Data Structure and Algorithm

**Data structure** is process to save the data in memory with organize ways. Array, linked list, stack,

Graph, tree is data structure which way we can save the data in memory.

**Algorithms** is a set of well-defined instructions to solve a particular problem. We can divide the any problem and solve each problem step by step is called algorithms.

**Why data structure and algorithms:**

We use data structure for save data in memory in organize way. But we want to use this data with perfect way. so that we need algorithms. with algorithms we can perform various operation with data in data structure.

# Pointer

In programming language, when we create a variable, it is assigned some space the computer memory. Every memory location has its address. The memory location is defined by 8-byte memory address. To know the location in the computer memory where the data is stored, C++ provides the **& (reference)** operator. The **&** operator returns the address that a variable occupies.

string food = "Pizza";

cout << &food; // output : 0x7fff5fbff8ac

**Pointer:**

1. A **pointer** is a variable that stores address of another variable.
2. A pointer can also be used to refer to another pointer function.
3. A pointer can be incremented/decremented to point to the next/ previous memory location.
4. The purpose of pointer is to save memory space and achieve faster execution time.

Declaring a pointer:

The pointer in c language can be declared using \* **(asterisk symbol).** It is also known as indirection pointer used to dereference a pointer.

**Datatype \*value\_name;**

**int \*a; //pointer to int**

**char \*b; //pointer to char**



**Code:**

int main () {

int a=5;

**…………………………pointer initial ………………………….**

**int \*p;**

**p=&a;**

**cout<<p<<" "<<\*p<<" "<<&p<<end;**

**…………………………. Dereferencing……………………..**

\*p = 12 ;

cout<<a<<" "<<\*p;

}

Rule :

1. **In pointer, variable data type and pointer variable data should be same.**

Int x, \*y; // no error

Int x;

Float \*y;

y = &x ; // error occur

1. **pointer variable, we cannot directly assign the value ;**

int\* x = ff4h;

**Pointers and Arrays:**

1. The array name itself denotes the base address of the array.
2. To assign the address of an array to a pointer, you should not use an ampersand (&).

**int arr [20];**

**int \* p;**

**p = arr;**

**Pointer to a function:**

void show (int);

void(\*p) (int) = &display;

**Pointer to structure:**

struct st {

    int i;

    float f;

} ref;

struct st \*p = &ref;

### Types of Pointers:

1. Null Pointer
2. Void Pointer
3. Wild pointer
4. Dangling pointer
5. Complex pointer
6. Near pointer
7. Far pointer
8. Huge pointer

**Null Pointer:**

1. If there is no exact address that is to be assigned, then the pointer variable can be assigned a NULL.
2. It should be done during the declaration.
3. The value of null is 0.
4. It is useful for handling errors when using malloc function.

**int main() {**

**int \*ptr = NULL;**

**cout << ptr ;**

**return 0;**

**}**

**Void pointer:**

1. void pointer is also called as a generic pointer.
2. It does not have any standard data type.
3. A void pointer is created by using the keyword void.
4. It can be used to store an address of any variable.

**int main () {**

**int n = 10;**

**int \*ptr = &n;**

**cout<< \*(int\*) ptr ;**

**}**

**Dangling pointer:**

1. A dangling pointer is a pointer which points to some non-existing memory location.
2. When we free the pointer but not re-initialize the pointer, then pointer is still pointing to the deallocated memory.
3. So, we need to re-initialized the memory.

**int main() {**

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

**……….**

**………**

**free(ptr);**

**}**

**Wild pointer:**

1. Wild pointers are also known as uninitialized pointers.
2. These pointers usually point to some arbitrary memory location and may cause a program to cash or misbehave .

**int main() {**

**int \*ptr ;**

**\*ptr = 10;**

**}**

# Memory allocation

1. Memory allocation is a process by which computer programs and services are assigned with physical or virtual memory space.
2. Memory allocation is the process of reserving a partial or complete portion of computer memory for the execution of programs and processes.
3. Memory allocation is achieved through a process known as memory management.
4. Memory allocation is primarily a computer hardware operation but is managed through operating system and software applications.

## Memory Layout

When we create a C program and run the program, its executable file is stored in the RAM of the computer in an organized manner.



A typical memory representation of a C program consists of the following sections.

1. Text segment/ code segment (instructions)
2. Initialized data segment
3. Uninitialized data segment (bss)
4. Heap
5. Stack

### Text segment:

The text segment is also known as the code segment. Text segment contains machine code of the compiled program which contains executable instructions. The text segment is sharable so that only a single copy needs to be in memory for frequently executed programs, such as text editors, the C compiler, the shells.

### Data Segment:

The data which we use in our program will be stored in the data section. the variables declared outside the main () method will be stored in the data section.

**The data section consists of two segments:**

1. Uninitialized data segment
2. Initialized data segment

**Uninitialized data segment:**

The uninitialized data segment is also known as a **bss (Block Started by symbol) segment**. **Uninitialized data segment** stores all the uninitialized global, local and external variables. If the global, static and external variables are not initialized, they are assigned with zero value by default.

#**include<stdio.h>**

**char a;    // uninitialized global variable..**

**int main()**

**{**

**static int a;   // uninitialized static variable..**

**return 0;**

**}**

**Initialized data segment**

An initialized data segment is also known as the data segment. A data segment is a virtual address space of a program that contains all the global and static variables which are explicitly initialized by the programmer.

**#include<stdio.h>**

**char string[] = "javatpoint";  // global variable stored in initialized data segment in read-write area..**

**int main()**

**{**

**static int i = 90;   // static variable stored in initialized data segment..**

**return 0;**

**}**

### Heap

Heap memory is used for the dynamic memory allocation. Heap memory begins from the end of the uninitialized data segment and grows upwards to the higher addresses. The malloc() and calloc() functions are used to allocate the memory in the heap. The heap memory can be used by all the shared libraries and dynamically loaded modules. The free() function is used to deallocate the memory from the heap.

**#include<stdio.h>**

**int main()**

**{**

**………………….. memory allocated in the heap segment…………………………….**

**int \*ptr = (int\*) malloc ( sizeof ( int )) ;  .**

**return 0;**

**}**

**Stack**

When we define a function and call that function then we use the stack frame. The variables which are declared inside the function are stored in the stack. Stack memory allocation is known as static memory allocation because all the variables are defined in the function, and the size of the variables is also defined at the compile time

## Type of memory allocation:

1. Static memory allocation
2. Dynamic memory allocation

### Static Memory Allocation

Static memory allocated during compile time. Memory allocated is fixed and cannot be increased or decreased during run time.

**int main(){  
 int arr[5] = { 1, 2, 3, 4, 5} ;**

**}**

### Dynamic memory allocation

The process of allocating memory at the time of execution is called dynamic memory allocation. Dynamic memory allocation takes place in heap segment. In heap memory allocated or deallocated without any order. Pointer play an important role in dynamic memory allocation. allocated memory can only be accessed through pointer.

Dynamic memory function:

1. Malloc ()
2. Calloc ()
3. Realloc ()
4. Free ()

#### Malloc ():

1. Malloc () is a built-in function declared in the header file <stdlib.h>.
2. Malloc () is the short name for “memory allocation”.
3. Malloc () is used to dynamically allocated a single large block of contiguous memory according to the specified size in heap.
4. It doesn't initialize memory at execution time, so it has garbage value initially.
5. It returns NULL if memory is not sufficient.
6. Malloc () function return **void pointer** pointing to the first byte of the allocated memory.
7. If memory allocated is fail it return **null pointer**.

**Syntax:**

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

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

**why malloc return void pointer?**  
malloc doesn’t have an idea of what it is pointing to. it merely allocates memory requested by the user without knowing the type of data to be stored inside the memory. It simply allocated memory for program. So that it returns void pointer. after allocated the memory void pointer can be type casted to an appropriate type. Malloc allocates 4 byte of memory in the heap and the address of the first byte is store in the pointer ptr.

**Int ptr = (int\*) malloc( 4 );**

**Code:**

**#include <iostream>**

**#include <cstdlib>**

**using namespace std;**

**int main(){**

**int\* m;**

**int\* n;**

**void\* p ;**

**................ malloc return void pointer.......**

**p = malloc(20);**

**cout<<p<<endl;**

**cout<<sizeof(p)<<endl;**

**................. type cast malloc ................**

**m = (int\*) malloc (20);**

**n = (int\*) malloc( 5 \* sizeof(int));**

**if( n == NULL) {**

**cout<<"Memory not allocated"<<endl;**

**} else {**

**cout<<"memory allocated"<<endl;**

**cout<< "data address = "<<n<<endl;**

**cout<< "next data address = "<<n+1<<endl;**

**cout<< "3rd data address = "<<n+2<<endl;**

**}**

**………....... melloc data insert .................**

**int x;**

**cout<<" insert the number of input"<<endl;**

**cin>>x;**

**int\* a = (int\*) malloc( x \* sizeof(int));**

**if(a==NULL) {**

**cout<<"memory not allocated"<<endl;**

**}**

**for(int i=0; i<x; i++) {**

**cout<<"enter the number"<<endl;**

**cin>>\*(a+i);**

**}**

**for (int i=0; i<x; i++) {**

**cout<<"address = "<<a+i<<" value ="<<\*(a+i)<<endl;**

**}**

**}**

#### calloc ():

1. The calloc () function allocates multiple block of requested memory.
2. It initially initializes all bytes to zero.
3. It returns NULL if memory is not sufficient

**Syntax:**

**ptr = (castType\*) calloc (n, size);**

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

**code:**

**#include <iostream>**

**#include <cstdlib>**

**using namespace std;**

**int main (){**

**int\* m;**

**int\* n;**

**void\* p;**

**…............. calloc return void pointer..........**

**p = calloc (5, sizeof(int));**

**cout << p <<endl;**

**cout << sizeof(p) <<endl;**

**…….............. type cast calloc ...................**

**n = (int\*) calloc (5, sizeof(int));**

**if (n == NULL) {**

**cout<<"Memory not allocated"<<endl;**

**}else{**

**cout<<"memory allocated"<<endl;**

**cout<< "data address = "<<n<<endl;**

**cout<< "next data address = "<<n+1<<endl;**

**cout<< "3rd data address = "<<n+2<<endl;**

**}**

**………. ...... calloc data insert…… ............**

**int x;**

**cout<<" insert the number of input"<<endl;**

**cin>>x;**

**int\* a = (int\*) calloc (x, sizeof(int));**

**if(a==NULL) {**

**cout<<"memory not allocated"<<endl;**

**}**

**for (int i=0; i<x; i++) {**

**cout<<"enter the number"<<endl;**

**cin>>\*(a+i);**

**}**

**for (int i=0; i<x; i++) {**

**cout<<"address = "<<a+i<<" value ="<<\*(a+i) <<endl;**

**}**

}

#### free ():

The memory occupied by malloc () or calloc () functions must be released by calling free () function.

**Syntax:**

**free(ptr)**

**code:**

**#include <iostream>**

**#include <cstdlib>**

**using namespace std;**

**int main(){**

**int \*p;**

**p = (int\*) malloc (20);**

**cout<<"after allocation p = "<<p<<endl;**

**free(p);**

**cout<<"after free the memory p= "<<p<<endl;**

**}**

#### realloc()

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

**Syntax :**

**ptr = realloc (ptr, x);**

Here, ptr is reallocated with a new size x.

int main () {

int \*ptr, i , n1, n2;

printf("Enter size: ");

scanf("%d", &n1);

ptr = (int\*) malloc (n1 \* sizeof(int));

**……………………..Addresses of previously allocated memory………**

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

cout<< ptr + i ;

**……………………………. rellocating the memory ……………………………………….**

ptr = realloc (ptr, 10 \* sizeof(int));

printf("Addresses of newly allocated memory:\n");

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

printf("%pc\n", ptr + i);

free(ptr);

return 0;

}

# Structure

1. A struct (or structure) is a collection of variables (can be of different types) under a single name.
2. It allows different variables to be accessed by using a single pointer to the structure.
3. struct keyword is used to define a structure. struct defines a new data type which is a collection of primary and derived data types.

**struct structure\_name**

**{**

**data\_type member1;**

**data\_type member2;**

**};**

**Declaring Structure Variables**

1. Declaring Structure variables separately:

**struct Student{**

**char name[25];**

**int age;**

**char branch[10];**

**char gender;**

**};**

**struct Student S1, S2;**

1. Declaring Structure variables with structure definition

**struct Student{**

**char name[25];**

**int age;**

**char branch [10];**

**char gender;**

**} S1, S2;**

**Access Members of a Structure:**

There are two types of operators used for accessing members of a structure.

* + - 1. Member operator ( . )
      2. Structure pointer operator ( -> )

**Program:**

**struct Person {**

**char name [50];**

**int cityNo;**

**float salary;**

**} p ;**

**int main () {**

**strcpy ( p.name, "George Orwell" );**

**p.cityNo = 1984;**

**p. salary = 2500;**

**cout<<Name: << p.name ;**

**cout<< Citizenship No << p.cityNo ;**

**cout<< Salary: << p.salary ;**

**return 0;**

**}**

Passing structure member as argument:

**#include <iostream>**

**using namespace std;**

**struct student{**

**char name[50];**

**int age;**

**int roll;**

**}s1;**

**void show(char name[], int age, int roll){**

**cout<<"----after funtion called----"<<endl;**

**cout<<name<<" "<<age<<" "<<roll<<endl;**

**}**

**int main(){**

**struct student s = {"shuvo", 20,36};**

**cout<<s.name<<" " <<s.age<<endl;**

**cout<<s1.name<<" " <<s1.age<<endl;**

**show(s.name,s.age,s.roll);**

**}**

Passing structure Reference at function:

# Data Structures

**Data:**

Data can be defined as a representation of facts, concepts, or instructions in a formalized manner, which should be suitable for communication, interpretation, or processing by human or electronic machine. Data is represented with the help of characters such as alphabets (A-Z, a-z), digits (0-9) or special characters (+,-,/,\*,<,>,= etc.) .

**student's name and its id are the data about the student. Catch**

**Group Items:**

Data items which have subordinate data items are called Group item, Example - **name of a student can have first name and the last name.**

**Record:**

Record can be defined as the collection of various data items. Example - **In student entity, then its name, address, course, and marks can be grouped together to form the record for the student**.

**File:**

A File is a collection of various records of one type of entity. Example - **if there are 60 employees in the class, then there will be 20 records in the related file where each record contains the data about each employee.**

**Attribute and Entity:**

An entity represents the class of certain objects. it contains various attributes. Each attribute represents the particular property of that entity.

**Field:**

Field is a single elementary unit of information representing the attribute of an entity.

**What are Data Structures?**

Data Structure can be defined as the group of data elements which provides an efficient way of storing and organizing data in the computer so that it can be used efficiently. **Arrays, Linked List, Stack, Queue.**

## Types of Data Structures:



**There are two types of data structures:**

1. Primitive data structure
2. Non-primitive data structure

### **Primitive Data structure:**

The primitive data structures are primitive data types that can hold a single value. They are ---

1. **int,**
2. **char,**
3. **float,**
4. **double, and**
5. **pointer**

### **Non-Primitive Data structure**:

The non-primitive data structure is divided into two types:

1. Linear data structure (Arrays, linked list, Stacks, and Queues)
2. Non-linear data structure (trees **and graphs)**

#### **Linear Data Structure**

1. The arrangement of data in a sequential manner is known as a linear data structure.
2. The data structures used for this purpose are Arrays, linked list, Stacks, and Queues.
3. In these data structures, one element is connected to only one another element in a linear form.

**Static data structure**: It is a type of data structure where the size is allocated at the compile time. Therefore, the maximum size is fixed. **Arrays**

**Dynamic data structure:** It is a type of data structure where the size is allocated at the run time. Therefore, the maximum size is flexible.

1. **Linked List**
2. **Stack**
3. **Queue**

#### **Non-Linear Data Structure**

1. **Tree**
2. **graph**

# Array

Array is a container which can hold a fix number of items and these items should be of the same type. Most of the data structures make use of arrays to implement their algorithms.

1. **Element** − Each item stored in an array is called an element.
2. **Index** − Each location of an element in an array has a numerical index, which is used to identify the element.

**Array Representation:**

Arrays can be declared in various ways in different languages.



**Arrays index:**

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

1. Index starts with 0.
2. Array length is 10 which means it can store 10 elements.
3. Each element can be accessed via its index.

**Program without array:**

**void** main () {

**int** marks\_1 = 56, marks\_2 = 78, marks\_3 = 88, marks\_4 = 76;

**float** avg = (marks\_1 + marks\_2 + marks\_3 + marks\_4 ) / 4 ;

printf(avg);

}

**Program by using array:**

**void** main () {

**int** marks[6] = {56,78,88,76,56,89);

**int** i;

**float** avg;

**for** (i=0; i<6; i++ )    {

        avg = avg + marks[i];

    }

    printf(avg);

}

## Types of Arrays in java:

* + - 1. Single Dimensional Array
      2. Multidimensional Array

**Single Dimensional Array :**

In single dimensional array data store in one sequential line .

**datatype arr\_name [array\_size] ;**

**Multi-Dimensional Array:**

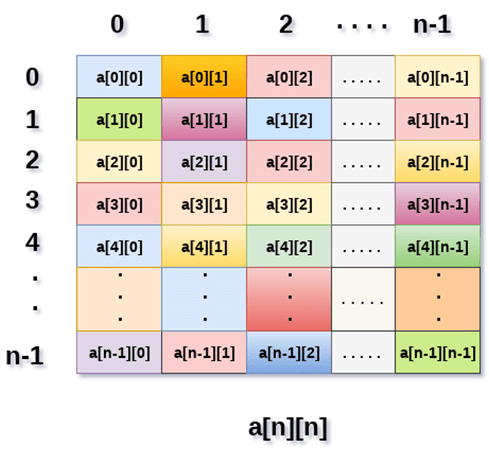
Multidimensional array is various type. That can be two dimensional, three or more.

**Syntax:**

**data\_type array\_name [n] [m];**

**Example:**

**int array [10] [20];**



## Basic Operations:

Following are the basic operations supported by an array.

1. Traverse − print all the array elements one by one.
2. Insertion − Adds an element at the given index.
3. Deletion − Deletes an element at the given index.
4. Search − Searches an element using the given index or by the value.
5. Update − Updates an element at the given index.

**CODE:**

**public** **class** main {

**static** **int** *count* = -1;

**static** **int** *size* = 10;

**public** **static** **void** main(String [] args) {

Scanner input = **new** Scanner(System.***in***);

**int**[] a = **new** **int**[*size*];

**while** (**true**) {

System.***out***.println("enter value for select menu");

System.***out***.println("0 - count ");

System.***out***.println("1 - show array ");

System.***out***.println("2 - input in start");

System.***out***.println("3 - input end ");

System.***out***.println("4 -- input any position ");

System.***out***.println("5 - delete any positon");

System.***out***.println("6 - delete in end ");

System.***out***.println("7 - search ");

**int** x = input.nextInt();

**switch** (x) {

**case** 0:

System.***out***.println(*count*);

**break**;

**case** 1:

*show*(a);

**break**;

**case** 2:

*input\_start*(a);

**break**;

**case** 3:

*input\_end*(a, input);

**break**;

**case** 4:

*input\_position*(a, input);

**break**;

**case** 5:

*delete\_position*(a, input);

**break**;

}

}

}

// >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> show all array

**static** **void** show(**int** a[]) {

**for** (**int** i = 0; i < *size*; i++) {

System.***out***.println(a[i]);

}

}

// array in end position

**static** **void** input\_end(**int** a[], Scanner input) {

**if** (*count* != a.length - 1) {

**for** (**int** i = *count* + 1; i < a.length; i++) {

System.***out***.println("give your input value");

a[i] = input.nextInt();

*count*++;

**break**;

}

} **else** {

System.***out***.println("give your input value");

a[*count*] = input.nextInt();

}

}

// >>>>>>>>>>>>>>>>>>>>> input in start <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

**static** **void** input\_start(**int**[] a) {

Scanner input = **new** Scanner(System.***in***);

**for** (**int** i = a.length - 1; i > 0; i--) {

a[i] = a[i - 1];

}

System.***out***.println("input value ");

**int** v = input.nextInt();

a[0] = v;

*count*++;

}

//…………………………………………………….input an any position

**static** **void** input\_position(**int**[] a, Scanner input) {

System.***out***.println("enter position of array");

**int** p = input.nextInt();

System.***out***.println("enter value of array");

**int** v = input.nextInt();

**if** (p >= *count*) {

*count* = p;

a[p] = v;

} **else** {

*count*++;

**for** (**int** i = *count*; i == p; i--) {

a[i] = a[i - 1];

}

a[p] = v;

}

}

**static** **void** delete\_position(**int**[] a,Scanner input) {

System.***out***.println("input position");

**int** v = input.nextInt();

**for**(**int** i=v;i<a.length-1;i++) {

a[i]=a[i+1];

}

a[*count*]=0;

*count*--;

}

}

# Linked list

1. Linked list is a linear data structure that includes a series of connected nodes.
2. Linked list can be defined as the nodes that are randomly stored in the memory.
3. The last node of the list contains a pointer to the null.

**Representation of a Linked list:**

Linked list can be represented as the connection of nodes in which each node points to the next node of the list.

**Representation of linked list:if**

Linked list contains two parts-

1. **Data part** - the simple variable
2. **The address part** - the pointer variable (address of next node).
3. We wrap both the data item and the next node reference in a struct as.

struct node{

int data;

struct node \*next;

}

1. Then Initialize the node.

struct node \*head;

struct node \*one = NULL;

struct node \*two = NULL;

struct node \*three = NULL;

1. Then allocate the memory.

one = malloc( sizeof(struct node) );

two = malloc( sizeof(struct node) );

three = malloc( sizeof(struct node) );

1. Assign the value.

one->next = two;

two ->next = three;

three ->next = Null;

1. Save the address of first node.

head = one;



## Types of Linked list:

Linked list is classified into the following types -

1. **Singly linked list**
2. **Doubly linked list**
3. **Circular singly linked list**
4. **Circular doubly linked list**

### Singly Linked List:

1. Singly linked list as like linked list.
2. Its navigation is forward only.



**Code :**

#include <iostream>

#include <cstdlib>

using namespace std;

**struct node {**

**int data;**

**struct node \*link;**

**};**

int main(){

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

**head->data = 5;**

**head->link = NULL;**

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

**current->data = 10;**

**current->link = NULL;**

**head->link = current;**

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

**next->data = 56;**

**next->link = NULL;**

**current->link = next;**

}

## Basic Operations:

Following are the basic operations supported by an array.

1. Traverse − print all the array elements one by one.
2. Insertion − Adds an element at the given index.
3. Deletion − Deletes an element at the given index.
4. Search − Searches an element using the given index or by the value.
5. Update − Updates an element at the given index.

#include<stdio.h>

#include<stdlib.h>

struct node {

    int data;

    struct node \*next;

};

struct node \*head;

void beginsert ();

void lastinsert ();

void randominsert();

void begin\_delete();

void last\_delete();

void random\_delete();

void display();

void search();

void main (){

    int choice =0;

    while(choice != 9)  {

        printf("\nEnter your choice?\n");

        scanf("\n%d",&choice);

        switch(choice)  {

            case 1:

            beginsert();

            break;

            case 2:

            lastinsert();

            break;

            case 3:

            randominsert();

            break;

            case 4:

            begin\_delete();

            break;

            case 5:

            last\_delete();

            break;

            case 6:

            random\_delete();

            break;

            case 7:

            search();

            break;

            case 8:

            display();

            break;

            default:

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

        }

    }

}

void beginsert(){

    struct node \*ptr;

    int item;

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

    if(ptr == NULL)  {

        printf("\nOVERFLOW");

    }else {

        printf("\nEnter value\n");

        scanf("%d",&item);

        ptr->data = item;

        ptr->next = head;

        head = ptr;

        printf("\nNode inserted");

    }

}

void lastinsert()

{

    struct node \*ptr,\*temp;

    int item;

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

    if(ptr == NULL) {

        printf("\nOVERFLOW");

    }else {

        printf("\nEnter value?\n");

        scanf("%d",&item);

        ptr->data = item;

        if(head == NULL) {

            ptr -> next = NULL;

            head = ptr;

            printf("\nNode inserted");

        } else {

            temp = head;

            while (temp -> next != NULL){

                temp = temp -> next;

            }

            temp->next = ptr;

            ptr->next = NULL;

            printf("\nNode inserted");

        }

    }

}

void randominsert(){

    int i,loc,item;

    struct node \*ptr, \*temp;

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

    if(ptr == NULL) {

        printf("\nOVERFLOW");

    } else {

        printf("\nEnter element value");

        scanf("%d",&item);

        ptr->data = item;

        printf("\nEnter the location after which you want to insert ");

        scanf("\n%d",&loc);

        temp=head;

        for(i=0;i<loc;i++){

            temp = temp->next;

            if(temp == NULL)  {

                printf("\ncan't insert\n");

                return;

            }

        }

        ptr ->next = temp ->next;

        temp ->next = ptr;

        printf("\nNode inserted");

    }

}

void begin\_delete(){

    struct node \*ptr;

    if(head == NULL){

        printf("\nList is empty\n");

    }else{

        ptr = head;

        head = ptr->next;

        free(ptr);

        printf("\nNode deleted from the begining ...\n");

    }

}

void last\_delete(){

    struct node \*ptr,\*ptr1;

    if(head == NULL){

        printf("\nlist is empty");

    }else if(head -> next == NULL) {

        head = NULL;

        free(head);

        printf("\nOnly node of the list deleted ...\n");

    }else{

        ptr = head;

        while(ptr->next != NULL){

            ptr1 = ptr;

            ptr = ptr ->next;

        }

        ptr1->next = NULL;

        free(ptr);

        printf("\nDeleted Node from the last ...\n");

    }

}

void random\_delete(){

    struct node \*ptr,\*ptr1;

    int loc,i;

    printf("\n Enter the location of the node after which you want to perform deletion \n");

    scanf("%d",&loc);

    ptr=head;

    for(i=0;i<loc;i++) {

        ptr1 = ptr;

        ptr = ptr->next;

        if(ptr == NULL) {

            printf("\nCan't delete");

            return;

        }

    }

    ptr1 ->next = ptr ->next;

    free(ptr);

    printf("\nDeleted node %d ",loc+1);

}

void search()

{

    struct node \*ptr;

    int item,i=0,flag;

    ptr = head;

    if(ptr == NULL){

        printf("\nEmpty List\n");

    }else {

        printf("\nEnter item which you want to search?\n");

        scanf("%d",&item);

        while (ptr!=NULL) {

            if(ptr->data == item) {

                printf("item found at location %d ",i+1);

                flag=0;

            } else {

                flag=1;

            }

            i++;

            ptr = ptr -> next;

        }

        if(flag==1) {

            printf("Item not found\n");

        }

    }

}

**void display(){**

**struct node \*ptr;**

**ptr = head;**

**if(ptr == NULL) {**

**printf("Nothing to print");**

**} else{**

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

**while (ptr!=NULL){**

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

**ptr = ptr -> next;**

**}**

**}**

**}**

### Doubly linked list:

Doubly linked list is a complex type of linked list in which a node contains a pointer to the previous as well as the next node in the sequence. Therefore, in a doubly linked list, a node consists of three parts:

1. node data,
2. pointer to the next node in sequence (next pointer) ,
3. pointer to the previous node (previous pointer).

A sample node in a doubly linked list is shown in the figure.



A doubly linked list containing three nodes having numbers from 1 to 3 in their data part, is shown in the following image.



struct node

{

**int** data;

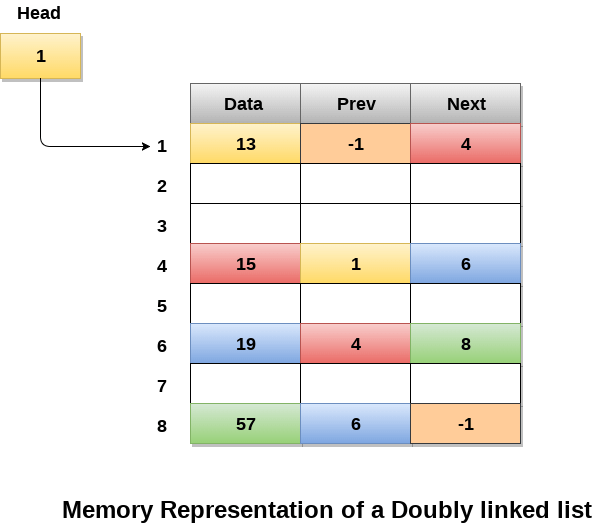
    struct node \*prev;

    struct node \*next;

}

The **prev** part of the first node and the **next** part of the last node will always contain null indicating end in each direction.

**Memory Representation of a doubly linked list**



**Circular Singly Linked List**

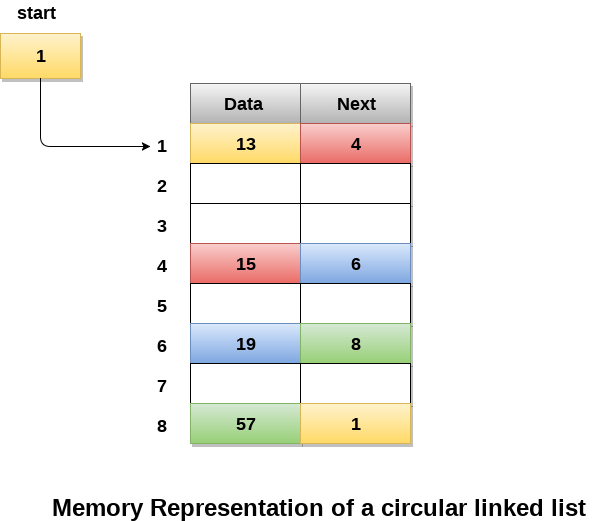
In a circular Singly linked list, the **Last node** of the list contains a pointer to the **first node** of the list.

1. We traverse a circular singly linked list until we reach the same node where we started.
2. the circular singly liked list has no beginning and no ending.
3. There is no null value present in the next part of any of the nodes.



**Memory Representation of circular linked list:**

circular linked list in the memory therefore the last node of the list contains the address of the first node of the list.

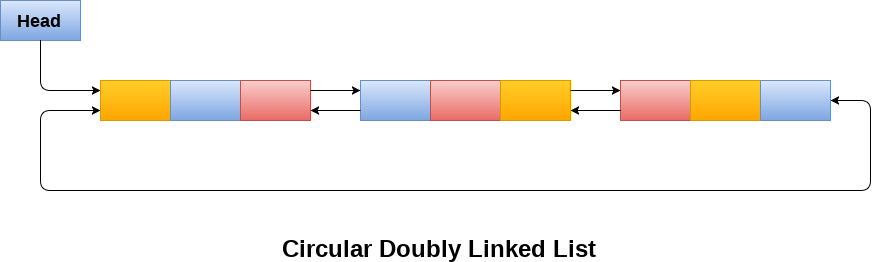


**Circular Doubly Linked List**

Circular doubly linked list is a more complexed type of data structure in which a node **contains pointers to its previous node as well as the next node**.

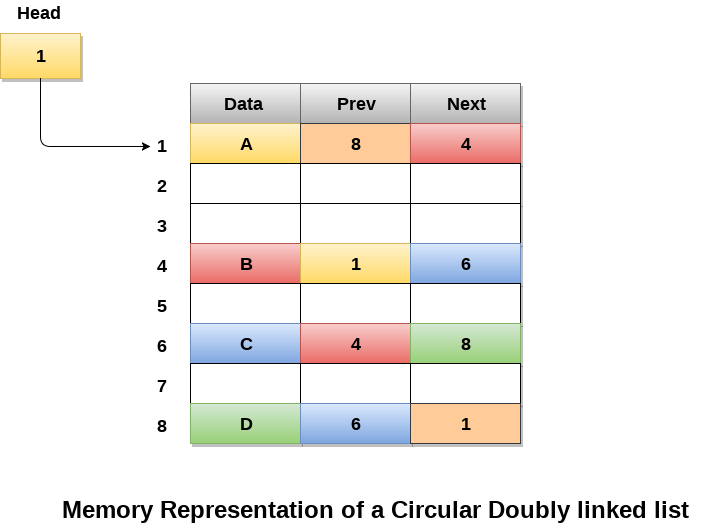
1. Circular doubly linked list doesn't contain NULL in any of the node.
2. The last node of the list contains the address of the first node of the list.
3. The first node of the list also contains address of the last node in its previous pointer.

A circular doubly linked list is shown in the following figure.



**Memory Management of Circular Doubly linked list:**

Difference between JDK, JRE, and JVM



**Basic Operations:**

Following are the basic operations supported by a list.

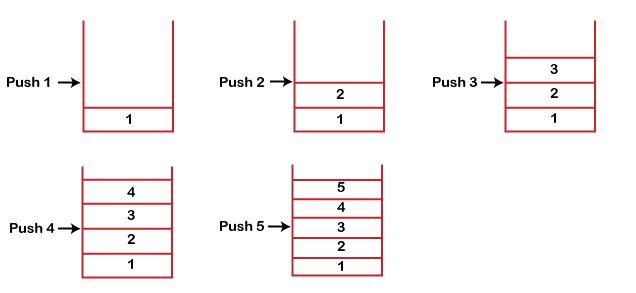
1. Insertion − Adds an element at the beginning of the list.
2. Deletion − Deletes an element at the beginning of the list.
3. Display − Displays the complete list.
4. Search − Searches an element using the given key.
5. Delete − Deletes an element using the given key

Stack

Stack is a linear list in which insertion and deletions are allowed only at one end, called **top**. A stack is an abstract data type (ADT). It can follow the LIFO (Last-In- First-Out) principle . it contains only one pointer **top pointer.**

**Working of Stack**

Stack works on the LIFO pattern. As we can observe in the below figure there are five memory blocks in the stack; therefore, the size of the stack is 5. Suppose we want to store the elements in a stack and let's assume that stack is empty. We have taken the stack of size 5 as shown below in which we are pushing the elements one by one until the stack becomes full.



Since our stack is full as the size of the stack is 5. In the above cases, we can observe that it goes from the top to the bottom when we were entering the new element in the stack. The stack gets filled up from the bottom to the top. When we perform the delete operation on the stack, there is only one way for entry and exit as the other end is closed. It follows the LIFO pattern, which means that the value entered first will be removed last. In the above case, the value 5 is entered first, so it will be removed only after the deletion of all the other elements.

**Standard Stack Operations**

**The following are some common operations implemented on the stack:**

1. **push():** When we insert an element in a stack then the operation is known as a push. If the stack is full, then the overflow condition occurs.
2. **pop():** When we delete an element from the stack, the operation is known as a pop. If the stack is empty means that no element exists in the stack, this state is known as an underflow state.
3. **isEmpty():** It determines whether the stack is empty or not.
4. **isFull():** It determines whether the stack is full or not.'
5. **peek():** It returns the element at the given position.
6. **count():** It returns the total number of elements available in a stack.
7. **change():** It changes the element at the given position.
8. **display():** It prints all the elements available in the stack.

**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). it contains two pointers. One pointer enables insert operations to be performed at end called **REAR** and other pointer delete operations to be performed at another end called **FRONT**.



**Basic Operations:**

Queue operations may involve initializing or defining the queue, utilizing it, and then completely erasing it from the memory.

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

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

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

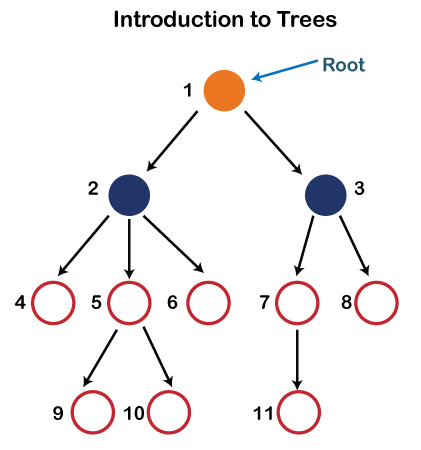
**Non-Linear Data Structures:**

This data structure does not form a sequence i.e., each item or element relates to two or more other items in a non-linear arrangement. The data elements are not arranged in sequential structure. On-Linear Data Structures:

1. **Trees**
2. **Graphs**

**Trees**

1. A tree data structure is defined as a collection of objects or entities known as nodes that are linked together to represent or simulate hierarchy.
2. A tree data structure is a non-linear data structure because it does not store in a sequential manner. It is a hierarchical structure as elements in a Tree are arranged in multiple levels.
3. In the Tree data structure, the topmost node is known as a root node. Each node contains some data, and data can be of any type. In the above tree structure, the node contains the name of the employee, so the type of data would be a string.
4. Each node contains some data and the link or reference of other nodes that can be called children.

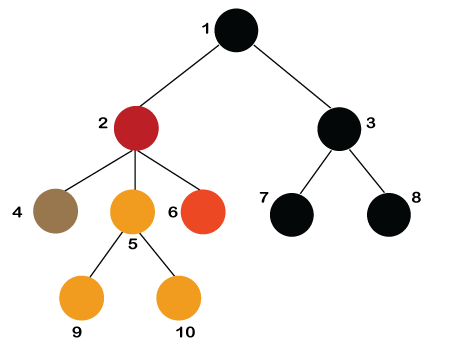


Let's consider the tree structure, which is shown below:

In the above structure, each node is labeled with some number. Each arrow shown in the above figure is known as a ***link*** between the two nodes.

1. **Root:** The root node is the topmost node in the tree hierarchy. In other words, the root node is the one that doesn't have any parent. In the above structure, node numbered 1 is **the root node of the tree.**
2. **Child node:** If the node is a descendant of any node, then the node is known as a child node.
3. **Parent:** If the node contains any sub-node, then that node is said to be the parent of that sub-node.
4. **Sibling:** The nodes that have the same parent are known as siblings.
5. **Leaf Node:-** The node of the tree, which doesn't have any child node, is called a leaf node. A leaf node is the bottom-most node of the tree. There can be any number of leaf nodes present in a general tree. Leaf nodes can also be called external nodes.
6. **Internal nodes:** A node has at least one child node known as an ***internal.***
7. **Ancestor node:-** An ancestor of a node is any predecessor node on a path from the root to that node. The root node doesn't have any ancestors. In the tree shown in the above image, nodes 1, 2, and 5 are the ancestors of node 10.
8. **Descendant:** The immediate successor of the given node is known as a descendant of a node. In the above figure, 10 is the descendant of node 5.

**Properties of Tree data structure**

1. **Recursive data structure:** The tree is also known as a **recursive data structure**. A tree can be defined as recursively because the distinguished node in a tree data structure is known as a **root node**. The root node of the tree contains a link to all the roots of its subtrees. The left subtree is shown in the yellow color in the below figure, and the right subtree is shown in the red color. The left subtree can be further split into subtrees shown in three different colors. Recursion means reducing something in a self-similar manner. So, this recursive property of the tree data structure is implemented in various applications.
2. **Number of edges:** If there are n nodes, then there would n-1 edges. Each arrow in the structure represents the link or path. Each node, except the root node, will have atleast one incoming link known as an edge. There would be one link for the parent-child relationship.
3. **Depth of node x:** The depth of node x can be defined as the length of the path from the root to the node x. One edge contributes one-unit length in the path. So, the depth of node x can also be defined as the number of edges between the root node and the node x. The root node has 0 depth.
4. **Height of node x:** The height of node x can be defined as the longest path from the node x to the leaf node.

**Implementation of Tree**

The tree data structure can be created by creating the nodes dynamically with the help of the pointers. The tree in the memory can be represented as shown below:



The above figure shows the representation of the tree data structure in the memory. In the above structure, the node contains three fields. The second field stores the data; the first field stores the address of the left child, and the third field stores the address of the right child.

In programming, the structure of a node can be defined as:

struct node

{

**int** data;

struct node \*left;

struct node \*right;

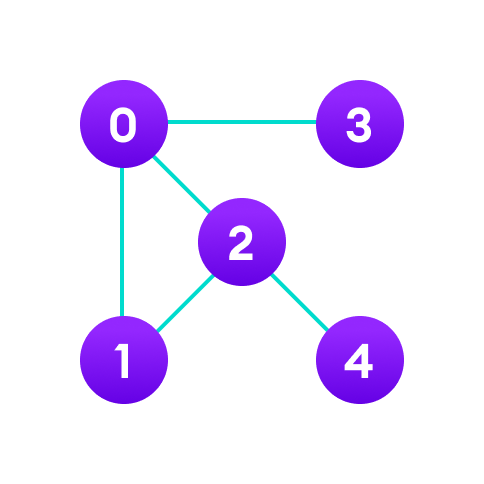
}

**Types of Tree data structure**

1. General tree
2. Binary tree
3. Binary search tree
4. AVL tree
5. Red-black tree
6. Splay tree
7. Treap
8. B-tree

**Graphs:**

Graphs can be defined as the pictorial representation of the set of elements (represented by vertices) connected by the links known as edges. A graph is different from tree in the sense that a graph can have cycle while the tree cannot have the one.



# Algorithm

An algorithm is a process, or a set of rules required to perform calculations or some other problem-solving operations especially by a computer. The formal definition of an algorithm is that it contains the finite set of instructions which are being carried in a specific order to perform the specific task. It is not the complete program or code; it is just a solution (logic) of a problem, which can be represented either as an informal description using a Flowchart or Pseudocode.

**Algorithm Example:**

The following are the steps required to add two numbers entered by the user:

Step 1: Start

Step 2: Declare three variables a, b, and sum.

Step 3: Enter the values of a and b.

Step 4: Add the values of a and b and store the result in the sum variable, i.e., sum= a + b.

Step 5: Print sum

Step 6: Stop

From the data structure point of view, following are some important categories of algorithms.

1. **Search − Algorithm to search an item in a data structure.**
2. **Sort − Algorithm to sort items in a certain order.**
3. **Insert − Algorithm to insert item in a data structure.**
4. **Update − Algorithm to update an existing item in a data structure.**
5. **Delete − Algorithm to delete an existing item from a data structure.**

**Algorithm Complexity**

An algorithm is said to be efficient and fast if it takes less time to execute and consumes less memory space. The performance of an algorithm is measured based on following properties:

**Time complexity:**

The time complexity of an algorithm is the amount of time required to complete the execution. The time complexity of an algorithm is denoted by the big O notation. Here, big O notation is the asymptotic notation to represent the time complexity. The time complexity is mainly calculated by counting the number of steps to finish the execution.

sum=0;

**for** i=1 to n

sum= sum + i ;

**return** sum;

In the above code, the time complexity of the loop statement will be at least n, and if the value of n increases, then the time complexity also increases. While the complexity of the code, i.e., return sum will be constant as its value is not dependent on the value of n and will provide the result in one step only. We generally consider the worst-time complexity as it is the maximum time taken for any given input size.

**Space complexity:**

Space complexity is the amount of memory used by the algorithm (including the input values to the algorithm) to execute and produce the result. So, to find space-complexity, it is enough to calculate the space occupied by the variables used in an algorithm/program.

**Space complexity = Auxiliary space + Input size.**

Auxiliary space: The extra space required by the algorithm, excluding the input size, is known as an auxiliary space. The space complexity considers both the spaces, i.e., auxiliary space, and space used by the input.

Asymptotic Analysis

Asymptotic notations are the mathematical notations used to describe the running time of an algorithm when the input tends towards a particular value or a limiting value. the ideal data structure is a structure that occupies the least possible time to perform all its operation and the memory space. our focus would be on finding the time complexity rather than space complexity. The commonly used asymptotic notations used for calculating the running time complexity of an algorithm is given below:

1. Big oh Notation (O) -- **Worst Case**
2. Omega Notation (Ω) – **Best Case**
3. Theta Notation (θ) -- **AVERAGE CASE**

**Big O Notation (O)**

The notation Ο(n) is the formal way to express the upper bound of an algorithm's running time. It measures **the worst-case** time complexity or the longest amount of time an algorithm can possibly take to complete.



**Omega Notation (Ω)**

Omega Notation (Ω) describes lower bound of an algorithm's running time. It measures the best amount of time an algorithm can possibly take to complete or the **best-case** time complexity. It determines what is the fastest time that an algorithm can run.



**Theta Notation (θ)**

The theta notation mainly describes the **average case** scenarios. It represents the realistic time complexity of an algorithm. Every time, an algorithm does not perform worst or best, in real-world problems, algorithms mainly fluctuate between the worst-case and best-case, and this gives us the average case of the algorithm. Big theta is mainly used when the value of worst-case and the best-case is same. It is the formal way to express both the upper bound and lower bound of an algorithm running time.



**Common Asymptotic Notations:**

|  |  |  |
| --- | --- | --- |
| constant | − | Ο(1) |
| logarithmic | − | Ο(log n) |
| linear | − | Ο(n) |
| n log n | − | Ο(n log n) |
| quadratic | − | Ο(n2) |
| cubic | − | Ο(n3) |
| polynomial | − | nΟ(1) |
| exponential | − | 2Ο(n) |

Approaches of Algorithm:

**The following are the approaches used after considering both the theoretical and practical importance of designing an algorithm:**

1. **Brute force algorithm:** The general logic structure is applied to design an algorithm. It is also known as an exhaustive search algorithm that searches all the possibilities to provide the required solution. Such algorithms are of two types:
   1. **Optimizing:** Finding all the solutions of a problem and then take out the best solution or if the value of the best solution is known then it will terminate if the best solution is known.
   2. **Sacrificing:** As soon as the best solution is found, then it will stop.
2. **Divide and conquer:** It is a very implementation of an algorithm. It allows you to design an algorithm in a step-by-step variation. It breaks down the algorithm to solve the problem in different methods, and valid output is produced for the valid input. This valid output is passed to some other function.
3. **Greedy algorithm:** It is an algorithm paradigm that makes an optimal choice on each iteration with the hope of getting the best solution. It is easy to implement and has a faster execution time. But there are very rare cases in which it provides the optimal solution.
4. **Dynamic programming:** It makes the algorithm more efficient by storing the intermediate results. It follows five different steps to find the optimal solution for the problem:
   1. It breaks down the problem into a sub problem to find the optimal solution.
   2. After breaking down the problem, it finds the optimal solution out of these sub problems.
   3. Stores the result of the sub problems is known as memorization.
   4. Reuse the result so that it cannot be recomputed for the same sub problems.
   5. Finally, it computes the result of the complex program.
5. **Branch and Bound Algorithm:** The branch and bound algorithm can be applied to only integer programming problems. This approach divides all the sets of feasible solutions into smaller subsets. These subsets are further evaluated to find the best solution.
6. **Randomized Algorithm:** As we have seen in a regular algorithm, we have predefined input and required output. Those algorithms that have some defined set of inputs and required output, and follow some described steps are known as deterministic algorithms. What happens that when the random variable is introduced in the randomized algorithm? In a randomized algorithm, some random bits are introduced by the algorithm and added in the input to produce the output, which is random in nature. Randomized algorithms are simpler and efficient than the deterministic algorithm.
7. **Backtracking:** Backtracking is an algorithmic technique that solves the problem recursively and removes the solution if it does not satisfy the constraints of a problem.

Types of Algorithms

**The following are the types of algorithms:**

1. **Search Algorithm**
2. **Sort Algorithm**

# **Search Algorithm**

On each day, we search for something in our day-to-day life. Similarly, with the case of computer, huge data is stored in a computer that whenever the user asks for any data then the computer searches for that data in the memory and provides that data to the user. There are mainly two techniques available to search the data in an array:

1. **Linear search**
2. **Binary search**

**Linear Search**

Linear search is a very simple algorithm that starts searching for an element or a value from the beginning of an array until the required element is not found. It compares the element to be searched with all the elements in an array, if the match is found, then it returns the index of the element else it returns -1. This algorithm can be implemented on the unsorted list.

**Binary Search**

A Binary algorithm is the simplest algorithm that searches the element very quickly. It is used to search the element from the sorted list. The elements must be stored in sequential order or the sorted manner to implement the binary algorithm. Binary search cannot be implemented if the elements are stored in a random manner. It is used to find the middle element of the list.

# Sorting Algorithms

Sorting algorithms are used to rearrange the elements in an array or a given data structure either in an ascending or descending order. The comparison operator decides the new order of the elements.

Why do we need a sorting algorithm?

1. An efficient sorting algorithm is required for optimizing the efficiency of other algorithms like binary search algorithm as a binary search algorithm requires an array to be sorted in a particular order, mainly in ascending order.
2. It produces information in a sorted order, which is a human-readable format.
3. Searching a particular element in a sorted list is faster than the unsorted list.