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 has a location and 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 and 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 and c++ language can be declared using \* (**asterisk symbol)/Dereference operator**. It is also known as indirection pointer used to dereference a pointer.

Example:

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. In 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:**

**Code:** void increase (void\* data, int psize){

if ( psize == sizeof(char) ){

char\* pchar;

pchar=(char\*) data;

++(\*pchar);

}

else if (psize == sizeof(int) ){

int\* pint;

pint=(int\*) data;

++(\*pint);

}

}

int main () {

char a = 'x';

int b = 1602;

increase (&a, sizeof(a));

increase (&b, sizeof(b));

cout << a << ", " << b << '\n';

return 0;

}

**Pointer to structure:**

Code: struct st {

int name;

float roll;

} user;

struct st \*shuvo = &user;

## 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. In pointer should be of the same type as specified in the pointer declaration. To overcome this problem, we use a pointer to void.
2. void pointer is also called as a **generic pointer.**
3. It does not have any standard data type.
4. A void pointer is created by using the keyword void.
5. It can be used to store an address of any variable.

int main () {

int n = 10;

void \*ptr = &n;

cout<< 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 ; // wild pointer

\*ptr = 10;

}

## Double Pointer (Pointer to Pointer)

pointer is used to store the address of a variable. we can also define a pointer to store the address of another pointer. Such pointer is known as a double pointer (pointer to pointer). The first pointer is used to store the address of a variable whereas the second pointer is used to store the address of the first pointer.



` **syntax:** int \*\*p;

**code:**

**int main () {**

**int number=50;**

**int \*p; //pointer to int**

**int \*\*p2; //pointer to pointer**

**p = &number; //stores the address of number variable**

**p2 = &p;**

**}**

## Pointer Arithmetic

**Increment:**

1. If we increment a pointer by 1, the pointer will start pointing to the immediate next location.
2. For 32-bit int variable, it will be incremented by 2 bytes.
3. For 64-bit int variable, it will be incremented by 4 bytes.

int main (){

int number=50;

int \*p;//pointer to int

p=&number;//stores the address of number variable

p=p+1;

}

**Decrement:**

1. If we decrement a pointer, it will start pointing to the previous location.
2. For 32-bit int variable, it will be decremented by 2 bytes.
3. For 64-bit int variable, it will be decremented by 4 bytes.

void main(){

int number=50;

int \*p;

p=&number;

p=p-1;

}

**Addition:**

1. We can add a value to the pointer variable.

int main(){

int number=50;

int \*p;//pointer to int

p=&number;//stores the address of number variable

printf("Address of p variable is %u \n",p);

p=p+3;   //adding 3 to pointer variable

printf("After adding 3: Address of p variable is %u \n",p);

return 0;

}

**Subtraction:**

1. we can subtract a value from the pointer variable.
2. Subtracting any number from a pointer will give an address.

Code:

new\_address = current\_address - (number \* size\_of(data type))

int main(){

int number=50;

int \*p;//pointer to int

p=&number;//stores the address of number variable

p=p-3; //subtracting 3 from pointer variable

return 0;

}

# 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
6. **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.
7. **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
3. **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;**

**}**

1. **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;**

**}**

1. **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;

}

1. **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
3. **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} ;**

**}**

1. **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.

Syntax: 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;

};

struct Student S1, S2;

1. Declaring Structure variables with structure definition

struct Student{

char name[25];

int age;

} 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

1. **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**

1. **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.**
2. **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**.
3. **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.**
4. **Attribute and Entity:**  An entity represents the class of certain objects. it contains various attributes. Each attribute represents the particular property of that entity.
5. **Field:** Field is a single elementary unit of information representing the attribute of an entity.
6. **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:** The primitive data structures are primitive data types that can hold a single value. They are --- **int, char, float, double, and pointer**
2. **Non-Primitive Data structure**: The non-primitive data structure is divided into two types:
3. **Linear Data Structure:** The arrangement of data in a sequential manner is known as a linear data structure. The data structures used for this purpose are Arrays, linked list, Stacks, and Queues. In these data structures, one element is connected to only one another element in a linear form.
4. **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**
5. **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.

**Linked List, Stack, Queue**

1. **Non-Linear Data Structure**
2. **Tree**
3. **graph**

# Linear data structure

# 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:**

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;

**float** avg = (marks\_1 + marks\_2) / 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 :** In single dimensional array data store in one sequential line .

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

1. **Multi-Dimensional Array:** Multidimensional array is various type. That can be two dimensional, three or more.

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

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

**Time complexity:**

1. O(1) to insert a single element
2. O(N) to insert all the array elements [where N is the size of the array]
3. Access/fatch elements in Array: O(1)
4. Searching in Array: O(N), where N is the size of the array.

## Basic Operations:

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.

Linked list contains two parts-

1. **Data part** - the simple variable
2. **The address part** - the pointer variable (address of next node).



**Representation of linked list:**

1. 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**:

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:**

**#include<iostream>**

**#include<stdlib.h>**

**using namespace std;**

**struct node{**

**int data;**

**struct node \*next;**

**};**

**struct node \*head;**

**void begin();**

**void last();**

**void position();**

**void del\_first();**

**void del\_last();**

**void show();**

**int main(){**

**int x;**

**while(true){**

**cout<<" 1 = insert in first"<<endl;**

**cout<<" 2 = insert in last"<<endl;**

**cout<<" 3 = insert in position"<<endl;**

**cout<<" 4 = delete in first"<<endl;**

**cout<<" 5 = delete in last"<<endl;**

**cout<<" 6 = delete in position"<<endl;**

**cout<<" 7 = show data"<<endl;**

**cout<<"plz enter your choice"<<endl;**

**cin>>x;**

**switch(x){**

**case 1:**

**begin();**

**break;**

**case 2:**

**last();**

**break;**

**case 3:**

**//position();**

**break;**

**case 4:**

**del\_first();**

**break;**

**case 5:**

**del\_last();**

**break;**

**case 7:**

**show();**

**break;**

**default:**

**cout<<"invalid";**

**}**

**}**

**}**

**void begin(){**

**int n;**

**cout<<"Enter value"<<endl;**

**cin>>n;**

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

**ptr->data=n;**

**ptr->next=NULL;**

**if(head==NULL){**

**head=ptr;**

**}else{**

**struct node \*temp = ptr;**

**ptr->next = head;**

**head=ptr;**

**}**

**}**

**void last(){**

**int n;**

**cout<<"Enter value"<<endl;**

**cin>>n;**

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

**ptr->data=n;**

**ptr->next=NULL;**

**if(head==NULL){**

**head=ptr;**

**}else{**

**struct node \*temp = head;**

**while(temp->next != NULL){**

**temp = temp->next;**

**}**

**temp->next=ptr;**

**}**

**}**

**void position(){**

**int n,c=0;**

**cout<<"Enter position"<<endl;**

**cin>>c;**

**cout<<"Enter value"<<endl;**

**cin>>n;**

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

**ptr->data=n;**

**ptr->next=NULL;**

**if(head==NULL){**

**if(c>1){**

**cout<<"invalid position"<<endl;**

**return;**

**}**

**head=ptr;**

**}else{**

**struct node \*temp = head;**

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

**temp = temp->next;**

**if(temp==NULL){**

**return;**

**}**

**}**

**ptr->next = temp->next;**

**temp->next = ptr;**

**}**

**}**

**void del\_first(){**

**struct node \*ptr;**

**if(head==NULL){**

**cout<<"list is empty"<<endl;**

**}else{**

**ptr = head;**

**head = ptr->next;**

**free(ptr);**

**}**

**}**

**void del\_last(){**

**struct node \*ptr;**

**if(head==NULL){**

**cout<<"list is empty"<<endl;**

**}else{**

**ptr = head;**

**head = ptr->next;**

**free(ptr);**

**}**

**}**

**void show(){**

**struct node \*temp = head;**

**if(temp==NULL){**

**cout<<"no data"<<endl;**

**}else{**

**while(temp!=NULL){**

**cout<<temp->data<<endl;**

**temp = temp->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).





struct node{

**int** data;

    struct node \*prev;

    struct node \*next;

}

#### Basic Operations:

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:**

### Circular Singly Linked List

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



**Basic Operations:**

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

1. A stack is a linear data structure that follows the principle of Last In First Out (LIFO).
2. Stack insertion and deletions are allowed only at one end, called top.
3. A stack is an abstract data type (ADT).
4. It can follow the LIFO (Last-In- First-Out) principle.
5. it contains only one pointer top pointer.
6. putting an item on top of the stack is called push and removing an item is called pop.



**Method of 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.

**ARRAY of stack Code:**

**#include<iostream>**

**int stack[100],i,j,choice=0,n,top=-1;**

**void push();**

**void pop();**

**void show();**

**void main (){**

**cout<<"Enter the number of elements in the stack "<<endl;**

**cin>>n;**

**while(choice != 4) {**

**cout<<"\n1.Push\n2.Pop\n3.Show";**

**cout<<"\n Enter your choice \n";**

**cin>>choice;**

**switch(choice) {**

**case 1:**

**push();**

**break;**

**case 2:**

**pop();**

**break;**

**case 3:**

**show();**

**break;**

**default:**

**cout<<"Please Enter valid choice ";**

**};**

**}**

**}**

**void push (){**

**int val;**

**if (top == n )**

**cout<<"n Overflow"<<endl;**

**else {**

**cout<<"Enter the value?";**

**cin>>val;**

**top = top +1;**

**stack[top] = val;**

**}**

**}**

**void pop (){**

**if(top == -1)**

**cout<<"Underflow";**

**else**

**top = top -1;**

**}**

**void show(){**

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

**cout<<stack[i]<<endl;**

**}**

**if(top == -1) {**

**cout<<"Stack is empty"<<endl;**

**}**

**}**

**Linked list stack code :**

# Queue:

1. A queue is a linear data structure that stores the elements sequentially.
2. Queue follows the First In First Out (FIFO) rule - the item that goes in first is the item that comes out first.
3. insert operations to be performed at one end called REAR and 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.
3. peek() − Gets the element at the front of the queue without removing it.
4. isfull() − Checks if the queue is full.
5. isempty() − Checks if the queue is empty.

**QUEUE with Array:**

#include <stdio.h>

#define SIZE 5

void enQueue(int);

void deQueue();

void display();

int items[SIZE], front = -1, rear = -1;

int main(){

deQueue();

enQueue(1);

enQueue(2);

enQueue(3);

enQueue(4);

enQueue(5);

enQueue(6);

display();

deQueue();

display();

return 0;

}

void enQueue(int value){

if (rear == SIZE - 1)

cout<<"\nQueue is Full!!";

else {

if (front == -1)

front = 0;

rear++;

items[rear] = value;

cout<<"\nInserted ->"<<value;

}

}

void deQueue(){

if (front == -1)

cout<<"\nQueue is Empty!!";

else {

cout<<"\nDeleted : "<<items[front];

front++;

if (front > rear)

front = rear = -1;

}

}

void display (){

if (rear == -1)

cout<<"\nQueue is Empty!!!";

else {

int i;

cout<<"\nQueue elements are:\n";

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

cout<<items[i];

}

cout<<"\n";

}

## Types of Queues

1. Simple Queue
2. Circular Queue
3. Priority Queue
4. Double Ended Queue

**Simple Queue:** simple queue, insertion takes place at the rear and removal occurs at the front. It strictly follows the FIFO (First in First out) rule.

**Circular Queue:** A circular queue is similar to a linear queue as it is also based on the FIFO (First In First Out) principle except that the last position is connected to the first position in a circular queue that forms a circle. It is also known as a Ring Buffer.

**Priority Queue:**

# 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.



## Tree Fundamental

**Basic Terminologies**

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. {B} is the parent node of {D, E}.
2. **Child node:** The node which is the immediate successor of a node is called the child node of that node. Examples: {D, E} are the child nodes of {B}.
3. **Parent:** The node which is a predecessor of a node is called the parent node of that node. {B} is the parent node of {D, E}.
4. **Sibling:** Children of the same parent node are called siblings. {D,E} are called siblings.
5. **Leaf Node or External Node:**The nodes which do not have any child nodes are called leaf nodes. {K, L, M, N, O, P} are the leaf nodes of the tree.
6. **Internal nodes:** A node has at least one child node known as an ***internal.***
7. **Ancestor node:**  Any predecessor nodes on the path of the root to that node are called Ancestors of that node. {A, B} are the ancestor nodes of the node {E}.
8. **Descendant:**  Any successor node on the path from the leaf node to that node. {E,I} are the descendants of the node {B}.
9. Level of a node: The count of edges on the path from the root node to that node. The root node has level 0.

**Properties of Tree data structure**

1. **Number of edges**: An edge can be defined as the connection between two nodes. If a tree has N nodes then it will have (N-1) edges. There is only one path from each node to any other node of the tree.
2. **Depth of a node:** The depth of a node is defined as the length of the path from the root to that node. Each edge adds 1 unit of length to the path. So, it can also be defined as the number of edges in the path from the root of the tree to the node.
3. **Height of a node**: The height of a node can be defined as the length of the longest path from the node to a leaf node of the tree.
4. **Height of the Tree**: The height of a tree is the length of the longest path from the root of the tree to a leaf node of the tree.
5. **Degree of a Node:** The total count of subtrees attached to that node is called the degree of the node. The degree of a leaf node must be 0. The degree of a tree is the maximum degree of a node among all the nodes in the tree.

**Implementation of Tree:**

The tree data structure can be created by creating the nodes dynamically with the help of the pointers. 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.

**Code:**

**struct node {**

**int data;**

**struct node \*left;**

**struct node \*right;**

**}**



**Types of Tree data structure**

1. General tree
2. Binary tree
3. Balanced tree
4. Binary search tree
5. AVL Tree
6. B-Tree

## Binary Tree

1. A binary tree is a tree data structure in which each parent node can have at most two children.
2. The maximum number of nodes at level ‘l’ of a binary tree is 2l.
3. The Maximum number of nodes in a binary tree of height ‘h’ is 2h – 1.
4. The minimum number of nodes possible at height h is equal to h+1.
5. The minimum height h = log2(n+1) - 1
6. The maximum height can h= n-1



**Representation of Binary Tree:**

struct node {

int data;

struct node \*leftChild;

struct node \*rightChild;

};

**Code:**

**struct node {**

**int data;**

**struct node \*left;**

**struct node \*right;**

**};**

**struct node \*newNode(int data) {**

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

**node->data = data;**

**node->left = NULL;**

**node->right = NULL;**

**return (node);**

**}**

**int main () {**

**struct node \*root = newNode(1);**

**root->left = newNode(2);**

**root->right = newNode(3);**

**root->left->left = newNode(4);**

**}**

**Types of Binary Tree**

1. **Full/ proper/ strict Binary tree:**
2. A Binary Tree is a full binary tree if every node has 0 or 2 children
3. The number of leaves is i + 1.
4. The total number of nodes is 2i + 1.
5. The number of internal nodes is (n – 1) / 2.
6. The number of leaves is (n + 1) / 2.
7. The total number of nodes is 2l – 1.
8. The number of internal nodes is l – 1.
9. The number of leaves is at most 2λ - 1.



1. **Complete Binary tree:**
2. The complete binary tree is a tree in which all the nodes are completely filled except the last level.
3. In the last level, all the nodes must be as left as possible.
4. The maximum number of nodes in complete binary tree is 2h+1 - 1.
5. The minimum number of nodes in complete binary tree is 2h.
6. The minimum height of a complete binary tree is log2(n+1) - 1.
7. The maximum height of a complete binary tree is



1. **Perfect Binary tree**
2. A tree is a perfect binary tree if all the internal nodes have 2 children, and all the leaf nodes are at the same level.
3. A perfect binary tree of height h has 2h + 1 – 1 node.
4. A perfect binary tree with n nodes has height log(n + 1) – 1 = Θ(ln(n)).
5. A perfect binary tree of height h has 2h leaf nodes.
6. The average depth of a node in a perfect binary tree is Θ(ln(n)).



1. **Degenerate Binary tree**
2. A Tree where every internal node has one child.
3. Such trees are performance-wise same as linked list



1. **Skewed Binary Trees:** A skewed binary tree is a type of binary tree in which all the nodes have only either one child or no child. There are 2 special types of skewed tree:
2. Left Skewed Binary Tree:   
   These are those skewed binary trees in which all the nodes are having a left child or no child at all. It is a left side dominated tree. All the right children remain as null.
3. Right Skewed Binary Tree:   
   These are those skewed binary trees in which all the nodes are having a right child or no child at all. It is a right side dominated tree. All the left children remain as null.



**Balanced Binary tree**

1. difference between the left and the right subtree for any node is not more than one
2. the left subtree is balanced
3. the right subtree is balanced

## Binary Search tree

1. the value of left node must be smaller than the parent node,
2. the value of right node must be greater than the parent node.



## AVL Tree

1. AVL Tree can be defined as height balanced binary search tree in which each node is associated with a balance factor which is calculated by subtracting the height of its right sub-tree from that of its left sub-tree.
2. Tree is said to be balanced if balance factor of each node is in between -1 to 1, otherwise, the tree will be unbalanced and need to be balanced.
3. Balance Factor (k) = height (left(k)) - height (right(k))
4. If balance factor of any node is 1, it means that the left sub-tree is one level higher than the right sub-tree.
5. If balance factor of any node is 0, it means that the left sub-tree and right sub-tree contain equal height.
6. If balance factor of any node is -1, it means that the left sub-tree is one level lower than the right sub-tree.

**AVL Rotations:**

We perform rotation in AVL tree only in case if Balance Factor is other than -1, 0, and 1. There are basically four types of rotations which are as follows:

1. L L rotation: Inserted node is in the left subtree of left subtree of A
2. R R rotation: Inserted node is in the right subtree of right subtree of A
3. L R rotation: Inserted node is in the right subtree of left subtree of A
4. R L rotation: Inserted node is in the left subtree of right subtree of A

Where node A is the node whose balance Factor is other than -1, 0, 1.

The first two rotations LL and RR are single rotations and the next two rotations LR and RL are double rotations. For a tree to be unbalanced, minimum height must be at least 2, Let us understand each rotation

1. **RR Rotation:** When BST becomes unbalanced, due to a node is inserted into the right subtree of the right subtree of A, then we perform RR rotation, RR rotation is an anticlockwise rotation, which is applied on the edge below a node having balance factor -2



node A has balance factor -2 because a node C is inserted in the right subtree of A right subtree. We perform the RR rotation on the edge below A.

1. **LL Rotation**: When BST becomes unbalanced, due to a node is inserted into the left subtree of the left subtree of C, then we perform LL rotation, LL rotation is clockwise rotation, which is applied on the edge below a node having balance factor 2.



node C has balance factor 2 because a node A is inserted in the left subtree of C left subtree. We perform the LL rotation on the edge below A.

1. **LR Rotation:** Double rotations are bit tougher than single rotation which has already explained above. LR rotation = RR rotation + LL rotation, i.e., first RR rotation is performed on subtree and then LL rotation is performed on full tree, by full tree we mean the first node from the path of inserted node whose balance factor is other than -1, 0, or 1.

* A node B has been inserted into the right subtree of The left subtree of C, because of which C has become an unbalanced node having balance factor 2. This case is L R rotation where: Inserted node is in the right subtree of left subtree of C



* As LR rotation = RR + LL rotation, hence RR (anticlockwise) on subtree rooted at A is performed first. By doing RR rotation, node A, has become the left subtree of B.



* After performing RR rotation, node C is still unbalanced, i.e., having balance factor 2, as inserted node A is in the left of left of C
* Now we perform LL clockwise rotation on full tree, i.e. on node C. node C has now become the right subtree of node B, A is left subtree of B



* Balance factor of each node is now either -1, 0, or 1, i.e. BST is balanced now.



1. **RL Rotation:** As already discussed, that double rotations are bit tougher than single rotation which has already explained above. R L rotation = LL rotation + RR rotation, i.e., first LL rotation is performed on subtree and then RR rotation is performed on full tree, by full tree we mean the first node from the path of inserted node whose balance factor is other than -1, 0, or 1.

* A node B has been inserted into the left subtree of C the right subtree of A, because of which A has become an unbalanced node having balance factor - 2. This case is RL rotation where: Inserted node is in the left subtree of right subtree of A



* As RL rotation = LL rotation + RR rotation, hence, LL (clockwise) on subtree rooted at C is performed first. By doing RR rotation, node C has become the right subtree of B.



* After performing LL rotation, node A is still unbalanced, i.e. having balance factor -2, which is because of the right-subtree of the right-subtree node A.



* Now we perform RR rotation (anticlockwise rotation) on full tree, i.e. on node A. node C has now become the right subtree of node B, and node A has become the left subtree of B.



* Balance factor of each node is now either -1, 0, or 1, i.e., BST is balanced now.



# Graph

1. A Graph is a non-linear data structure consisting of vertices and edges.
2. The edges connect any two nodes in the graph, and the nodes are also known as vertices.
3. Formally, a graph is a pair of sets (V, E), where V is the set of vertices and E is the set of edges, connecting the pairs of vertices.



In the above graph,

Vertices = {a, b, c, d, e}

Edges = {ab, ac, bd, cd, de}

**Graph Terminology:**

1. Path : A path can be defined as the sequence of nodes that are followed in order to reach some terminal node V from the initial node U.
2. Closed Path : A path will be called as closed path if the initial node is same as terminal node. A path will be closed path if V0=VN.
3. Simple Path : If all the nodes of the graph are distinct with an exception V0=VN, then such path P is called as closed simple path.
4. Cycle : A cycle can be defined as the path which has no repeated edges or vertices except the first and last vertices.
5. Degree of the Node : A degree of a node is the number of edges that are connected with that node. A node with degree 0 is called as isolated node.
6. Adjacent Nodes : If two nodes u and v are connected via an edge e, then the nodes u and v are called as neighbours or adjacent nodes.

## Types of Graphs in Data Structures

1. **Finite Graph**: The graph G=(V, E) is called a finite graph if the number of vertices and edges in the graph is limited in number



1. **Infinite Graph:** The graph G=(V, E) is called a finite graph if the number of vertices and edges in the graph is interminable.



1. **Trivial Graph:** A graph G= (V, E) is trivial if it contains only a single vertex and no edges.



1. **Simple Graph:** If each pair of nodes or vertices in a graph G=(V, E) has only one edge, it is a simple graph. As a result, there is just one edge linking two vertices, depicting one-to-one interactions between two elements.



1. **Multi Graph:** If there are numerous edges between a pair of vertices in a graph G= (V, E), the graph is referred to as a multigraph. There are no self-loops in a Multigraph.



1. Null Graph: It's a reworked version of a trivial graph. If several vertices but no edges connect them, a graph G= (V, E) is a null graph.



1. **Complete Graph:** If a graph G= (V, E) is also a simple graph, it is complete. Using the edges, with n number of vertices must be connected. It's also known as a full graph because each vertex's degree must be n-1.
2. **Pseudo Graph:** If a graph G= (V, E) contains a self-loop besides other edges, it is a pseudograph.



**9. Regular Graph:** If a graph G= (V, E) is a simple graph with the same degree at each vertex, it is a regular graph. As a result, every whole graph is a regular graph.



1. **Weighted Graph:** A graph G= (V, E) is called a labeled or weighted graph because each edge has a value or weight representing the cost of traversing that edge.



1. **Directed Graph:** A directed graph also referred to as a digraph, is a set of nodes connected by edges, each with a direction.



1. **Undirected Graph:** An undirected graph comprises a set of nodes and links connecting them. The order of the two connected vertices is irrelevant and has no direction. You can form an undirected graph with a finite number of vertices and edges.



1. **Connected Graph:** If there is a path between one vertex of a graph data structure and any other vertex, the graph is connected.



1. Disconnected Graph: When there is no edge linking the vertices, you refer to the null graph as a disconnected graph.



1. **Cyclic Graph:** If a graph contains at least one graph cycle, it is cyclic.



1. **Acyclic Graph:** When there are no cycles in a graph, it is called an acyclic graph.



1. **Directed Acyclic Graph:** It's also known as a directed acyclic graph (DAG), and it's a graph with directed edges but no cycle. It represents the edges using an ordered pair of vertices since it directs the vertices and stores some data.



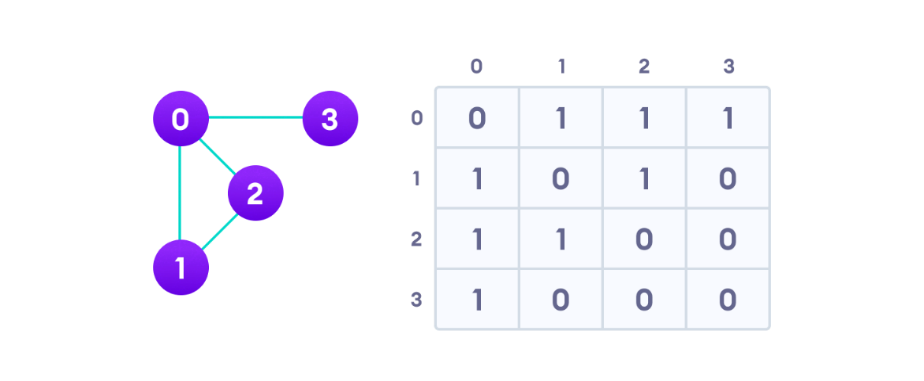
1. **Subgraph:** The vertices and edges of a graph that are subsets of another graph are known as a subgraph.



## Graph Representation

**Adjacency Matrix:**

1. An adjacency matrix is a 2D array of V x V vertices. Each row and column represent a vertex.
2. If the value of any element a[i][j] is 1, it represents that there is an edge connecting vertex i and vertex j.



Since it is an undirected graph, for edge (0,2), we also need to mark edge (2,0); making the adjacency matrix symmetric about the diagonal.

Edge lookup (checking if an edge exists between vertex A and vertex B) is extremely fast in adjacency matrix representation but we must reserve space for every possible link between all vertices (V x V), so it requires more space.

**Adjacency List:**

1. An adjacency list represents a graph as an array of linked lists.
2. The index of the array represents a vertex and each element in its linked list represents the other vertices that form an edge with the vertex.

An adjacency list is efficient in terms of storage because we only need to store the values for the edges. For a graph with millions of vertices, this can mean a lot of saved space.

# 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.

Example:

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 .

Step 5: store the result in the sum variable. sum= a + b.

Step 6: Print sum

Step 7: 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 Analysis:**

You can find more than one algorithms for solving a particular problem after that you have to analyze them and use the most efficient one.

The analysis of an algorithm is done base on its efficiency. The two important terms used for the analysis of an algorithm are “Priori Analysis” and “Posterior Analysis”.

1. **A Priori Analysis** − It is done before the actual implementation of the algorithm when the algorithm is written in the general theoretical language. In this, the efficiency of the algorithm is calculated based on its complexity. It is just an approximate analysis.
2. **A Posterior Analysis** − It is done after the actual implementation and execution of the algorithm using any programming language like C, C++, Java, or Python. It is the actual analysis in which the space and time complexity of the algorithm is calculated more correctly.

|  |  |
| --- | --- |
| **A Posteriori analysis** | **A priori analysis** |
| Posteriori analysis is a relative analysis. | Priori analysis is an absolute analysis. |
| It is dependent on language of compiler and type of hardware. | It is independent of language of compiler and types of hardware. |
| It will give exact answer. | It will give approximate answer. |
| It doesn’t use asymptotic notations to represent the time complexity of an algorithm. | It uses the asymptotic notations to represent how much time the algorithm will take in order to complete its execution. |
| The time complexity of an algorithm using a posteriori analysis differ from system to system. | The time complexity of an algorithm using a priori analysis is same for every system. |
| If the time taken by the program is less, then the credit will go to compiler and hardware. | If the algorithm running faster, credit goes to the programmer. |
| It is done after execution of an algorithm. | It is done before execution of an algorithm. |
| It is costlier than priori analysis because  of requirement of software and hardware for execution. | It is cheaper than Posteriori Analysis. |
| Maintenance Phase is required to tune the algorithm. | Maintenance Phase is not  required to tune the algorithm. |

**Algorithm Complexity Analysis:**

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:

1. **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.

1. **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**
4. **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.



1. **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.



1. **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:**

1. Constant − Ο(1)
2. logarithmic − Ο(log n)
3. linear − Ο(n)
4. n log n − Ο(n log n)
5. quadratic − Ο(n2)
6. cubic − Ο(n3)
7. polynomial − nΟ(1)
8. exponential − 2Ο(n)

# Algorithms Classification

1. **Classification by Implementation Method:**
2. **Recursion or Iteration:** A recursive algorithm is an algorithm which calls itself again and again until a base condition is achieved. Example: The Tower of Hanoi.
3. **Exact or Approximate:** Algorithms that are capable of finding an optimal solution for any problem are known as the exact algorithm. Example: Approximation algorithms
4. **Serial or Parallel or Distributed Algorithms:** In serial algorithms, one instruction is executed at a time while parallel algorithms are those in which we divide the problem into sub problems and execute them on different processors
5. **Classification by Design Method:**
6. Greedy Method:
7. Divide and Conquer:
8. Dynamic Programming:
9. Linear Programming:
10. Reduction (Transform and Conquer):
11. Backtracking:
12. Branch and Bound:
13. **Classification by Design Approaches:**
14. **Top-Down Approach:** In the top-down approach, a large problem is divided into small sub-problem. and keep repeating the process of decomposing problems until the complex problem is solved. Breaking down a complex problem into smaller, more manageable sub-problems and solving each sub-problem individually. Designing a system starting from the highest level of abstraction and moving towards the lower levels.
15. **Bottom-up approach:** The bottom-up approach is also known as the reverse of top-down approaches. In approach different, part of a complex program is solved using a programming language and then this is combined into a complete program.

Building a system by starting with the individual components and gradually integrating them to form a larger system. Solving sub-problems first and then using the solutions to build up to a solution of a larger problem.

1. **Other Classifications:**
2. **Randomized Algorithms:** Algorithms that make random choices for faster solutions are known as randomized algorithms.  Example: Randomized Quicksort Algorithm.
3. **Classification by complexity:** Algorithms that are classified on the basis of time taken to get a solution to any problem for input size. This analysis is known as time complexity analysis.   
   Example: Some algorithms take O(n), while some take exponential time.
4. **Classification by Research Area: In** CS each field has its own problems and needs efficient algorithms.  Example: Sorting Algorithm, Searching Algorithm, Machine Learning etc.

# Algorithm Design Paradigms

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

1. **Brute force approach:**

A brute force approach is an approach that finds all the possible solutions to find a satisfactory solution to a given problem. The brute force algorithm tries out all the possibilities till a satisfactory solution is not found. Brute force algorithm is a technique that guarantees solutions for problems of any domain helps in solving the simpler problems and also provides a solution that can serve as a benchmark for evaluating other design techniques, but takes a lot of run time and inefficient.

The time complexity of brute force is O(mn), which is sometimes written as O(n\*m).

1. **Backtracking approach:**

Backtracking is an algorithmic technique where the goal is to get all solutions to a problem using the brute force approach. It consists of building a set of all the solutions incrementally. Since a problem would have constraints, the solutions that fail to satisfy them will be removed.

It uses recursive calling to find a solution set by building a solution step by step, increasing levels with time. In order to find these solutions, a search tree named state-space tree is used. In a state-space tree, each branch is a variable, and each level represents a solution.

Backtracking Algorithm Applications

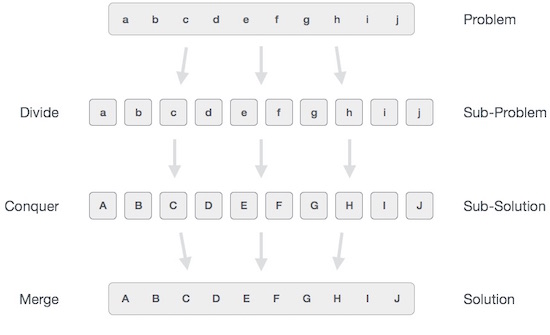
1. Hamiltonian Paths.
2. N Queen problem.
3. Maze solving problem.
4. The Knight's tour problem.
5. **Divide and conquer approach:**

A divide and conquer algorithm is a strategy of solving a large problem by

1. breaking the problem into smaller sub-problems
2. solving the sub-problems, and
3. combining them to get the desired output.

**How Divide and Conquer Algorithms Work?**

1. **Divide:** Divide the given problem into sub-problems using recursion.
2. **Conquer**: Solve the smaller sub-problems recursively. If the subproblem is small enough, then solve it directly.
3. **Combine:** Combine the solutions of the sub-problems that are part of the recursive process to solve the actual problem.



Algorithms are based on divide-and-conquer programming approach --

1. Merge Sort
2. Quick Sort
3. Binary Search
4. Strassen's Matrix Multiplication
5. Closest pair (points)
6. Cooley–Tukey Fast Fourier Transform (FFT) algorithm
7. Karatsuba algorithm.
8. **Greedy approach:**

A greedy algorithm is an approach for solving a problem by selecting the best option available at the moment. It doesn't worry whether the current best result will bring the overall optimal result. The algorithm never reverses the earlier decision even if the choice is wrong. It works in a top-down approach. This algorithm may not produce the best result for all the problems. It's because it always goes for the local best choice to produce the global best result.

We can choose greedy approach if the problem has the following properties:

* 1. **Greedy Choice Property:** If an optimal solution to the problem can be found by choosing the best choice at each step without reconsidering the previous steps once chosen, the problem can be solved using a greedy approach. This property is called greedy choice property.
  2. **Optimal Substructure:** If the optimal overall solution to the problem corresponds to the optimal solution to its sub problems, then the problem can be solved using a greedy approach. This property is called optimal substructure.

**Standard Greedy Algorithms:**

1. Activity Selection Problem
2. Job Sequencing Problem
3. **Huffman Coding**
4. **Huffman Decoding**
5. Water Connection Problem
6. Minimum Swaps for Bracket Balancing
7. Egyptian Fraction
8. Policemen catch thieves
9. Fitting Shelves Problem
10. Assign Mice to Holes

**Greedy Problems on Array:**

1. Minimum product subset of an array
2. Maximize array sum after K negations using Sorting
3. Minimum sum of product of two arrays
4. Minimum sum of absolute difference of pairs of two arrays
5. Minimum increment/decrement to make array non-Increasing
6. Sorting array with reverse around middle
7. Sum of Areas of Rectangles possible for an array
8. Largest lexicographic array with at-most K consecutive swaps
9. Partition into two subarrays of lengths k and (N – k) such that the difference of sums is maximum

**Greedy Problems on Operating System:**

1. First Fit algorithm in Memory Management
2. Best Fit algorithm in Memory Management
3. Worst Fit algorithm in Memory Management
4. Shortest Job First Scheduling
5. Job Scheduling with two jobs allowed at a time
6. Program for Optimal Page Replacement Algorithm

**Greedy Problems on Graph:**

1. **Kruskal’s Minimum Spanning Tree**
2. **Prim’s Minimum Spanning Tree**
3. Boruvka’s Minimum Spanning Tree
4. **Dijkastra’s Shortest Path Algorithm**
5. Dial’s Algorithm
6. Minimum cost to connect all cities
7. Max Flow Problem Introduction
8. Number of single cycle components in an undirected graph

**Approximate Greedy Algorithm for NP Complete:**

1. Set cover problem
2. Bin Packing Problem
3. Graph Coloring
4. K-centers problem
5. Shortest superstring problem
6. Approximate solution for Travelling Salesman Problem using MST

**Greedy for Special cases of DP:**

1. **Fractional Knapsack Problem**
2. Minimum number of coins required
3. **Dynamic programming approach:**

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.

**Approaches of dynamic programming**

1. **Top-down approach**: The top-down approach follows the memorization technique, while bottom-up approach follows the tabulation method. Here memorization is equal to the sum of recursion and caching. Recursion means calling the function itself, while caching means storing the intermediate results.
2. B**ottom-Up approach:** The bottom-up approach is also one of the techniques which can be used to implement the dynamic programming. It uses the tabulation technique to implement the dynamic programming approach. It solves the same kind of problems but it removes the recursion. If we remove the recursion, there is no stack overflow issue and no overhead of the recursive functions. In this tabulation technique, we solve the problems and store the results in a matrix.

The following computer problems can be solved using dynamic programming approach −

1. Fibonacci number series
2. Knapsack problem
3. Tower of Hanoi
4. All pair shortest path by Floyd-Warshall
5. Shortest path by Dijkstra
6. Project scheduling
7. Binomial Coefficient
8. Coin change problem
9. Floyd Warshall Algorithm
10. Bellman–Ford Algorithm
11. **Branch and Bound approach:**

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.

1. **Randomized approach:** 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.

# ****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
3. **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.
4. **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.

# **Traversal Algorithms**

## Tree Traversal

Tree Traversal using Depth-First Search (DFS)

1. **Preorder Traversal (current-left-right):** Visit the current node before visiting any nodes inside the left or right subtrees. Here, the traversal is root – left child – right child. It means that the root node is traversed first then its left child and finally the right child.
2. **Inorder Traversal (left-current-right):** Visit the current node after visiting all nodes inside the left subtree but before visiting any node within the right subtree. Here, the traversal is left child – root – right child.  It means that the left child is traversed first then its root node and finally the right child.
3. **Postorder Traversal (left-right-current):** Visit the current node after visiting all the nodes of the left and right subtrees.  Here, the traversal is left child – right child – root.  It means that the left child has traversed first then the right child and finally its root node.

### Tree Traversal using Breadth-First Search (BFS)

1. Level Order Traversal:  Visit nodes level-by-level and left-to-right fashion at the same level. Here, the traversal is level-wise. It means that the most left child has traversed first and then the other children of the same level from left to right have traversed.

## Graph Traversal Algorithm

The process of visiting or updating each vertex in a graph is known as graph traversal. The sequence in which they visit the vertices is used to classify such traversals. Graph traversal is a subset of tree traversal.

There are two techniques to implement a graph traversal algorithm:

1. Breadth-first search
2. Depth-first search

Breadth-first search

There are many ways to traverse the graph, but among them, BFS is the most commonly used approach. It is a recursive algorithm to search all the vertices of a tree or graph data structure. BFS puts every vertex of the graph into two categories - visited and non-visited. It selects a single node in a graph and, after that, visits all the nodes adjacent to the selected node.