using pointers).

### ****Compile-time or Static allocation****

* **Static memory allocation** allocated by the compiler. Exact size and type of memory must be known at compile time.​​​​​​​

int x, y;

float a[5];

When the first statement is encountered, the compiler will allocate two bytes to each variables x and y. The second statement results into the allocction of 20 bytes to the array a (5\*4, where there are five elements and each element of float type tales four bytes). Note that as there is no bound checking in C for array boundaries, i.e., if you have declared an array of five elements, as above and by mistake you are intending to read more than five values in the array **a,** it will still work without error. For example you are reading the above array as follows :

for ( i = 0 ; i < 10 ; i++)

{

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

}

### ****Run-time or Dynamic allocation****

* Dynamic memory allocation is when an executing program requests that the operating system give it a block of main memory. The program then uses this memory for some purpose. Usually the purpose is to add a node to a data structure. In object-oriented languages, dynamic memory allocation is used to get the memory for a new object.
* The memory comes from above the static part of the data segment. Programs may request memory and may also return previously dynamically allocated memory. Memory may be returned whenever it is no longer needed. Memory can be returned in any order without any relation to the order in which it was allocated. The heap may develop "holes" where previously allocated memory has been returned between blocks of memory still in use.
* A new dynamic request for memory might return a range of addresses out of one of the holes. But it might not use up all the hole, so further dynamic requests might be satisfied out of the original hole.

​​

C provides the following dynamic allocation and de-allocation functions :

* + malloc( )
* calloc( )
* free( )
* realloc( )

**​​​​​​​The Malloc( ) Function**

The malloc( ) function allocates a block of memory in bytes. The user should explicitly give the block sixe it requires of the use. The malloc( ) is like a request to the RAM of the system to allocate memoty.

**Syntax:-**

malloc (number of elements \* size of each element) ;

**Example:-**

int \*ptr ;

ptr = malloc (10 \*sizeof(int))

### ****The Calloc( ) Function****

This function works exactly similar to malloc( ) function except for the fact that it needs two arguments as against one argument required by malloc( ).

**Example:-**

int \*ptr ;

ptr = (int \*) calloc (10, 2);

**The Free( ) Function**

The free( ) function is used to de-allocate the previously allocated memory using malloc( ) or calloc( ) functions.

**Syntax:-**

free (ptr\_var); // Where ptr\_var is the pointer in which the address of the allocated memory block is assigned.

**The Realloc( ) Function**

This function is used to resize the size of memory block, which is already allocated. It found use of in two situations :

* If the allocated memory block is insufficient for current application.
* If the allocated memory is much more than what is required by the current application.

**Syntax:-**

ptr\_var = realloc (ptr\_var, new\_size);

 **Linked list**

A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations. The elements in a linked list are linked using pointers as shown in the below image:


In simple words, a linked list consists of nodes where each node contains a data field and a reference(link) to the next node in the list.

**Topics :**

* [Singly Linked List](https://www.geeksforgeeks.org/data-structures/linked-list/#singlyLinkedList)
* [Circular Linked List](https://www.geeksforgeeks.org/data-structures/linked-list/#circularLinkedList)
* [Doubly Linked List](https://www.geeksforgeeks.org/data-structures/linked-list/#doublyLinkedList)

# Singly Linked List

A **linked list** is a way to store a collection of elements. Like an array these can be character or integers. Each element in a linked list is stored in the form of a **node**.

**Node**:

A node is a collection of two sub-elements or parts. A **data** part that stores the element and a **next** part that stores the link to the next node.

**Declaring a Linked list** :

In C language, a linked list can be implemented using structure and pointers .

struct LinkedList{

 int data;

 struct LinkedList \*next;

 };

The above definition is used to create every node in the list. The **data** field stores the element and the **next** is a pointer to store the address of the next node.

Noticed something unusual with next?

In place of a data type, **struct LinkedList** is written before next. That's because its a **self-referencing pointer**. It means a pointer that points to whatever it is a part of. Here **next** is a part of a node and it will point to the next node.

**Creating a Node**:

Let's define a data type of struct LinkedListto make code cleaner.

typedef struct LinkedList \*node; //Define node as pointer of data type struct LinkedList

node createNode(){

 node temp; // declare a node

 temp = (node)malloc(sizeof(struct LinkedList)); // allocate memory using malloc()

 temp->next = NULL;// make next point to NULL

 return temp;//return the new node

}

**typedef** is used to define a data type in C.

**malloc()** is used to dynamically allocate a single block of memory in C, it is available in the header file stdlib.h.

**sizeof()** is used to determine size in bytes of an element in C. Here it is used to determine size of each node and sent as a parameter to malloc.

## Basic Operations

Following are the basic operations supported by a list.

* **Insertion** − add an element at the beginning of the list.
* **Deletion** − delete an element at the beginning of the list.
* **Display** − displaying complete list.
* **Search** − search an element using given key.
* **Delete** − delete an element using given key.

**NO.1 Create a Simple Linked list and Display it.**

#include <stdio.h>

#include <stdlib.h>

struct node

{

 int num; //Data of the node

 struct node \*nextptr; //Address of the next node

}\*stnode;

void createNodeList(int n); // function to create the list

void displayList(); // function to display the list

int main()

{

 int n;

 printf("\n\n Linked List : To create and display Singly Linked List :\n");

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

 printf(" Input the number of nodes : ");

 scanf("%d", &n);

 createNodeList(n);

 printf("\n Data entered in the list : \n");

 displayList();

 return 0;

}

void createNodeList(int n)

{

 struct node \*fnNode, \*tmp;

 int num, i;

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

 if(stnode == NULL) //check whether the fnnode is NULL and if so no memory allocation

 {

 printf(" Memory can not be allocated.");

 }

 else

 {

// reads data for the node through keyboard

 printf(" Input data for node 1 : ");

 scanf("%d", &num);

 stnode->num = num;

 stnode->nextptr = NULL; // links the address field to NULL

 tmp = stnode;

// Creating n nodes and adding to linked list

 for(i=2; i<=n; i++)

 {

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

 if(fnNode == NULL)

 {

 printf(" Memory can not be allocated.");

 break;

 }

 else

 {

 printf(" Input data for node %d : ", i);

 scanf(" %d", &num);

 fnNode->num = num; // links the num field of fnNode with num

 fnNode->nextptr = NULL; // links the address field of fnNode with NULL

 tmp->nextptr = fnNode; // links previous node i.e. tmp to the fnNode

 tmp = tmp->nextptr;

 }

 }

 }

}

void displayList()

{

 struct node \*tmp;

 if(stnode == NULL)

 {

 printf(" List is empty.");

 }

 else

 {

 tmp = stnode;

 while(tmp != NULL)

 {

 printf(" Data = %d\n", tmp->num); // prints the data of current node

 tmp = tmp->nextptr; // advances the position of current node

 }

 }

}

Sample Output:

 Linked List : To create and display Singly Linked List :

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

 Input the number of nodes : 3

 Input data for node 1 : 5

 Input data for node 2 : 6

 Input data for node 3 : 7

 Data entered in the list :

 Data = 5

 Data = 6

 Data = 7

**NO.2 C program to insert an element at any position in a linked list**

#include <stdio.h>

#include <stdlib.h>

#include <conio.h>

/\* Node Stucture \*/

typedef struct node\_t {

 int data;

 struct node\_t \*next;

} Node;

/\* Function Declarations \*/

Node \* insert\_top(int, Node \*);

Node \* insert\_bottom(int, Node \*);

Node \* insert\_after(int, int, Node \*);

Node \* insert\_before(int, int, Node \*);

void print(Node \*);

int count(Node \*);

/\* Add a new node to the top of a list \*/

Node \* insert\_top(int num, Node \*head) {

 Node \*new\_node;

 new\_node = (Node \*) malloc(sizeof(Node));

 new\_node->data = num;

 new\_node->next= head;

 head = new\_node;

return head;

}

/\* Add a new node to the bottom of a list \*/

Node \* insert\_bottom(int num, Node \*head) {

 Node \*current\_node = head;

 Node \*new\_node;

 while ( current\_node != NULL && current\_node->next != NULL) {

 current\_node = current\_node->next;

 }

 new\_node = (Node \*) malloc(sizeof(Node));

 new\_node->data = num;

 new\_node->next= NULL;

 if (current\_node != NULL)

 current\_node->next = new\_node;

 else

 head = new\_node;

return head;

}

/\* Add a new node after an element in the list \*/

Node \* insert\_after(int num, int prev\_num, Node \*head) {

 Node \*current\_node = head;

 Node \*new\_node;

 while ( current\_node->data != prev\_num) {

 current\_node = current\_node->next;

 }

 new\_node = (Node \*) malloc(sizeof(Node));

 new\_node->data = num;

 new\_node->next= current\_node->next;

 current\_node->next = new\_node;

return head;

}

/\* Add a new node before an element in the list \*/

 Node \* insert\_before(int num, int next\_num, Node \*head) {

 Node \*current\_node = head;

 Node \*new\_node;

 while ( current\_node->next->data != next\_num) {

 current\_node = current\_node->next;

 }

 new\_node = (Node \*) malloc(sizeof(Node));

 new\_node->data = num;

 new\_node->next= current\_node->next;

 current\_node->next = new\_node;

return head;

}

/\* Print all the elements in the linked list \*/

void print(Node \*head) {

 Node \*current\_node = head;

 while ( current\_node != NULL) {

 printf("%d ", current\_node->data);

 current\_node = current\_node->next;

 }

}

/\* Program main \*/

int main()

{

 Node \*head = NULL;

 int num, prev\_num, next\_num;

 int option;

 char \* temp;

 char ch;

 /\* Display Menu \*/

 while(1) {

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

 printf("\n \* Linked list operations: \*\n");

 printf("\n \* 1. Insert at the top of list \*\n");

 printf("\n \* 2. Insert at bottom of list \*\n");

 printf("\n \* 3. Insert after an element \*\n");

 printf("\n \* 4. Insert before an element \*\n");

 printf("\n \* 5. Show all elements \*\n");

 printf("\n \* 6. Quit \*\n");

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

 printf("\n Choose an option [1-5] : ");

 if (scanf("%d", &option) != 1) {

 printf(" \*Error: Invalid input. Try again.\n");

 scanf("%s", &temp); /\*clear input buffer \*/

 continue;

 }

 switch (option) {

 case 1: /\* Add to top\*/

 printf(" Enter a number to insert : ");

 if (scanf("%d", &num) != 1) {

 printf(" \*Error: Invalid input.\n");

 scanf("%s", &temp); /\*clear input buffer \*/

 continue;

 }

 head = insert\_top(num, head);

 printf("Number %d added to the top of the list", num);

 printf("\nPress any key to continue...");

 getch();

 break;

 case 2: /\* add to bottom \*/

 printf(" Enter a number to insert : ");

 if (scanf("%d", &num) != 1) {

 printf(" \*Error: Invalid input. \n");

 scanf("%s", &temp);

 continue;

 }

 head = insert\_bottom(num, head);

 printf("%d added to the bottom of the list", num);

 printf("\nPress any key to continue...");

 getch();

 break;

 case 3: /\* Insert After \*/

 printf(" Enter a number to insert : ");

 if (scanf("%d", &num) != 1) {

 printf(" \*Error: Invalid input.\n");

 scanf("%s", &temp);

 continue;

 }

 printf(" After which number do you want to insert : ");

 if (scanf("%d", &prev\_num) != 1) {

 printf(" \*Error: Invalid input.\n");

 scanf("%s", &temp);

 continue;

 }

 if (head != NULL) {

 head = insert\_after(num, prev\_num, head);

 printf("%d is inserted after %d", num, prev\_num);

 }else {

 printf("The list is empty", num, prev\_num);

 }

 printf("\nPress any key to continue...");

 getch();

 break;

 case 4: /\* Insert Before \*/

 printf(" Enter a number to insert : ");

 if (scanf("%d", &num) != 1) {

 printf(" \*Error: Invalid input. \n");

 scanf("%s", &temp);

 continue;

 }

 printf(" Before which number do you want to insert : ");

 if (scanf("%d", &prev\_num) != 1) {

 printf(" \*Error: Invalid input.\n");

 scanf("%s", &temp);

 continue;

 }

 if (head != NULL) {

 head = insert\_before(num, prev\_num, head);

 printf("Number %d inserted before %d", num, prev\_num);

 }else {

 printf("The list is empty", num, prev\_num);

 }

 printf("\nPress any key to continue...");

 getch();

 break;

 case 5: /\* Show all elements \*/

 printf("\nElements in the list: \n [ ");

 print(head);

 printf("]\n\nPress any key to continue...");

 getch();

 break;

 case 6: /\* Exit \*/

 return(0);

 break;

 default:

 printf("Invalid Option. Please Try again.");

 getch();

 } /\* End of Switch \*/

 } /\* End of While \*/

return(0);

}

# C:\Users\Jagannath\Desktop\Untitled.jpg

# 3.Write a program to insert, delete and Search a node in single linked list.

# # include<stdio.h>

# # include<stdlib.h>

# typedef struct Node

# {

#  int data;

#  struct Node \*next;

# }

# node;

# void insert(node \*pointer, int data)

# {

#

#  while(pointer->next!=NULL)

#  {

#  pointer = pointer -> next;

#  }

#  pointer->next = (node \*)malloc(sizeof(node));

#  pointer = pointer->next;

#  pointer->data = data;

#  pointer->next = NULL;

# }

# int find(node \*pointer, int key)

# {

#  pointer = pointer -> next; //First node is dummy node.

#  {

#  if(pointer->data == key) //key is found.

#  {

#  return 1;

#  }

#  pointer = pointer -> next;//Search in the next node.

#  }

#  /\*Key is not found \*/

#  return 0;

# }

# void delet(node \*pointer, int data)

# {

#

#  {

#  pointer = pointer -> next;

#  }

#  if(pointer->next==NULL)

#  {

#  printf("Element %d is not present in the list\n",data);

#  return;

#  }

#

#  return;

# }

# void print(node \*pointer)

# {

#  if(pointer==NULL)

#  {

#  return;

#  }

#  printf("%d ",pointer->data);

#  print(pointer->next);

# }

# int main()

# {

#

#  node \*start,\*temp;

#  start = (node \*)malloc(sizeof(node));

#  temp = start;

#  temp -> next = NULL;

#

#  printf("1. Insert\n");

#  printf("2. Delete\n");

#  printf("3. Print\n");

#  printf("4. Find\n");

#  while(1)

#  {

#  int query;

#  scanf("%d",&query);

#  if(query==1)

#  {

#  int data;

#  printf("\nEnter the data: ");

#  scanf("%d",&data);

#  insert(start,data);

#  }

#  else if(query==2)

#  {

#  int data;

#  scanf("%d",&data);

#  delet(start,data);

#  }

#  else if(query==3)

#  {

#  printf("The list is ");

#  print(start->next);

#  printf("\n");

#  }

#  else if(query==4)

#  {

#  int data;

#  scanf("%d",&data);

#  int status = find(start,data);

#  if(status)

#  {

#  printf("Element Found\n");

#  }

#  else

#  {

#  printf("Element Not Found\n"); } } } }

# 3.C program to create and traverse a Linked List

# #include <stdio.h>

# #include <stdlib.h>

# /\* Structure of a node \*/

# struct node {

#  int data; // Data

#  struct node \*next; // Address

# }\*head;

# void createList(int n);

# void traverseList();

# int main()

# {

#  int n;

#  printf("Enter the total number of nodes: ");

#  scanf("%d", &n);

#  createList(n);

#  printf("\nData in the list \n");

#  traverseList();

#  return 0;

# }

# void createList(int n)

# {

#  struct node \*newNode, \*temp;

#  int data, i;

#  head = (struct node \*)malloc(sizeof(struct node));

#  if(head == NULL)

#  {

#  printf("Unable to allocate memory.");

#  exit(0);

#  }

#  printf("Enter the data of node 1: ");

#  scanf("%d", &data);

#  head->data = data; // Link data field with data

#  head->next = NULL; // Link address field to NULL

#  temp = head;

#  for(i=2; i<=n; i++)

#  {

#  newNode = (struct node \*)malloc(sizeof(struct node));

#  if(newNode == NULL)

#  {

#  printf("Unable to allocate memory.");

#  break;

#  }

#  printf("Enter the data of node %d: ", i);

#  scanf("%d", &data);

#  newNode->data = data; // Link data field of newNode

#  newNode->next = NULL; // Make sure new node points to NULL

#  temp->next = newNode; // Link previous node with newNode

#  temp = temp->next; // Make current node as previous node

#  }

# }

# void traverseList()

# {

#  struct node \*temp;

#  // Return if list is empty

#  if(head == NULL)

#  {

#  printf("List is empty.");

#  return;

#  }

#

#  temp = head;

#  while(temp != NULL)

#  {

#  printf("Data = %d\n", temp->data); // Print data of current node

#  temp = temp->next; // Move to next node

#  }

# }

# OUTPUT

# Enter the total number of nodes: 5

# Enter the data of node 1: 10

# Enter the data of node 2: 20

# Enter the data of node 3: 30

# Enter the data of node 4: 40

# Enter the data of node 5: 50

# Data in the list

# Data = 10

# Data = 20

# Data = 30

# Data = 40

# Data = 50

**Doubly Linked**

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

# C:\Users\Jagannath\Desktop\doubly_linked_list.jpg

# Basic Operations

Following are the basic operations supported by a list.

* **Insertion** − Adds an element at the beginning of the list.
* **Deletion** − Deletes an element at the beginning of the list.
* **Insert Last** − Adds an element at the end of the list.
* **Delete Last** − Deletes an element from the end of the list.
* **Insert After** − Adds an element after an item of the list.
* **Delete** − Deletes an element from the list using the key.
* **Display forward** − Displays the complete list in a forward manner.
* **Display backward** − Displays the complete list in a backward manner.

# (All the above Program Refer From Sir or Inter Net)

# Sparse Matrix

# What is Sparse Matrix?

In computer programming, a matrix can be defined with a 2-dimensional array. Any array with 'm' columns and 'n' rows represent a m X n matrix. There may be a situation in which a matrix contains more number of ZERO values than NON-ZERO values. Such matrix is known as sparse matrix.

When a sparse matrix is represented with a 2-dimensional array, we waste a lot of space to represent that matrix. For example, consider a matrix of size 100 X 100 containing only 10 non-zero elements. In this matrix, only 10 spaces are filled with non-zero values and remaining spaces of the matrix are filled with zero. That means, totally we allocate 100 X 100 X 2 = 20000 bytes of space to store this integer matrix. And to access these 10 non-zero elements we have to make scanning for 10000 times. To make it simple we use the following sparse matrix representation.

## Sparse Matrix Representations

A sparse matrix can be represented by using TWO representations, those are as follows...

1. Triplet Representation (Array Representation)
2. Linked Representation

### Triplet Representation (Array Representation)

In this representation, we consider only non-zero values along with their row and column index values. In this representation, the 0throw stores the total number of rows, total number of columns and the total number of non-zero values in the sparse matrix.

For example, consider a matrix of size 5 X 6 containing 6 number of non-zero values. This matrix can be represented as shown in the image...



In above example matrix, there are only 6 non-zero elements ( those are 9, 8, 4, 2, 5 & 2) and matrix size is 5 X 6. We represent this matrix as shown in the above image. Here the first row in the right side table is filled with values 5, 6 & 6 which indicates that it is a sparse matrix with 5 rows, 6 columns & 6 non-zero values. The second row is filled with 0, 4, & 9 which indicates the non-zero value 9 is at the 0th-row 4th column in the Sparse matrix. In the same way, the remaining non-zero values also follow a similar pattern.

**Implementation of Array Representation of Sparse Matrix using C++**

#include<iostream>

using namespace std;

int main()

{ // sparse matrix of class 5x6 with 6 non-zero values

 int sparseMatrix[5][6] =

 {

 {0 , 0 , 0 , 0 , 9, 0 },

 {0 , 8 , 0 , 0 , 0, 0 },

 {4 , 0 , 0 , 2 , 0, 0 },

 {0 , 0 , 0 , 0 , 0, 5 },

 {0 , 0 , 2 , 0 , 0, 0 }

 };

 // Finding total non-zero values in the sparse matrix

 int size = 0;

 for (int row = 0; row < 5; row++)

 for (int column = 0; column < 6; column++)

 if (sparseMatrix[row][column] != 0)

 size++;

 // Defining result Matrix

 int resultMatrix[3][size];

 // Generating result matrix

 int k = 0;

 for (int row = 0; row < 5; row++)

 for (int column = 0; column < 6; column++)

 if (sparseMatrix[row][column] != 0)

 {

 resultMatrix[0][k] = row;

 resultMatrix[1][k] = column;

 resultMatrix[2][k] = sparseMatrix[row][column];

 k++;

 }

 // Displaying result matrix

 cout<<"Triplet Representation : "<<endl;

 for (int row=0; row<3; row++)

 {

 for (int column = 0; column<size; column++)

 cout<<resultMatrix[row][column]<<" ";

 cout<<endl;

 }

 return 0;

}

# UNIT-2

# What is a Stack?

Stack is a linear data structure in which the insertion and deletion operations are performed at only one end. In a stack, adding and removing of elements are performed at a single position which is known as "**top**". That means, a new element is added at top of the stack and an element is removed from the top of the stack. In stack, the insertion and deletion operations are performed based on **LIFO**(Last In First Out) principle.



In a stack, the insertion operation is performed using a function called **"push"** and deletion operation is performed using a function called **"pop"**.

In the figure, PUSH and POP operations are performed at a top position in the stack. That means, both the insertion and deletion operations are performed at one end (i.e., at Top)

A stack data structure can be defined as follows...

**Stack is a linear data structure in which the operations are performed based on LIFO principle.**

Stack can also be defined as

**"A Collection of similar data items in which both insertion and deletion operations are performed based on LIFO principle".**

# Example

If we want to create a stack by inserting 10,45,12,16,35 and 50. Then 10 becomes the bottom-most element and 50 is the topmost element. The last inserted element 50 is at Top of the stack as shown in the image below...



## Operations on a Stack

The following operations are performed on the stack...

1. **Push (To insert an element on to the stack)**
2. **Pop (To delete an element from the stack)**
3. **Display (To display elements of the stack)**

Stack data structure can be implemented in two ways. They are as follows...

1. **Using Array**
2. **Using Linked List**

When a stack is implemented using an array, that stack can organize an only limited number of elements. When a stack is implemented using a linked list, that stack can organize an unlimited number of elements. A stack data structure can be implemented using a one-dimensional array. But stack implemented using array stores only a fixed number of data values. This implementation is very simple. Just define a one dimensional array of specific size and insert or delete the values into that array by using **LIFO principle** with the help of a variable called **'top'**. Initially, the top is set to -1. Whenever we want to insert a value into the stack, increment the top value by one and then insert. Whenever we want to delete a value from the stack, then delete the top value and decrement the top value by one.

# Stack Operations using Array

# A stack can be implemented using array as follows...

# Before implementing actual operations, first follow the below steps to create an empty stack.

* **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 the **functions** used in stack implementation.
* **Step 3 -**Create a one dimensional array with fixed size (**int stack[SIZE]**)
* **Step 4 -**Define a integer variable **'top'** and initialize with **'-1'**. (**int top = -1**)
* **Step 5 -**In main method, display menu with list of operations and make suitable function calls to perform operation selected by the user on the stack.

## push(value) - Inserting value into the stack

In a stack, push() is a function used to insert an element into the stack. In a stack, the new element is always inserted at **top** position. Push function takes one integer value as parameter and inserts that value into the stack. We can use the following steps to push an element on to the stack...

* **Step 1 -**Check whether **stack** is **FULL**. (**top == SIZE-1**)
* **Step 2 -**If it is **FULL**, then display **"Stack is FULL!!! Insertion is not possible!!!"**and terminate the function.
* **Step 3 -**If it is **NOT FULL**, then increment **top** value by one (**top++**) and set stack[top] to value (**stack[top] = value**).

### pop() - Delete a value from the Stack

In a stack, pop() is a function used to delete an element from the stack. In a stack, the element is always deleted from **top** position. Pop function does not take any value as parameter. We can use the following steps to pop an element from the stack...

* **Step 1 -**Check whether **stack** is **EMPTY**. (**top == -1**)
* **Step 2 -**If it is **EMPTY**, then display **"Stack is EMPTY!!! Deletion is not possible!!!"** and terminate the function.
* **Step 3 -**If it is **NOT EMPTY**, then delete **stack[top]** and decrement **top** value by one (**top--**).

#### display() - Displays the elements of a Stack

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

* **Step 1 -**Check whether **stack** is **EMPTY**. (**top == -1**)
* **Step 2 -**If it is **EMPTY**, then display **"Stack is EMPTY!!!"** and terminate the function.
* **Step 3 -**If it is **NOT EMPTY**, then define a variable '**i**' and initialize with top. Display **stack[i]** value and decrement **i** value by one (**i--**).
* **Step 3 -**Repeat above step until **i** value becomes '0'.

# Implementation of Stack using Array

#include<stdio.h>

#include<conio.h>

#define SIZE 10

void push(int);

void pop();

void display();

int stack[SIZE], top = -1;

void main()

{

 int value, choice;

 clrscr();

 while(1){

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

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

 printf("\nEnter 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!!! Try again!!!");

 }

 }

}

void push(int value){

 if(top == SIZE-1)

 printf("\nStack is Full!!! Insertion is not possible!!!");

 else{

 top++;

 stack[top] = value;

 printf("\nInsertion success!!!");

 }

}

void pop(){

 if(top == -1)

 printf("\nStack is Empty!!! Deletion is not possible!!!");

 else{

 printf("\nDeleted : %d", stack[top]);

 top--;

 }

}

void display(){

 if(top == -1)

 printf("\nStack is Empty!!!");

 else{

 int i;

 printf("\nStack elements are:\n");

 for(i=top; i>=0; i--)

 printf("%d\n",stack[i]);

 }

}

**OutPut**

**3.Display**

**4.Exit**

**Enter your Choice**

**Enter the value to be Insert**

**Insertion Success**

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

**1.Push**

**2.Pop**

**3.Display**

**4.Exit**

**Enter your Choice :3**

**Stack element are**

**20**

**10**

#  Infix, Postfix and Prefix

Infix, Postfix and Prefix notations are three different but equivalent ways of writing expressions. It is easiest to demonstrate the differences by looking at examples of operators that take two operands.

Infix notation: X + Y

Postfix notation (also known as "Reverse Polish notation"): X Y +

Prefix notation (also known as "Polish notation"): + X Y

**Rule for Conversation**

1. If the input character is an operand, print it.
2. If the input character is an operator-
	* If stack is empty push it to the stack.
	* If it’s precedence value is greater than the precedence value of the character on top, push.
	* If it’s precedence value is lower then pop from stack and print while precedence of top char is more than the precedence value of the input character.
3. If the input character is ‘)’, then pop and print until top is ‘(‘. (Pop ‘(‘ but don’t print it.)
4. If stack becomes empty before encountering ‘(‘, then it’s a invalid expression.
5. Repeat steps 1-4 until input expression is completely read.
6. Pop the remaining elements from stack and print them.

**Convert infix to Postfix**

Convert A \* (B + C) \* D to postfix notation.

|  |  |  |  |
| --- | --- | --- | --- |
| **Move** | **Curren Ttoken** | **Stack** | **Output** |
| 1 | A | empty | A |
| 2 | \* | \* | A |
| 3 | ( | (\* | A |
| 4 | B | (\* | A B |
| 5 | + | +(\* | A B |
| 6 | C | +(\* | A B C |
| 7 | ) | \* | A B C + |
| 8 | \* | \* | A B C + \* |
| 9 | D | \* | A B C + \* D |
| 10 |  | empty |   |
| We use the same to convert Infix to Prefix.Step 1: Reverse the infix expression Step 2: Obey the rule for infix to Post fixStep 3: Reverse the postfix expression. |

Example **(A+B^C)\*D+E^5**
**Step 1.** Reverse the infix expression.
               **5^E+D\*(C^B+A)**

|  |  |  |  |
| --- | --- | --- | --- |
| **Expression** | **Stack** | **Output** | **Comment** |
| 5^E+D\*(C^B+A) | Empty | - | Initial |
| ^E+D\*(C^B+A) | Empty | 5 | Print |
| E+D\*(C^B+A) | ^ | 5 | Push |
| +D\*(C^B+A) | ^ | 5E | Push |
| D\*(C^B+A) | + | 5E^ | Pop And Push |
| \*(C^B+A) | + | 5E^D | Print |
| (C^B+A) | +\* | 5E^D | Push |
| C^B+A) | +\*( | 5E^D | Push |
| ^B+A) | +\*( | 5E^DC | Print |
| B+A) | +\*(^ | 5E^DC | Push |
| +A) | +\*(^ | 5E^DCB | Print |
| A) | +\*(+ | 5E^DCB^ | Pop And Push |
| ) | +\*(+ | 5E^DCB^A | Print |
| End | +\* | 5E^DCB^A+ | Pop Until '(' |
| End | Empty | 5E^DCB^A+\*+ | Pop Every element |

# Queue Data Structure

### [Recent articles on Queue](https://www.geeksforgeeks.org/category/queue/)

A Queue is a linear structure which follows a particular order in which the operations are performed. The order is First In First Out (FIFO). A good example of a queue is any queue of consumers for a resource where the consumer that came first is served first. The difference between [stacks](https://www.geeksforgeeks.org/stack-data-structure/)and queues is in removing. In a stack we remove the item the most recently added; in a queue, we remove the item the least recently added.



# Applications of Queue Data Structure

[Queue](http://en.wikipedia.org/wiki/Queue_%28data_structure%29)is used when things don’t have to be processed immediately, but have to be processed in **F**irst **I**n **F**irst **O**ut order like [Breadth First Search](http://en.wikipedia.org/wiki/Breadth-first_search). This property of Queue makes it also useful in following kind of scenarios.

**1)** When a resource is shared among multiple consumers. Examples include CPU scheduling, Disk Scheduling.

**2)**When data is transferred asynchronously (data not necessarily received at same rate as sent) between two processes. Examples include IO Buffers, pipes, file IO, etc.

## Basic features of Queue

1. Like stack, queue is also an ordered list of elements of similar data types.
2. Queue is a FIFO( First in First Out ) structure.
3. Once a new element is inserted into the Queue, all the elements inserted before the new element in the queue must be removed, to remove the new element.
4. peek( ) function is oftenly used to return the value of first element without dequeuing it.

# Types of Queues in Data Structure

Queue is an important structure for storing and retrieving data and hence is used extensively among all the data structures. Queue, just like any queue (queues for bus or tickets etc.) follows a FIFO mechanism for data retrieval which means the data which gets into the queue first will be the first one to be taken out from it, the second one would be the second to be retrieved and so on.

## Types of Queues in Data Structure

### Simple Queue

As is clear from the name itself, simple queue lets us perform the operations simply. i.e., the insertion and deletions are performed likewise. Insertion occurs at the rear (end) of the queue and deletions are performed at the front (beginning) of the queue list.

All nodes are connected to each other in a sequential manner. The pointer of the first node points to the value of the second and so on.

The first node has no pointer pointing towards it whereas the last node has no pointer pointing out from it.

### Circular Queue



Unlike the simple queues, in a circular queue each node is connected to the next node in sequence but the last node’s pointer is also connected to the first node’s address. Hence, the last node and the first node also gets connected making a circular link overall.

### Priority Queue

Priority queue makes data retrieval possible only through a pre determined priority number assigned to the data items.

While the deletion is performed in accordance to priority number (the data item with highest priority is removed first), insertion is performed only in the order.

### Doubly Ended Queue (Dequeue)



The doubly ended queue or dequeue allows the insert and delete operations from both ends (front and rear) of the queue.

Queues are an important concept of the data structures and understanding their types is very necessary for working appropriately with them.

***/\* Simple Queue Program in C\*/***

**#include<stdio.h>**

**#include<conio.h>**

**#include<stdlib.h>**

**#define MAX\_SIZE 100**

int **main**()

 {

 int item, choice, i;

 int arr\_queue[MAX\_SIZE];

 int rear = 0;

 int front = 0;

 int exit = 1;

 printf("\nSimple Queue Example - Array");

 do {

 printf("\n\n Queue Main Menu");

 printf("\n1.Insert \n2.Remove \n3.Display \nOthers to exit");

 printf("\nEnter Your Choice : ");

 scanf("%d", &choice);

 switch (choice)

 {

 case 1:

 if (rear == MAX\_SIZE)

 printf("\n## Queue Reached Max!!");

 else {

 printf("\nEnter The Value to be Insert : ");

 scanf("%d", &item);

 printf("\n## Position : %d , Insert Value : %d ", rear + 1, item);

 arr\_queue[rear++] = item;

 }

 break;

 case 2:

 if (front == rear)

 printf("\n## Queue is Empty!");

 else {

 printf("\n## Position : %d , Remove Value : %d ", front, arr\_queue[front]);

 front++;

 }

 break;

 case 3:

 printf("\n## Queue Size : %d ", rear);

 for (i = front; i < rear; i++)

 printf("\n## Position : %d , Value : %d ", i, arr\_queue[i]);

 break;

 default:

 exit = 0;

 break;

 }

 } while (exit);

 return 0;

}

**Circular Queue Program.**

#include<stdio.h>

# define MAX 5

int cqueue\_arr[MAX];

int front = -1;

int rear = -1;

void insert(int item)

{

if((front == 0 && rear == MAX-1) || (front == rear+1))

{

printf("Queue Overflow n");

return;

}

if(front == -1)

{

front = 0;

rear = 0;

}

else

{

if(rear == MAX-1)

rear = 0;

else

rear = rear+1;

}

cqueue\_arr[rear] = item ;

}

void deletion()

{

if(front == -1)

{

printf("Queue Underflown");

return ;

}

printf("Element deleted from queue is : %dn",cqueue\_arr[front]);

if(front == rear)

{

front = -1;

rear=-1;

}

else

{

if(front == MAX-1)

front = 0;

else

front = front+1;

}

}

void display()

{

int front\_pos = front,rear\_pos = rear;

if(front == -1)

{

printf("Queue is emptyn");

return;

}

printf("Queue elements :n");

if( front\_pos <= rear\_pos )

while(front\_pos <= rear\_pos)

{

printf("%d ",cqueue\_arr[front\_pos]);

front\_pos++;

}

else

{

while(front\_pos <= MAX-1)

{

printf("%d ",cqueue\_arr[front\_pos])

front\_pos++;

}

front\_pos = 0;

while(front\_pos <= rear\_pos)

{

printf("%d ",cqueue\_arr[front\_pos]);

front\_pos++;

}

}

printf("n");

}

int main()

{

int choice,item;

do

{

printf("1.Insertn");

printf("\n2.Deleten");

printf("\n3.Displayn");

printf("\n4.Quitn");

printf("\nEnter your choice : ");

scanf("%d",&choice);

switch(choice)

{

case 1 :

printf("Input the element for insertion in queue : ");

scanf("%d", &item);

insert(item);

break;

case 2 :

deletion();

break;

case 3:

display();

break;

case 4:

break;

default:

printf("Wrong choicen");

}

}

while(choice!=4);

return 0;

}