

LECTURE:01

DATA STRUCTURES AND ALGORITHMS

CONCEPTS OF DATA STRUCTURES

Data structures and algorithms (DSA) are two important aspects of any programming language. Every programming language has its own data structures and different types of algorithms to handle these data structures.

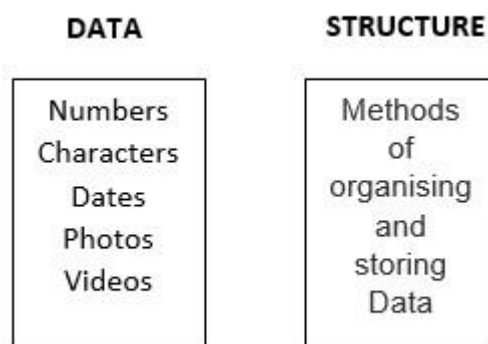
Data Structures are used to organise and store data to use it in an effective way when performing data operations.

Algorithm is a step-by-step procedure, which defines a set of instructions to be executed in a certain order to get the desired output. Algorithms are generally created independent of underlying languages, i.e. an algorithm can be implemented in more than one programming language.

Intoduction to data structures

The name "Data Structure" is a combination of two words: "Data" and "Structure". Let's go over each word individually:

- **Data:** Data is information that must be processed and stored in a computer. Data can be anything from numbers to characters to dates to photos to videos.
- **Structure:** Structure refers to how data is structured. It specifies the relationship between various data elements and how they are kept in memory. The structure has an impact on the efficiency of data operations.



What is Data Structure?

*a **data structure** is a particular way of organising data in a computer so that it can be used effectively. the idea is to reduce the space and time complexities of different tasks.*

or

*a **data structure** is a systematic way of storing and managing data in a computer so that it can be retrieved and used efficiently.*

Importance of Data Structures

Data structures are a fundamental concept in computer science and play a crucial role in many aspects of software development. Here are some of the reasons why data structures are important:

1. **Efficient storage and retrieval of data**

Data structures allow you to store and retrieve data in an efficient manner. For example, using a hash table data structure, you can access an element in constant time, whereas searching for an element in an unordered list would take linear time.

2. **Improved performance**

Data structures can help improve the performance of your code by reducing the time and space complexity of algorithms. For example, using a binary search tree instead of a linear search can significantly reduce the time it takes to find a specific element in a large dataset.

3. **Better problem solving**

Data structures can help you solve complex problems by breaking them down into smaller, more manageable parts. For example, using a graph data structure can help you solve problems related to network flow or finding the shortest path between two points.

4. **Abstraction**

Data structures provide a way to abstract the underlying complexity of a problem and simplify it, making it easier to understand and solve.

5. **Reusability**

Data structures can be used in many different algorithms and applications, making them a useful tool in your programming toolbox.

In short, data structures are important because they provide a way to organize and manage data in a way that is efficient, flexible, and scalable.

Whether you are working on a large-scale software project or a simple program, a solid understanding of data structures is essential for success.

GOALS OF DATA STRUCTURE

during the data structure implementation, its operations have two goals, these are:-

- a. **Correctness:** for all the inputs within a program, the data structures are designed to operate correctly for a specific problem intended to be solved.
- b. **Efficiency:** Data are processed at the required speed without using much computer resources, and stored in a specific memory location.

FEATURES OF DATA STRUCTURES

some of the essential features of data structures are described as follows:

- a. **Reusability:** by implementing quality data structures, it is possible to develop reusable software which tends to be cost-effective and time-saving.
- b. **Robustness:** generally, all developers expect to produce software that generates correct output for every possible input provided to it and executes efficiently on all the hardware platforms. This kind of robust software must manage both valid and invalid inputs.
- c. **Adaptability:** Some data structures are more adaptable to certain types of data and operations than others. For example, a stack is more suitable for problems that require Last-In-First-Out (LIFO) behavior, while a queue is better suited for problems that require First-In-First-Out (FIFO) behavior.
- d. **Time and Space Complexity:** Data structures can have different time and space complexities, depending on the operations they support and the way they organize data.

FACTORS TO CONSIDER WHEN SELECTING A DATA STRUCTURE

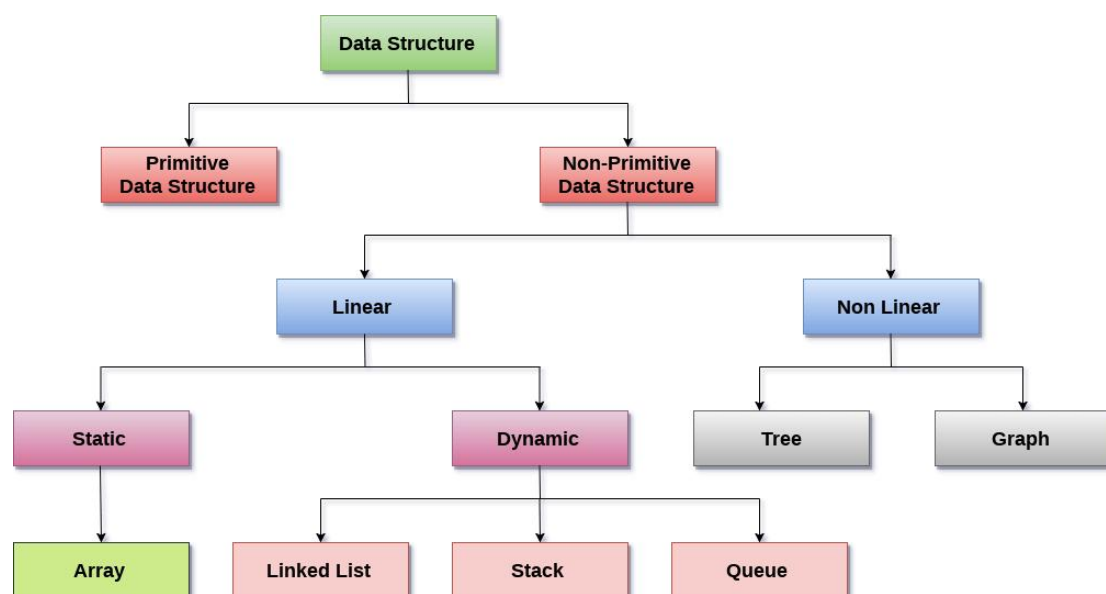
it is essential to choose the appropriate data structure for each tasks. the factor to consider when selecting a proper data structure include the following:-

- Type of information to be stored.
- The uses of data or information.
- Ways or methods to organise the data.
- The aspects of memory and storage arrangement plan and their management.
- Storage devices for data after being generated.

CLASSIFICATION OF DATA STRUCTURE

The data structures can broadly be categorised into two distinct types:-

- Primitive data structure.
- Non-primitive data structure.



from the above figure, we have classified data structures based on various aspects;-

- Linear and Non-linear data structure based on the arrangement or organisation of the structures.

- b. static data structure and dynamic data structure based on the memory size of the structure.
- c. Based on the nature of the operation that can be conducted on the unit of respected data, the primitive data structure can store only values of the same data type, while non-primitive can store values with more than one data type.

LECTURE:02

PRIMITIVE DATA STRUCTURE.

The *primitive data structure* is a fundamental category of data type. its operations and functionalities are only performed at the machine level because they consist of numbers and characters built into the program.

They are also called *pre-defined* or *built-in* data structures. because user does not define them.

Example of primitive data structures are *integers, floats, doubles, characters, boolean, pointers and strings*. the primitive data structure stores only one type of data or may contain empty values.

Integer

- ✧ represents numbers without decimal points, foreexample 5,20,4,1244.
- ✧ it has a 32 bits size.

Float

- ✧ represent variables defined with fractional values. these variables have decimal points, ie. 10.2, 33.6, 7484.4
- ✧ it has a 32 bits size containing up to 7 decimals.
- ✧ floats are often used in applications where memory is a critical resource.

Double

- ✧ it is the same as float type but has 64 bits size and contains up to 15 decimals.
- ✧ for example, -1.79769313486231E308 to -4.94065645841247E324 for negative values and 4.94065645841247E324 to 1.79769313486231E308 for positive values.
- ✧ doubles are commonly used in applications where higher precision is essential, such as scientific calculations, financial applications and a situations where a wide range of values is encountered.

Characters

- ✧ represents te variables that use symbols/characters or letters. foreexample: r,g,o,X,U, "-".
- ✧ it has a 16 bits size.

Boolean

- ✧ represents logical values such as *true* or *false*, *yes* or *no*, 1 or 0.
- ✧ it has a 16 bits size.

NON-PRIMITIVE DATA STRUCTURE

The non-primitive data structures are not predefined in programming languages, and they are created by the users. these structures are used to organize and manage a collection of data items.unlike primitive data types(ie. intergers, floats, characters), which represent single value, non-primitive can hold multiple values and have more complex organizational patterns.

- ✧ Is a user-defined data structure that can hold several values in adjoining or random locations.ie. they can not be empty.
- ✧ it should have a specific function.
- ✧ The *non-primitive data structure* is further categorised into *Linear* and *Non-linear data structures* based on the arrangement and organisation of data.

LINEAR DATA STRUCTURE

The linear data structure store data in a sequence, one after another and the data is accessed from one place and continues to others sequentially.

The linear data structure is a data structure in which it's elements are linked with the subsequent one sequentially.

There are two techniques for representing Linear structure within a memory.

- ✧ *Arrays*(linear relationship among all the elements described using linear memory location).
- ✧ *Linked Lists*(linear relationship among all the elements using pointers or links).

Furthermore,the Linear Data Structure is classified into two types based on how data can be stored:

- ✧ Static Data Structures
- ✧ Dynamic Data Structures

The Arrays, Stack, Linked List and Queue are examples of linear data structures.

STATIC DATA STRUCTURES

Is a method of storing data where the amount of data stored and the memory used to hold it are fixed.

In a static data structure, the content of the data structure can be modified without changing the memory space allocated to it. the size of structure is fixed and permanent at compile time.

Accessing individual data elements within a static structure is speedy as their memory location is fixed.

example of static data structure is an *Array*.

DYNAMIC DATA STRUCTURE

In dynamic data structure, the size of the structure is not fixed and can be modified when operations are executed on it.

Dynamic data structures aim to facilitate the changes of the data structures in the run time.

examples of dynamic data structures are stacks, linked lists, and queues.

NON-LINEAR DATA STRUCTURE

In non-linear data structure, the elements may link to more than one element, although their data items are not sequential.

example of non-linear data structure is a tree.

Meaning of static data type

The *static data structure* has a fixed size of the memory structure. The content of data structure can be modified without changing the assigned memory space.

ADVANTAGES OF STATIC DATA STRUCTURES.

the following are the advantages of static data structures:-

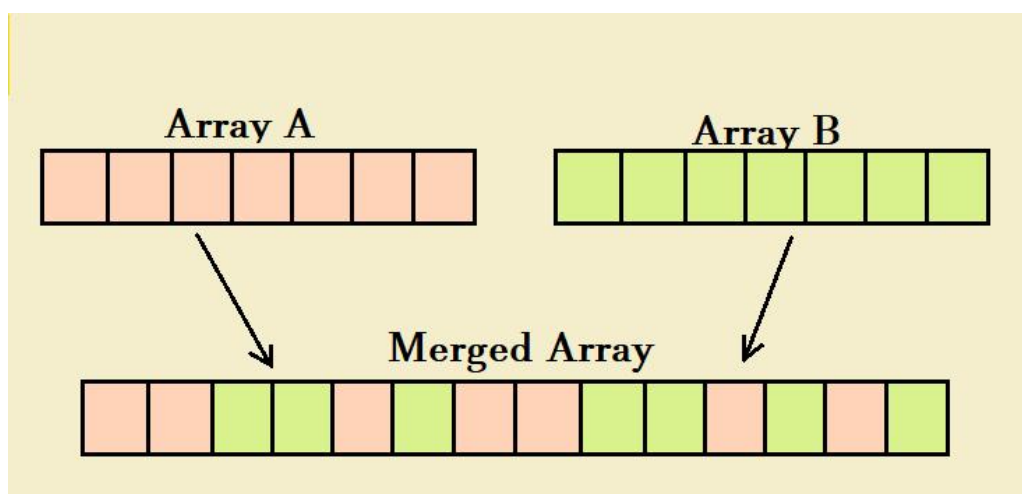
- a. **Easier to comprehend:** it is easier to define and use static data structure than the dynamic one. this is because the size and structure of static data structures are defined at compile time and do not change during the program execution.
- b. **size:** has a fixed size.
- c. **Faster:** the memory for static data is assigned during the compilation time, making the program execute more quickly than when it has to stop and allocate the required memory each time. An array index can randomly access any element of an array.

DISADVANTAGES OF STATIC DATA STRUCTURES.

- a. **Memory Wastage:** declaring an array with a large size is a memory wastage. for example if you declare an array of 10 and only a tiny portion less than the declared is used, the rest of the memory will be wasted.
- b. **insufficient memory space:** declaring an array with a fixed size leads to a situation where the size of an array is less than what you intended to use, for example, `int y[6]`; it has 6 elements, but the program needs 10 memory.
- c. **slow:** some algorithms are slower when done statically than when done dynamically.

for example:- how would you merge two list?

solution:- the solution with the static arrays is to define a more extensive array and copy the two arrays into a more considerable array.



- d. **not ordinary:** more suitable algorithms can be implemented dynamically rather than statically.

LECTURE : 3.

TYPES OF STATIC DATA STRUCTURE

the static data structure is a method of storing data where the amount of data stored and the memory used to hold it are fixed. an example of static data structure is an array.

ARRAYS.

- ✓ the term array is generally used in computer programming to mean a contiguous chunk of memory locations where each memory location holds one fixed length data item.
- ✓ the array also can mean logical data type composed of a typically homogeneous collection of data items, each identified by an **Index Number**.
- ✓ An **array** is a list or table of data with a variable name identifying such a list or table. each item in the table is called an **Element**.
- ✓ an array is a container that can hold a fixed number of items. these items should be of the same data type. for example integers only or float only or string only.
- ✓ In some programming languages, the size of an array must be established once and for all during the program designing time and can not change during the execution. such arrays are called *Static Arrays*.
- ✓ Static Arrays are the fundamental array type in most older procedural languages, such as Fortran, Basic and C and many newer object-oriented languages as well, such as Java.
- ✓ the following are the essential terms to understand the concept of an array;-
 - a. Element - refers to every item stored in an array.
 - b. Index - used to identify the location of an element.

CLASSIFICATION OF ARRAYS IN ALGORITHMS AND DATA STRUCTURE.

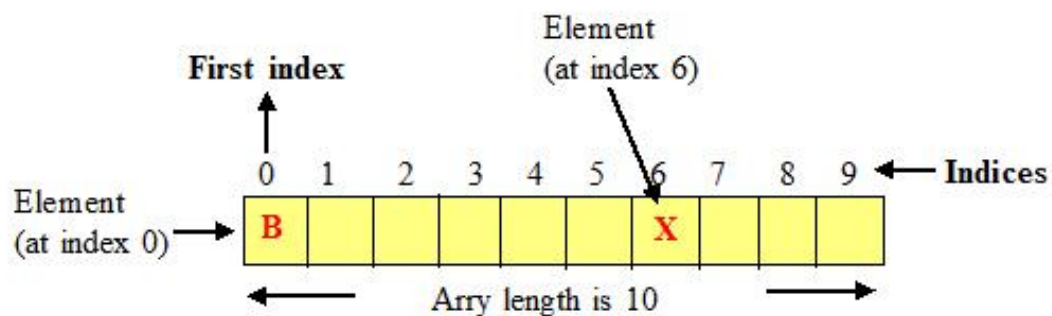
Arrays must be classified before being used in the process. they are classified into two dimensions as follows:-

- a. **One-dimensional Array**: this array has one row of elements and is stored in ascending storage location in its table.
- b. **multi-dimensional Arrays**: this kind of array includes as many indices as required because they are not bound to two indices or two dimensions. example of multi-dimensional arrays are :-
 - i. two-dimensional(2-D) arrays or matrix array.
 - ii. Three-dimensional(3-D) Arrays.

ONE-DIMENSIONAL ARRAY STRUCTURE

they can be declared in various ways in different languages. for example, from the c++ array declaration, the structure of a one-dimensional array is as follows;-

```
int array[10] = {54,63,90,34,23,44,12,47,97,123};
```



Task 1: from the above example, identify the following the concepts

- I. index
- II. Array Length.
- III. how to access an element in an array.

When declaring an array in a program, we assign initial value to each of it's elements by enclosing the values in braces {}.

```
int Num[10] = {4,6,67,34,52,61,51,90,33,62}.
```

Task 2: Add all the elements of an array using c++.

Task 3: using c++ write a program to find laragest and smallest element of an array.

Task 4: using c++ write a program to reverse the elements of the above array.

Task 5: using c++ write a program to swap the first and last element of an array above.

SOLUTIONS.

```
#include <iostream>
using namespace std;

int main() {
// Task 1: Define the array
int Num[10] = {4, 6, 67, 34, 52, 61, 51, 90, 33, 62};
```

```
// Task 2: Sum all elements of the array
int sum = 0;
for (int i = 0; i < 10; i++) {
sum += Num[i];
}
cout << "Task 2: Sum of array elements is " << sum <<
endl;
```

```
// Task 3: Find the largest and smallest elements of the
array
int largest = Num[0];
int smallest = Num[0];
```

```
for (int i = 1; i < 10; i++) {
if (Num[i] > largest) {
largest = Num[i];
}
if (Num[i] < smallest) {
smallest = Num[i];
}
}
```

```
cout << "Task 3: Largest element is " << largest << endl;
```

```
cout << " Smallest element is " << smallest << endl;
```

```
// Task 4: Reverse the elements of the array
int temp;
for (int i = 0, j = 9; i < j; i++, j--) {
    // Swap elements at indices i and j
    temp = Num[i];
    Num[i] = Num[j];
    Num[j] = temp;
}
```

```
cout << "Task 4: Reversed array elements: ";
for (int i = 0; i < 10; i++) {
    cout << Num[i] << " ";
}
cout << endl;
```

```
// Task 5: Swap the first and last elements of the array
int firstElement = Num[0];
Num[0] = Num[9];
Num[9] = firstElement;
```

```
cout << "Task 5: Array after swapping first and last
elements: ";
for (int i = 0; i < 10; i++) {
    cout << Num[i] << " ";
}
cout << endl;
```

```
return 0;
}
```

LECTURE : 4.

TWO-DIMENSIONAL ARRAY(2-D Array).

are indexed by two scripts, one for the row and one for the column. the syntax is as follows:-

(Data Type)(Name of Array) [Number of rows][Number of Columns];

ie. `int matrix[7][7]`

when you type this statement, your compiler will generate a 2-D array of a matrix that consists of 7 rows and 7 columns.

	Col1	Col2	Col3	Col4
Row1	<code>Arr[0][0]</code>	<code>Arr[0][1]</code>	<code>Arr[0][2]</code>	<code>Arr[0][3]</code>	
Row2	<code>Arr[1][0]</code>	<code>Arr[1][1]</code>	<code>Arr[1][2]</code>	<code>Arr[1][3]</code>	
Row3	<code>Arr[2][0]</code>	<code>Arr[2][1]</code>	<code>Arr[2][2]</code>	<code>Arr[2][3]</code>	
Row4	<code>Arr[3][0]</code>	<code>Arr[3][1]</code>	<code>Arr[3][2]</code>	<code>Arr[3][3]</code>	
⋮					

fig. 2-d array

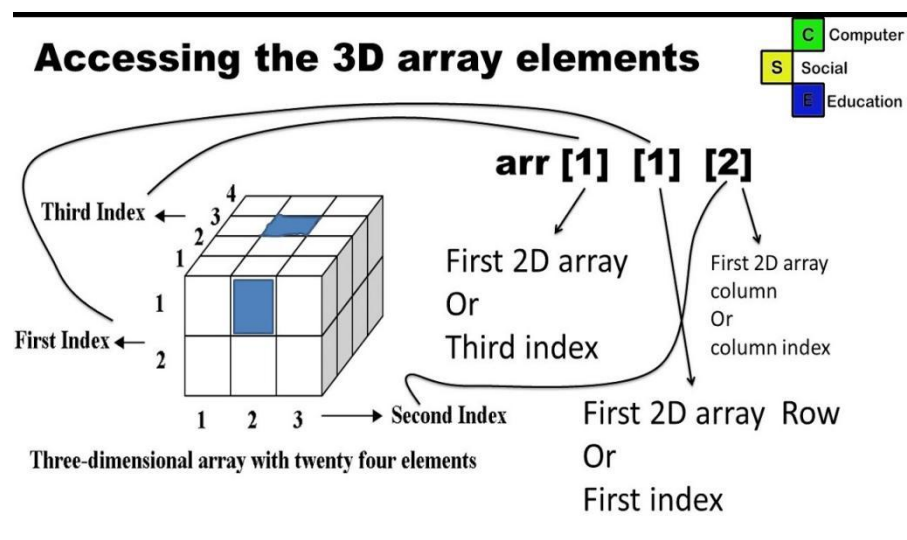


fig. 3-d arrays.

Initializing a 2-D Arrays.

```
int arr[4][2] = {  
    {1234, 56},  
    {1212, 33},  
    {1434, 80},  
    {1312, 78}  
};
```

or

```
int arr[4][2] = {1234, 56, 1212, 33, 1434, 80, 1312, 78};
```

Printing a 2-D Array

```
#include<iostream>  
using namespace std;  
int main( )  
{  
    int arr[4][2] = {  
        { 10, 11 },  
        { 20, 21 },  
        { 30, 31 },  
        { 40, 41 }  
    } ;  
    int i,j;  
    cout<<"Printing a 2D Array:\n";  
    for(i=0;i<4;i++)  
    {  
        for(j=0;j<2;j++)  
        {  
            cout<<"\t"<<arr[i][j];  
        }  
        cout<<endl;  
    }  
}
```

accessing location of two-Dimensional Array Elements.

To store values in a c++ 2-d array, the programmer has to specify the number of rows and columns of a matrix.

to access each matrix location to store the values, the user must also provide the exact number of row and columns.

forexample;-

```
int matrix[3][3]
```

```
matrix[0][0] = 67
```

```
matrix[0][1] = 56
```

```
matrix[0][2] = 34
```

```
matrix[1][0] = 7
```

```
matrix[1][1] = 16
```

```
matrix[1][2] = 64
```

```
matrix[3][0] = 7
```

```
matrix[3][1] = 16
```

```
matrix[3][2] = 64
```

Entering the data in two-Dimensional Arrays.

A nested loop is used to enter data into 2-D arrays. it depend up on the programmer which loop they want to use, which can be a **while loop** or a **for loop**.

the outer loop act as the number of rows of a matrix and the inner loop act as the number of columns of a matrix.

Task1. write a c++ program to enter and display a two dimensional array.

Task 2: write a c++ program to perform a matrix addition in 2-d array.

Task 2: Write a user defined function named Upper-half() which takes a two dimensional array A, with size N rows and N columns as argument and prints the upper half of the array.

e.g.,

2 3 1 5 0		2 3 1 5 0
7 1 5 3 1		1 5 3 1
2 5 7 8 1	Output will be:	1 7 8
0 1 5 0 1		0 1
3 4 9 1 5		5

APPLICATIONS OF ARRAYS

Arrays are useful because they allow many values to be entered in a single data structure while quickly accessing each value. This is possible as all values in the array are of the same data type, therefore need the same amount of memory store.

Also, the elements are kept in a contiguous memory location.

ARRAY OPERATIONS.

operations with a one-dimensional array are as follows;-

a. Deletion - involves deleting specified elements from an array.

```
void DeleteElement(int arr[], int& size, int position) {
    if (position < 0 || position >= size) {
        cout << "Invalid position for deletion." << endl;
        return;
    }
}
```

```
// Shift elements
for (int i = position; i < size - 1; ++i) {
    arr[i] = arr[i + 1];
}
```

```
// Decrease the size
```

```
--size;  
}
```

b. Insertion - used to insert an element at a specified position in an array.

ie. pseudocode

```
void InsertElement(int arr[], int& size, int position,  
int value) {  
    if (position < 0 || position > size) {  
        cout << "Invalid position for insertion." << endl;  
        return;  
    }  
  
    // Shift elements  
    for (int i = size; i > position; --i) {  
        arr[i] = arr[i - 1];  
    }  
}
```

```
// add new element  
arr[position] = value;
```

```
++size;  
}
```

c. Searching - an array element can be searched. the process of seeking specific elements in an array is called searching.

```
int SearchElement(const int arr[], int size, int target)  
{  
    for (int i = 0; i < size; ++i) {  
        if (arr[i] == target) {  
            return i; // if found  
        }  
    }  
}
```

```
}  
  
return -1; // if not found  
}
```

d. Merging - the elements of two arrays are merged into a single one.

```
void MergeArrays(const int arr1[], int size1, const int  
arr2[], int size2, int result[]) {  
    // Copy array 1  
    for (int i = 0; i < size1; ++i) {  
        result[i] = arr1[i];  
    }  
  
    // Copy array 2  
    for (int i = 0; i < size2; ++i) {  
        result[size1 + i] = arr2[i];  
    }  
}
```

e. Sorting - arranging elements in a particular order, either in ascending or descending order.

LIMITATIONS OF USING ARRAYS

several limitations must be carefully considered during the implementation of arrays, these includes:-

- a. Array are of fixed size.
- b. inserting and deleting elements can be problematic because of the shifting of elements from their positions within a table row or column.
- c. data elements are stored in a contiguous memory location that may not always be available .To solve this implication of using arrays, you can use linked lists arrays.

ADVANTAGES OF ARRAYS.

the following are the advantages of arrays:-

- a. The array is simple and easy to use.

b. it is also, faster to access the elements.

DISADVANTAGES OF ARRAYS.

The Arrays contain the following disadvantages:-

- a. The cluster size should be known before utilising it.
- b. all the components in the cluster must be adorned and put away in the memory.
- c. embedding any element in the set needs moving of every element of its originals.
- d. expanding the size of the cluster is a period taking cycle.it is not easy to grow the size of the exhibit at runtime.

LECTURE: 5

POINTERS.

defn:

- ◆ a pointer is a variable whose value is the address of another variable, that is, the direct address of the memory location.
- ◆ it is a user-defined data type that creates variables for holding the memory address of other variables.
- ◆ it is denoted by the "*" operator.
- ◆ it represents storage space in memory(RAM) partitioned into a small portion called cells which are used to store values. each cell has a unique address and occupies one byte in memory, whose address is always unassigned integers.
- ◆ if one variable carries the address of another variable, the first variable is said to be the pointer of another.

thus, a **pointer** is the variable whose value is also an address where each variable has two attributes:(address and value).

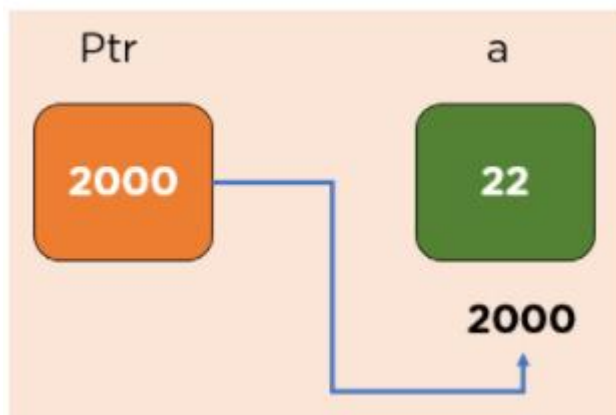


fig above. the pointer and it's variable

Syntax;-

type *ptr_name;

where type is any data type of the pointer.
and ptr_name is a pointer's name.

declaring pointers in c++

the pointers can be declared as follows:-

Datatype *variable_name;

- the **datatype** is the base type of the pointer, which has to be valid c++ datatype in case of c++.
- the **variable_name** should be the name of the pointer variable.
- an asterisk (*) is just an asterisk used to perform multiplication operations. it marks the variable as a pointer.

the c++ uses the '&' (reference) operator to return a variable's address.

task 1: how to apply pointer in c++.

```
#include<iostream>
using namespace std;

int main()
{
    int y = 10;
    int *b;
    b = &y;

    cout<<"the value of y = "<<y<<endl;
    cout<<"the address of y = "<<b<<endl;

    cout<<"the pointer of y = "<<*b<<endl;

    return 0;
}
```

the output will be

the value of y = 10
the address of y = 0x7ff7b1b2e588
the pointer of y = 10

Note: the datatype of pointer and variable should be the same.

Advantages of pointers

the following are the advantages of pointers:-

a. it can return multiple values from a function.

->it allow a function to return multiple values by passing memory address as argument to the function.

b. it reduces the code and improves performance.

->by manipulating the memory address directly, you can often achieve better performance compared to working with values directly.

c. it retrieves strings and trees.

->pointers are commonly used in handling strings, where they can be used to efficiently manipulate and traverse characters in memory.

->also are also essential for working with dynamic data structure like trees, where nodes are connected using pointers.

d. used with arrays, structures and functions.

->they enable efficient access and manipulation of elements in arrays, facilitate dynamic memory allocation for structures, and are crucial in passing arrays and structures to functions without making copies, thus saving memory and improving performance.

e. it allows you to access any memory location in computer's memory.

f. it can allocate and de-allocates memory dynamically.

->pointers are extensively used for dynamic memory allocation. functions like 'malloc()', 'calloc()', and 'realloc()' allow you to allocate memory dynamically at runtime and the 'free()' is used to de-allocate memory. this dynamic memory management is particularly when the size of data is not known at compile time or when memory need to be released when it is no longer needed.

Disadvantages of pointers

the following are the disadvantages of pointers:-

a. if it is wrongly referenced, it will affect the entire program.

->if a pointer is wrongly dereferenced, meaning it is used to access or modify the value at an incorrect memory location.it can lead to unpredictable behavior and potentially affect the entire program.

b. it may lead to memory leak if the dynamically allocated memory is not freed.

->memory leaks occurs when memory that is no longer needed is not released, causing the program's memory usage to grow overtime.

c. an uninitialised pointer leads to a segmentation fault.

->using an uninitialized pointers, or a pointer that has not been assigned a valid memory address, can result into segmentation fault.

->a segmentation fault occur when a program tries to access a restricted area of memory, often leading to a program crash.

LECTURE: 6

APPLICATIONS OF POINTERS.

a pointer is used in various ways as follows:-

- a. it is used to access array elements.
 - >an array of components can be accessed using pointers.

Task 1: access an array elements using C++ pointers.

```
int main()
{
int a[] = {4,6,3,10,34,23};

cout<<*(a+1);
return 0;
}
```


b. it is used for dynamic memory allocation.

Task 2: using C++ allocate dynamic memory.

```
int main()
{
    int i, *ptr;

    ptr = (int*) malloc(3*sizeof(int));
    if(ptr == NULL)
    {
        cout<<"Error! memory not allocated.";
        exit(0);
    }
    *(ptr + 0) = 10;
    *(ptr + 1) = 20;
    *(ptr + 2) = 30;
    cout<<"elements are: ";
    for(i=0; i<3; i++)
    {
        cout<<*(ptr+i)<<endl;
    }
    free(ptr);
    return 0;
}
```

NOTE:

A **void pointer** is a general-purpose pointer that can hold the address of any data type, but it is not associated with any data type.

Syntax of void pointer

void *ptr;

c. the pointers are used to pass arguments by reference in functions to increase efficiency.

Task 3: using C++ pass arguments by reference in functions.

```
void swap(int* x, int* y)
{
    int t=*x;
    *x=*y;
    *y=t;
}

int main()
{
    int r = 3, s=9;
    cout<<"the values of r and s before swap:"<<endl;
    cout<<"r: "<<r<<endl<<"s: "<<s<<endl;
    swap(&r, &s);
    cout<<"values after swapping"<<endl;
    cout<<"r: "<<r<<endl<<"s: "<<s<<endl;
    return 0;
}
```

d. the pointer can be used to implement data structures like linked lists and trees.

POINTER OPERATIONS.

In C++, the pointer operator are of two kinds:-

a. **Reference Operator(&):** the reference operator(&) returns the variable's address.

b. **Dereference Operator**(*): the dereference operator(*) help us to get the value that has been stored in a memory address.

forexample:-if you have a variable given the name num1, stored in the address 0x236 and storing the value 27, then

- a. the reference operator (&) will return 0x236.
- b. the dereference operator (*) will return 27.

Task 4: using C++ demonstrate pointer operators (reference and dereference operators).

```
int main()
{
    int y = 10;
    int *b;
    b = &y;
    cout<<"the value of y = "<<y<<endl;
    cout<<"the address of y = "<<b<<endl;
    cout<<"the pointer of y = "<<*b<<endl;
    return 0;
}
```

POINTERS IN ARRAYS

arrays and pointers work based on related ideas. there are different things to consider when working with arrays having pointers.

The array name denotes the base address of the array. this means that, when assigning an array's address to a pointer, do not use an ampersand(&).

forexample.

p=arr; this is correct, because arr represent the arrays' address but

p=&arr; is incorrect.

you can implicitly convert an array into a pointer, forexample:-

```
int arr[20];  
int *y;
```

this is the valid operation:

```
y=arr;
```

after the declaration, y and arr will be equivalent and share properties.

However, a different address can be assigned to **y**, but you can not assign anything to **arr**.

Task 5: using c++, traverse an array using pointers.

```
int main()  
{  
    int *y;  
    int arr[] = {23,56,12,78,34,22,10};  
  
    y = arr;  
    for(int i = 0; i<7; i++)  
    {  
        cout<<*y<<endl;  
        y++;  
    }  
    return 0;  
}
```

NOTE:

When an array name is used by itself, the array's address is returned. we can assign this address to a pointer as illustrated below:-

```
int arr[4] = {1,5,4,9};
```

```
int *mypointer;
```

the variable **mypointer** is the pointer to the first element of the array and not the array itself.

LECTURE: 7

RECORDS.

- Records are composite datatypes formed by several related items of different datatypes.
- This enables a programmer to refer to these items using the same identifier, enabling a structured approach to use related items.
- A record will contain a fixed number of items.
- *Forexample*, a record for a book may include the *title*, *author*, *publisher*, *number of pages*, and whether it is *literature* or *fiction*.
- Records store a collection of related data items, where all items have different data types.
- *forexample*, you might set up a record called book, which stores the book's title, author name, and ISBN.

Title and author are *text*, where as PublicationDate is set as *date* data type.
you can write it as follows:-

Book = Record

Title, Author As Text * 50

ISBN As Text * 13

PublicationDate As Date

generally,

A record is a data structure that consists of a fixed number of variables called fields.

- Every fields has an *identifier* (field name) and a *data type*. each field in a record can have a different data type.
- This is very common in Spreadsheets and MS Access, and other databases.
- some languages provide a built-in structure type that can be used to define a record.
- forexample, assume that you want to organise the individual student records such as registration number, name, sex, date of birth, average score, and grade into a single data structure.

student's_regNo: integer

student's_name: string

student's_sex: string

student's_average_score: real

student's_grade: char

structure of records

Application of Records

the following are the application of records:

- a. it is used to store data or information in the database such as Microsoft Access.
- b. it is used to group similar data.

Record Operations.

the record operations include creating, accessing, and updating the field in the record.

creating a student's record

the keyword 'record' is used to create records specified with the record name and fields. its syntax is as follows:-

record(recordname, {field1,field2,field3.....fieldn})

the syntax to *insert values into the record using c++* is as follows:-

#recordname {fieldname1 = value 1, fieldname2 = value2, fieldname3 = value 3, fieldname4 = value 4..... fieldnamex = value x}

TASK 1: write a c++ program to create a record.

```
class student{
public:
string student_name;
int student_id;
};

int main()
{
student S;
S.student_name = "fumbuka";
S.student_id = 4747;
```

```
return 0;  
}
```

TASK 2: write a c++ program to access record values.

```
class student{  
public:  
string student_name;  
int student_id;  
};  
  
int main()  
{  
student S;  
S.student_name = "kharid aucho";  
S.student_id = 7880;
```

```
cout<<S.student_id<<"\n"<<S.student_name;
```

```
return 0;  
}
```

LECTURE 8: USER-DEFINED DATA TYPES

The user-defined data types allow a programmer to develop his/her data types and define what values program can take.

ie. class, structure, union, and enumeration.

These data types hold more complexity than pre-defined data types, but they can assist a programmer in reducing errors.

In computer science, the data types defined by the user are called derived data types or user-defined data types.

User defined data types are also known as Composite data types. they are called composite because they are derived from more than one build-in data type used to store complex data. these complex format data might contain tabular data, graphical data, and databases.

In C++ we have the following user-defined datatypes ,
class, union, structure, enumeration, and typedef.

a. **Class.**

This is the user-defined datatype that holds data members and functions whose access can be specified as private, public, or protected.

it uses the "**class**" keyword for defining the data structure.

b. **Structure.**

a structured data type groups data items of different types into a single type.

for example, a structure can an address, which contains information such as block number, plot number, building name, street, city, country and a pin code.

the keyword "**struct**" is used to define this.

c. **Union.**

the union is a type of data structure where all the members of that union share the same memory location.

if any changes are made in the union, they will be also be visible to others.

the "**union**" keyword is used to define this.

d. **Enumeration.**

it help to assign names to integer constants in the program. The keyword '**enum**' is used.

it is used to increase the readability of the code.

e. **Typedef.**

this defines a new name for an existing datatype. it does not create a new data class.

it makes code readability easy and gives more clarity to the user.

APPLICATION OF USER-DEFINED DATA STRUCTURE.

a. user-defined data types are the building blocks for other data types that model the behaviour of data.

b. they are used to implement mathematical vectors and matrices.

c. they are used to model sets or collections in computer programming.

d. it creates other data structures such as , lists, stacks and queues.

THE STRUCTURE.

A structure is a collection of multiple variables of different data types grouped under a single name for convenient handling.

the members of structures can be ordinary variables, pointers, arrays or even another structure.

declaring structures

the structures are created by using the **struct** keyword.

```
struct member
{
    member a;
    member b;
    member c;
    .....;
    member n;
};
```

forexample, you can create a structure to store the student's marks

```
struct student //student structure name
{
// structure member
int stdid;
char stdname [40];
float marks, attendance;
};
```

TASK: The c++ program to demonstrate structures in c++.

```
#include<iostream>
using namespace std;

struct point{
int x,y;
};
int main()
{
// create an array of structure
struct point arr[20];
// access array members
arr[0].x = 30;
arr[1].x = 20;
arr[1].y = 40;
arr[0].y = 50;
cout<<arr[0].x<<"<<arr[0].y<<"<<arr[0].y;
return 0;
}
```

UNION

the union is a user-defined datatype in which all members share the same memory place.

forexample, in the following c++ program , y and z share the same location. if we change z, you see the changes reflected in y.

```
#include<iostream>
using namespace std;

union myunion
{
    int x,y;
};

int main()
{
    union myunion p;
    p.x = 10;
    cout<<"the value of x = "<<p.x<<"", "<<"the value of y = "
    "<<p.y<<endl;
    p.y = 80;
    cout<<"the value of x = "<<p.x<<"", "<<"the value of y = "
    "<<p.y<<endl;
    return 0;
}
```

ENUMERATION

in c++, an enumeration(or enum) is primarily used to give integral constant names, making the program easier to comprehend and maintain.

->An enumeration is a user-defined data type that consists of integral constants.

in enumeration, if you do not provide the integral values explicitly to the strings, then, in that case, the strings automatically start assigning the integral values start from value 0, the same as the case of 0-indexing.

Points to remember for c++ Enum:-

a. enum improves type safety.

->it reduce the risk of using the incorrect values. this helps improve type safety because the compiler can catch type mismatches during compilation.

ie. enum color {green, red, brown};

b. enum can be easily used in switch.

->it provide a cleaner and more readable way to handle multiple cases compared to using integers constants or strings.

```
enum Day { MON, TUE, WED, THU, FRI, SAT, SUN };

Day today = WED;
switch (today) {
case MON:
case TUE:
case WED:
case THU:
case FRI:
cout << "Weekday" << endl;
break;
case SAT:
case SUN:
cout << "Weekend" << endl;
break;
}
```

c. enum can be traversed.

ie. it can iterate through the days of the week using loops

```
enum Weekday { MON, TUE, WED, THU, FRI };

for (int day = MON; day <= FRI; ++day) {
cout << "Day: " << day << endl;
}
```

d. enum can have fields, constructors and methods.

->in c++, enums can have more additional features beyond just defining constants. Enums can have fields(members), constructors, and even methods, providing more flexibility and functionality.

e. enum may implement many interfaces but can not extend any class because it internally extends the Enum class.

->Enums in c++ does not support class inheritance, while an enum can implement multiple interfaces, it can not extend any class. internally, enums in c++ are often implemented as classes that extends the 'Enum' class, providing a certain level of type safety and functionality.

example:-

```
enum season {spring, summer, autumn, winter};
```

note: by default these names are associated with consecutive integer values starting from 0(spring) to 3(winter).

```
enum result {pass = 60, fail = 16};
```

here , we have given the integer value 60 to be pass and 16 as fail; therefore, if we write

```
enum result res;
```

```
res=pass;
```

then, the value of res would automatically be 60.

TYPDEF

the keyword *typedef* in c++ allows you to define new datatype names explicitly.

the use of typedef does not create a new data class but establishes a name for an existing type. this can improve a program's portability.

note:

program's portability means the ability of the program to be used across different types of machines; I.e. mini, mainframe and micro without requiring significant changes to the code. because only the typedef statements would need to be changed.

By allowing descriptive names for standard data types, typedef can also help with self-documenting code.

syntax;

Typedef type name;

TASK: c++ program to demonstrate typedef user-defined datatype

```
#include<iostream>
using namespace std;

typedef char fumbu;
int main()
{
    fumbu c1, c2;
    c1 = 'p';
    cout<<" "<<c1;
    return 0;
}
```

BENEFITS OF USER-DEFINED DATA TYPES

- a. user-defined data types store data elements of either the same or different types. this gives more flexibility for the programmer to store different data types in a single variable as per their needs and requirements.
- b. Reusability: once defined, these data types can be reused within many definitions, saving coding time.
- c. Flexibility: they allow us to create a data structure per our requirements and needs.
- d. Encapsulation: in java/python, the variables and data values stored in user-defined data types are hidden from other classes as per their accessibility declaration, ie. public, private, and protected. the data values remain hidden and safe.

PROJECT GROUP WORK.

Develop a simple C++ program that includes the struct, union, and enum concepts. the program should address any problem in your school or community.

LECTURE 9:

DYNAMIC DATA STRUCTURE

Meaning of Dynamic Data Structures

In dynamic data structures, the size of the data structure can be modified after initialization. It means we can add/delete elements from the data structure, thus modifying its size at runtime. Dynamic data structures are memory efficient and especially useful when we don't know the data size beforehand.

defn:-

A dynamic data structure is a data structure whose size and shape can change during runtime to accommodate different data requirements.

Examples of dynamic data structures include linked lists, trees, queues, and stacks.

Features of Dynamic Data Structures

Here are some features of dynamic data structures:

- The size of a dynamic data structure is not fixed and can change during runtime.
- Memory allocation for dynamic data structures is done during runtime using techniques such as heap memory allocation or pointer-based data structures.
- Dynamic data structures can grow or shrink as needed and are best suited for applications that have varying sizes or a changing number of elements.
- Insertion and deletion operations are generally faster in dynamic data structures since elements can be added or removed without the need to shift other elements.
- Accessing elements in a dynamic data structure may be slower than in a static data structure since memory may be spread out in different locations.

Advantages and Disadvantages of Dynamic Data Structures?

Let's look into the advantages and disadvantages of Dynamic Data Structures:

Advantages

Here are some advantages of dynamic data structures:

- They can change in size and shape during runtime, making them flexible and adaptable to different applications.
- Memory usage is optimized, as it can grow or shrink based on actual data requirements.
- Insertion and deletion of elements are faster and more efficient than static data structures.
- They are well-suited for large and complex data sets.

Disadvantages

Here are some disadvantages of dynamic data structures:

- Memory allocation and deallocation can lead to performance issues, such as fragmentation, overhead, or memory leaks.
- They may require more memory than static data structures for bookkeeping and pointer storage.
- Accessing elements can be slower than static data structures due to the need for pointer traversal or indirection.
- They can be more challenging to implement and debug than static data structures.

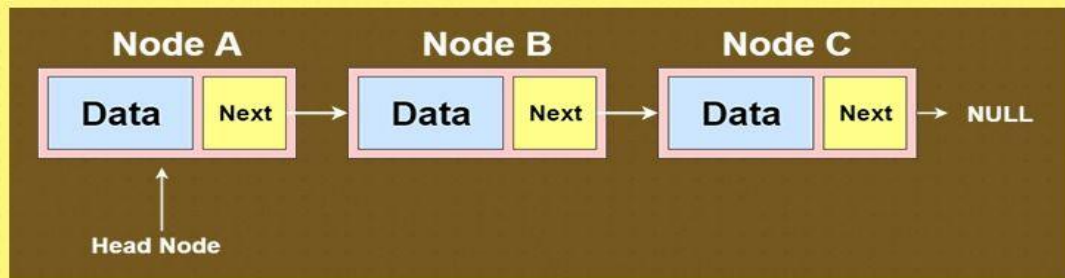
LINKED LIST.

A Linked Lists is a linear **Data Structure** that consists of a group of nodes. Unlike an array, it has elements that are stored in random memory locations. Each node contains two fields:

- **data** stored at that particular address.
- The **pointer** contains the address of the next node.

The last node of the Linked list contains a pointer to **null** to represent the termination of the linked list. Generally, we call the first node as the **head** node and the last node as the **Tail** node in Linked List.

LINKED LIST DATA STRUCTURE



Why Linked List Over Array?

The array contains the following limitations:

- The size of an array is fixed. We must know the size of the array at the time of its creation, hence it is impossible to change its size at runtime.
- Inserting a new element in an array of elements is expensive because we need to shift elements to create room for new elements to insert.
- Deleting an element in an array is also expensive as it also takes the shifting of elements in the array.

Advantages of Linked List

The advantages of Linked List are:-

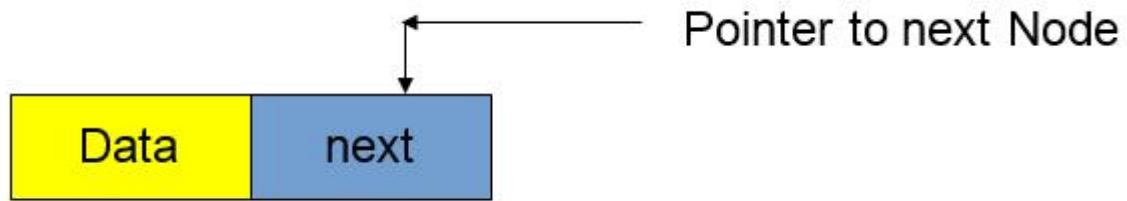
- Insertion and deletion operations can be implemented very easily and these are not costly as they do not require shifting of elements.
- They are dynamic in nature. Hence, we can change their size whenever required.
- Stacks and queues can be implemented very easily using Linked Lists.

Disadvantages of Linked List

The disadvantages of Linked List are:-

- Random access of an element is not possible in Linked Lists, we need to traverse Linked List from starting to search an element into it.
- It is relatively slow to process in comparison to an Array.
- Since node of a Linked List contains both data and pointer to the next node, hence extra memory is required to store the pointer of each node.

Types of Linked List

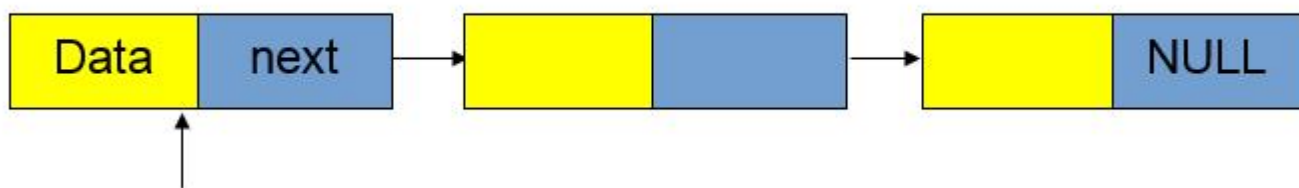


There are three types of Linked Lists:

- Singly Linked List
- Circular Linked List
- Doubly Linked List

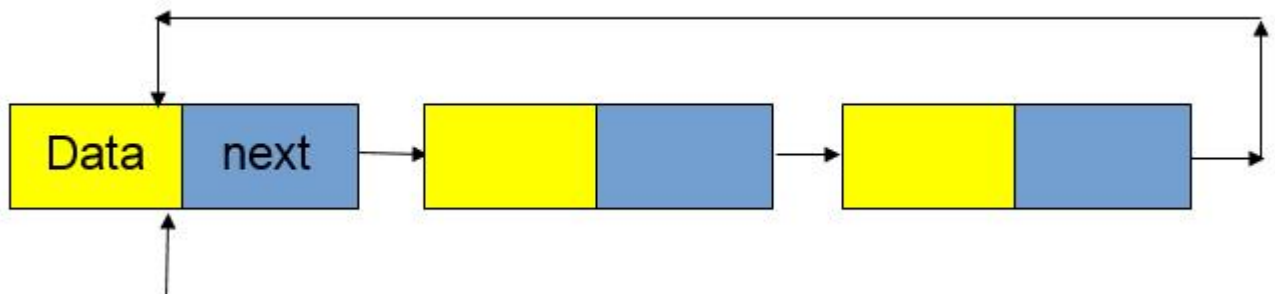
Singly Linked List

A Singly Linked List contains a node that has both the data part and pointer to the next node. The last node of the Singly Linked List has a pointer to null to represent the end of the Linked List. Traversal to previous nodes is not possible in singly Linked List i.e We can not traverse in a backward direction.



Circular Linked List

Circular Linked List is similar to singly Linked List but the last node of singly Linked List has a pointer to node which points to the first node (head node) of Linked List.



Doubly Linked List

Doubly Linked List contains a node that has three entries: (1) data part, (2) pointer to the next node, and (3) pointer to the previous node. We can traverse in both forward and backward directions in doubly Linked Lists.

Operations in a linked list

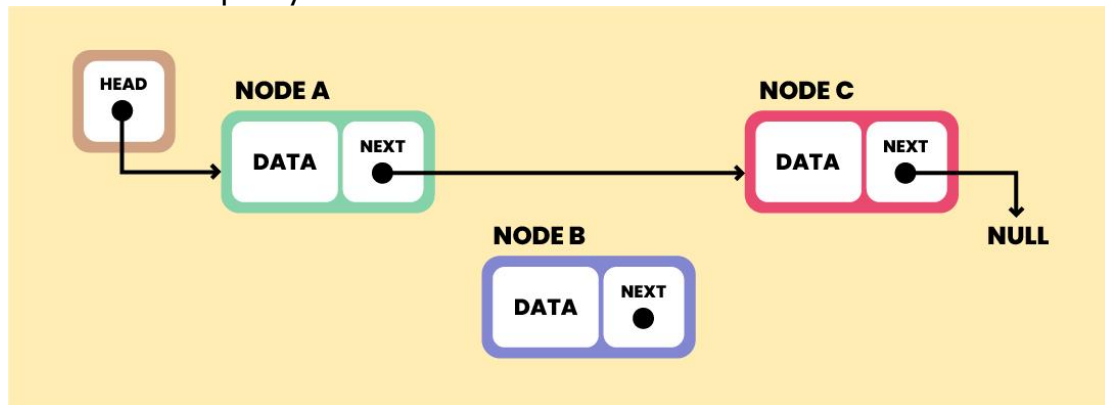
Following are the various operations to perform the required action on a linked list:

- Traversal - To access each element of the linked list.
- Insertion - To add/insert a new node to the list.
- Deletion - To remove an existing node from the list.
- Search - To find a node in the list.
- Sort - To sort the nodes.

Now, let's look at each of these operations separately-

Insertion

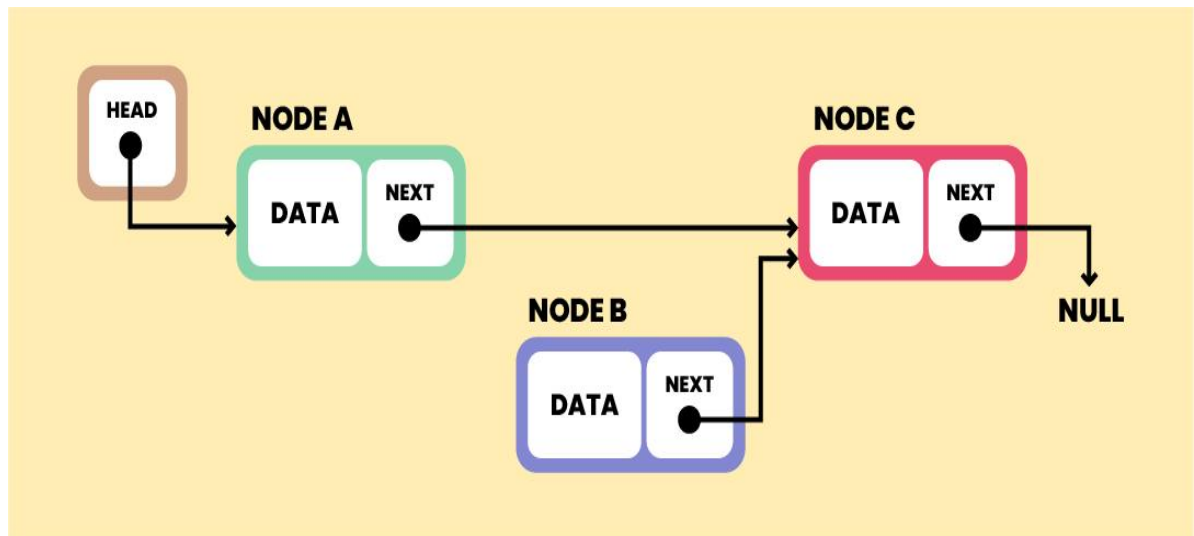
It's more than a one-step activity. First, we create a node with the same structure and specify the location where we'll insert it.



Suppose, we have to insert Node B between the two nodes A and C.

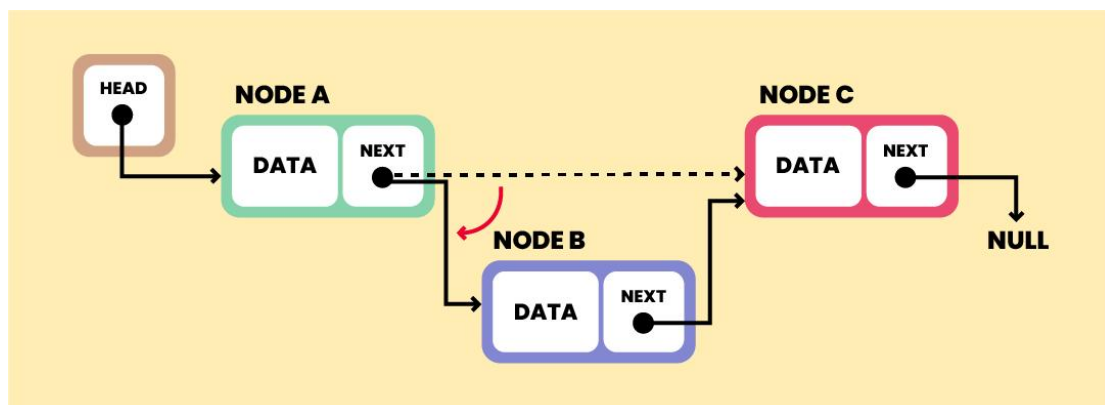
The new Node B will point to Node C.

NodeB.next - Node C

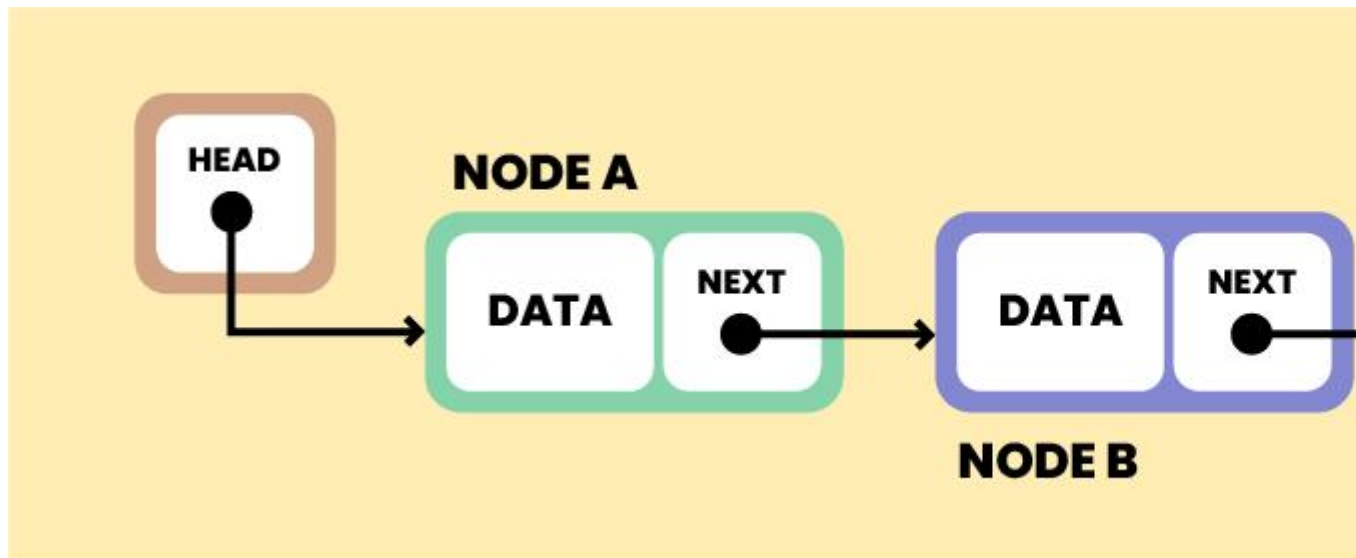


Now, Node A should point to Node B

NodeA.next — NodeB



Doing this will place the new node in the middle of the two nodes; as we wanted.



If we have to insert a node at the beginning of the list; the new node should point to the Head (First).

Similarly, if we have to insert a node at the end; the last node should be made to point to the new node, and the new node should be pointing to 'Null'.

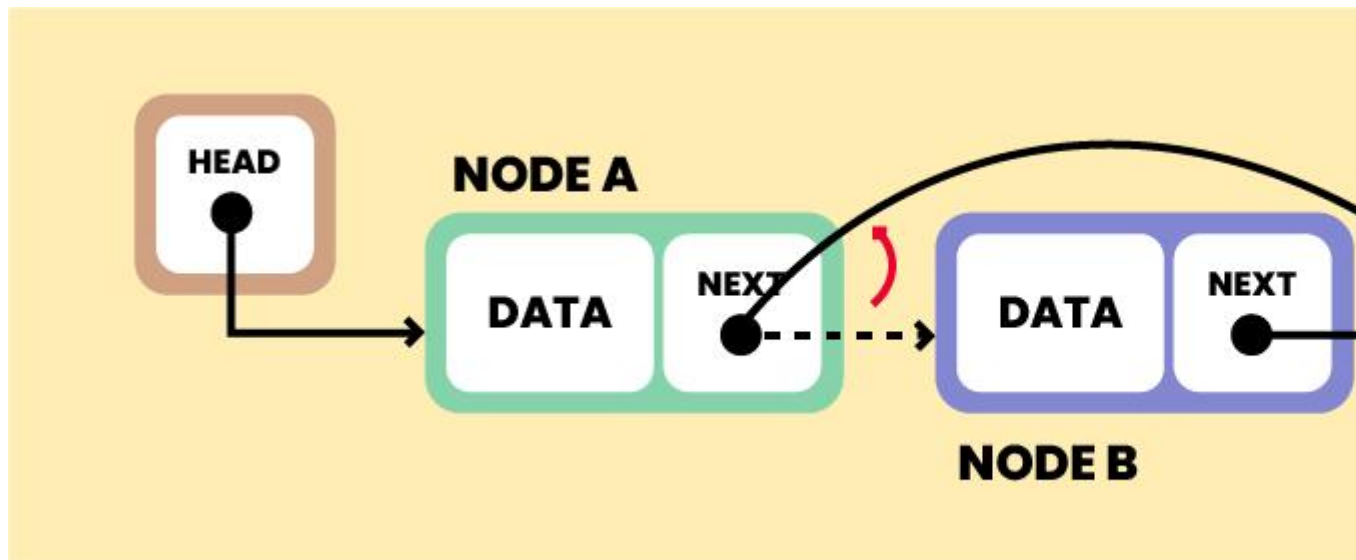
Deletion

Just like insertion, deletion is also a multi-step process.

First, we track the node to be removed, by using searching algorithms.

Once, it's located, we'll change the reference link of the previous node to the node that is next to our target node.

In this case, Node A will now point to Node C.

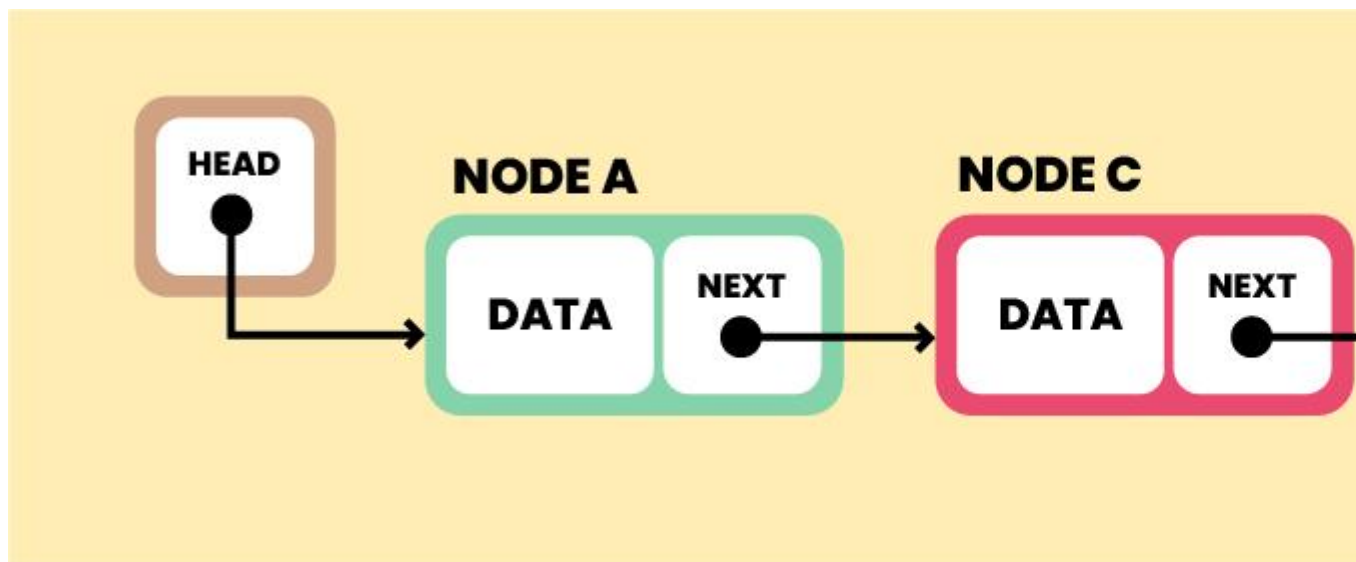


NodeA.next - NodeC

Once, the link pointing to the target node is removed, we'll also remove the link from Node B to Node C.

Now, Node B should point to 'Null'.

Node B.next — Null



Now we can either keep it in memory or deallocate memory and eliminate Node B.

To delete a node from the beginning, we'll simply direct the 'head' to the second node in the list.

In case of deleting the last node, the second last element will be directed to point to Null.

Search

We search for a node on a linked list using a loop. Suppose, we have to find Node X on a list.

- First, we'll assign the 'Head' as the 'Current' node.
- Then, we'll use a loop until the 'Current' node is 'Null' since the last element points to 'Null'.
- In each repetition, we'll check if the key of the node is equal to 'Node X'. If the key matches this item, return true, else return false.

Sort

To sort elements in ascending order, we'll use [Bubble Sort](#). This is how-

- Assign 'Head' as the 'Current' node and create another node index for later use.
- If 'Head' is null, return
- Otherwise, we'll run a loop until it becomes null.
- In each iteration, we'll store the next node of 'Current' in the index.
- We'll then check if the data of the current node is greater than the next node. In case, it is greater, we'll swap 'Current' and 'Index'. We'll follow the same process throughout.

Applications of linked lists

In programming

Implementing stacks and queues

Linked lists are used to implement stack and queue data structures. How?

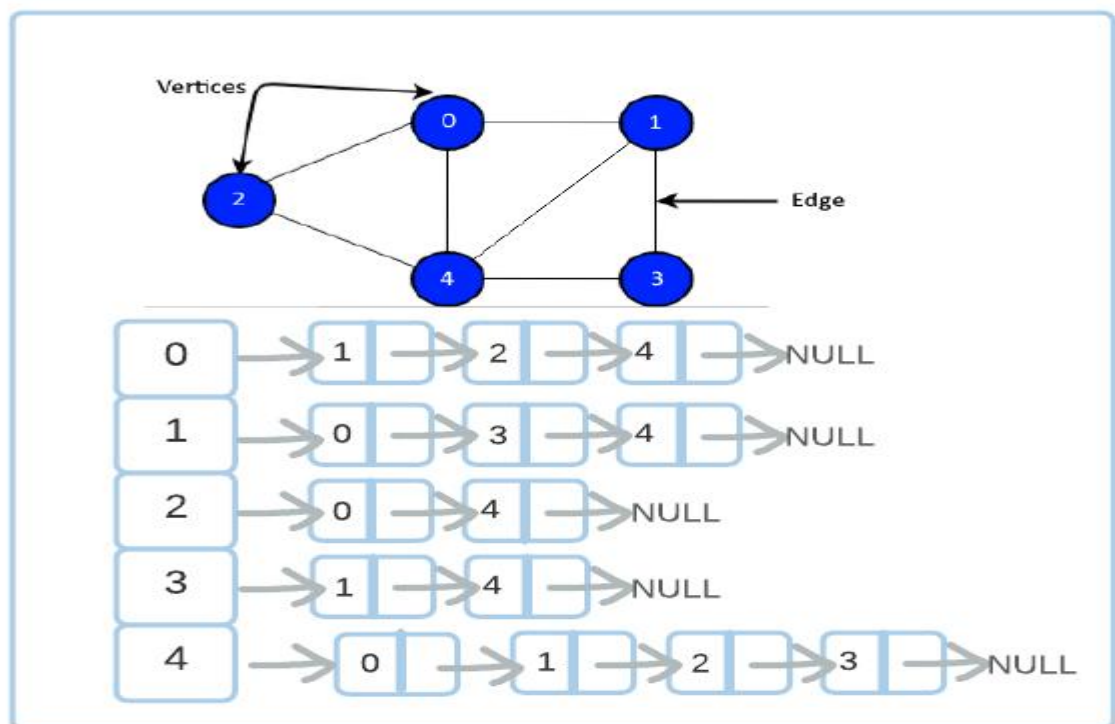
As a stack follows the Last In First Out principle, every PUSH operation will be INSERT_IN_BEGINNING and every POP operation will be DELETE_IN_BEGINNING in the linked list.

And, in the First In First Out structure of a queue, we'll maintain two pointers - the head pointing to the start of the linked list, and the tail pointing to the end of the linked list. Every PUSH operation will be INSERT_IN_END and every POP operation will be DELETE_IN_BEGINNING.

Implementing graphs

Graphs can also be implemented using the [adjacency list representation](#).

It can be depicted as an array of linked lists where lists store the adjacent vertices.



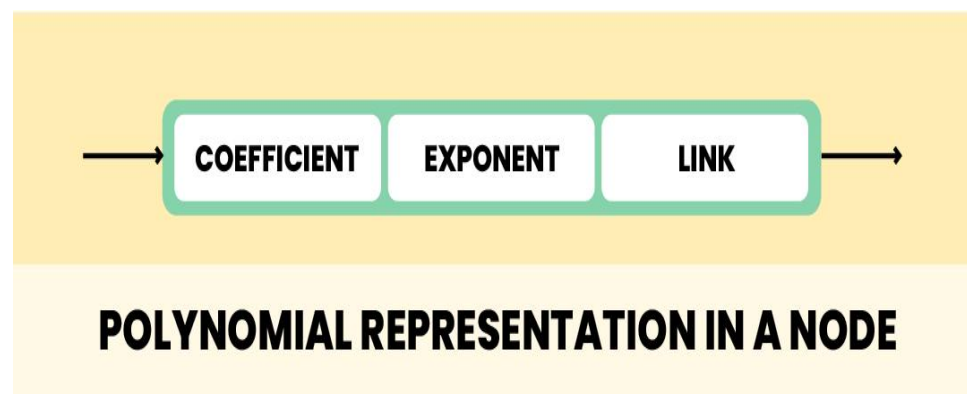
Polynomial manipulation

A polynomial, in mathematics, is a collection of different terms, comprising coefficients, variables, and exponents. They're not supported as a data type by most languages. However, linked lists can represent polynomials with ease.

How?

Each polynomial term takes up a node in the linked list. It is also made sure that the terms are arranged in the decreasing order of exponents for better efficiency.

Each node would consist of 3 parts-



Let's say we have-

$$4x^4 + 11x^3 - 2x^2 + 1$$

Here - 4, 11, 2, and 1 are the coefficients while 4, 3, 2, and 0 are the exponential values respectively.

To represent 4 terms, we'll need a linked list with 4 nodes.

Representing polynomials this way allows us to perform various mathematical operations easily.

Dynamic memory allocation

As we know that nodes of a linked list are not stored in contiguous memory locations, but instead created dynamically at runtime. So, linked lists can be helpful in [dynamic memory allocation](#) where we don't know the exact number of elements we're going to use.

In real life

Having a significant role in programming, linked lists certainly have many real-life applications. Let's look at a few of them -

In web browsers

While moving through web pages, you have both options available - to either move to the next page or the previous page. This is only possible because of a doubly linked list.

In music players

Similarly, we can switch to the next and the previous song in a music player using links between songs. We can also play songs either from the starting or the end of the playlist.

In image viewer

The next and the previous photos are linked, and we can easily access them by using the previous and next buttons.


In operating systems

The thread scheduler uses a doubly-linked list to maintain the list of all processes running at any time. It enables us to move a certain process from one queue to another with ease.

In Multiplayer games

Online multiplayer games such as the likes of PUBG and Call of Duty use a circular linked list to swap between players in a loop.

Linked list vs Array

masai.	
Array [...]	Linked List 
Elements are stored in contiguous memory locations	Elements are connected using pointers
Supports random access to elements	Only supports sequential access to elements
Insertions and deletions are tricky: elements need to be shifted	Insertions and deletions can be done efficiently without shifting
Fixed memory: static memory allocation	Dynamic memory allocation at runtime
Elements are independent of each other	Each node points to the next node or both the next and the previous node

LECTURE 10:

STACK DATASTRUCTURE

Introduction to Stack Data Structure

Stack is a linear data structure that stores elements in a specific order.

Let's take our browser history as an example. When we jump to a new page, the previous page gets stored in the browser history. We can access that page by simply clicking the back button.

This is possible because all the pages we've visited are stored in a new-to-old arrangement in the form of a stack. Hitting the back button opens up the page that is right below the newest page.

Stack is an abstract data type with a pre-defined capacity.

What is an Abstract data type?

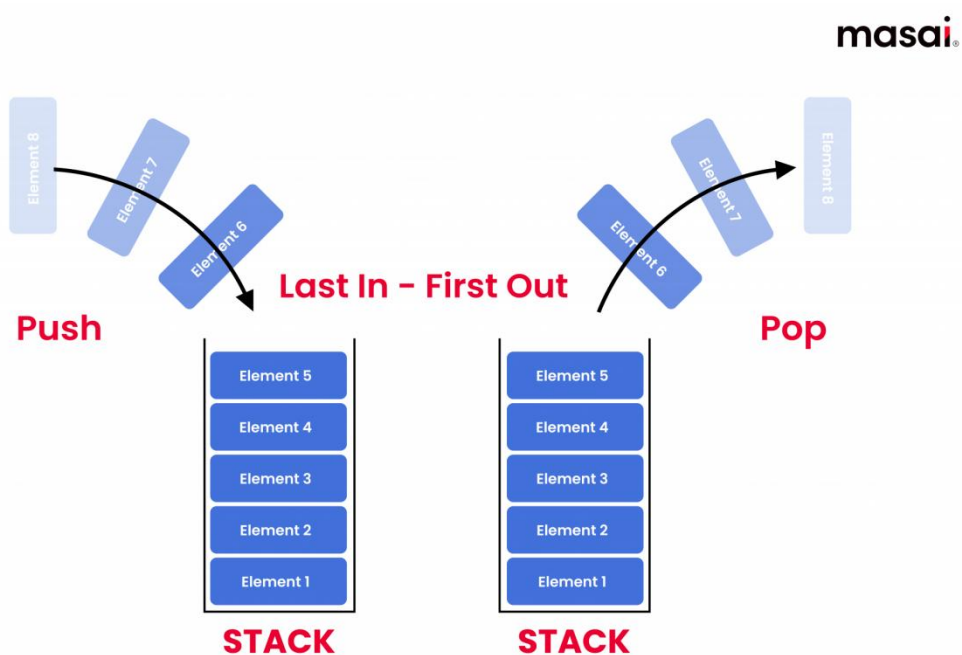
ADTs specify the data and its applications but is independent of how it's implemented. ADTs are implemented using different methods and logic in different programming languages.

Stack follows a **Last In First Out** principle for adding or removing elements.

Consider a stack of books arranged in an order.

Now, if we want to take the bottom-most book out, we will have to take out the topmost book first, then the second, then the third until we finally reach the first one at the bottom.

The book that goes in last will be the first one to come out. This is called the **LIFO** principle. In the same case, the book added first will be removed at last which refers to the **First In Last Out method** aka FILO.



A stack can only contain elements of the same data type.

Every time we add an element, it goes to the top of the stack, and that'll always be the first one to be removed from the stack.

In other words, stacks can be defined as containers in which operations like insertion and deletion can only take place at one end.

How does Stack work?

To implement a stack, we need to keep a pointer to the 'top' of the stack i.e. the last element to be added. As the operations always happen at the top of the stack, this pointer always follows the topmost element.

Operations in Stack Data Structure

push() – Push operation means adding/inserting an element into the stack.

pop() – Pop operation refers to removing an element from the stack. As we know that we can only work with one element at a time, we remove the top of the stack.

peek() – This operation allows us to view the element which is at the top. There are no changes made in the stack here.

isEmpty() – It checks if the stack is empty or not to prevent the programmers from performing operations on an empty stack. This operation returns a boolean value – 'True' if size = 0, otherwise it returns 'false'

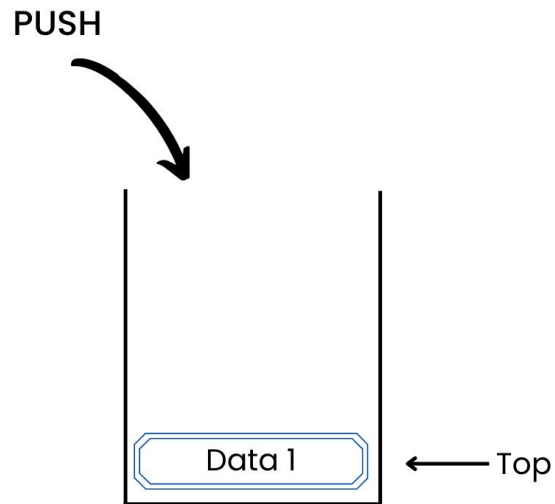
isFull() – It's the exact opposite of isEmpty operation, as it returns 'true' if the stack is full, else returns 'false'.

Now, let's go through a step by step process to see how these operations come together-

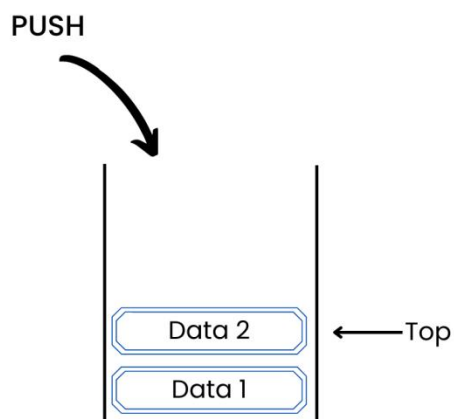
1) While initializing the stack, we set the value of the pointer 'top' = -1. This value means that the stack is empty.



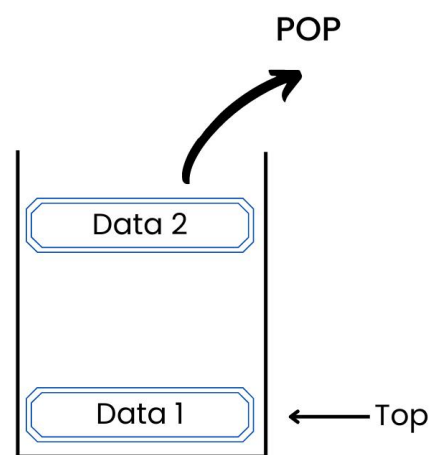
2) When we add/push an element, the value increases to 'top' = 0, and the new element is placed in the position corresponding to the 'top' pointer.



3) Similarly, adding another element increases its value to 'top' = 1' and the pointer comes up to the newest element.



4) Now, when we remove/pop an element, the pointer just moves below to show that the topmost element is no longer a part of the stack, and it can be deleted or replaced.



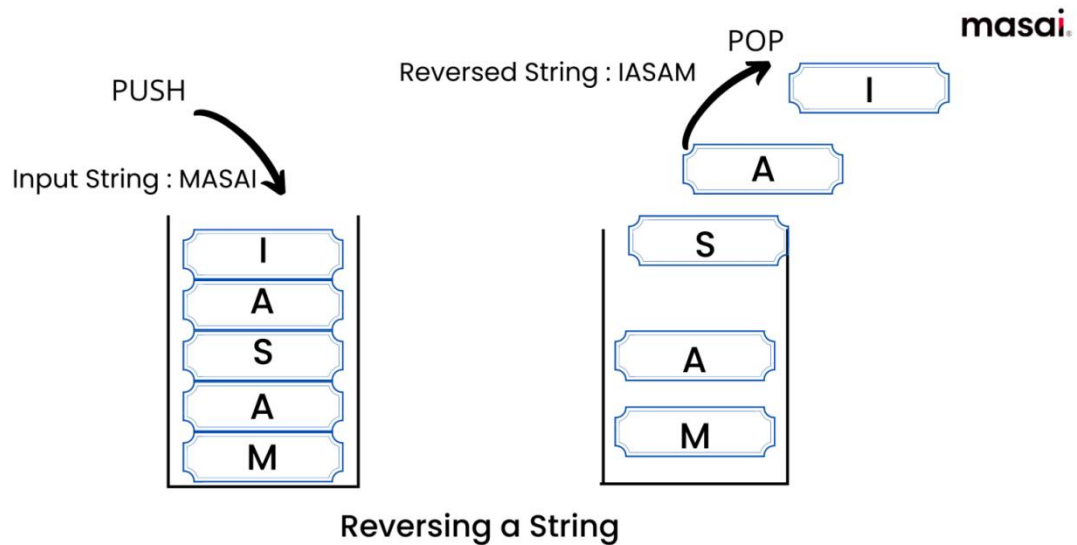
- We check if the stack is already full before pushing.
- We check if the stack is empty before implementing the pop operation.

Applications of Stack Data Structure

Reversing a String

The characters of a string can be reversed using a stack. We use the basic 'push' and 'pop' functions to do this. First, we push the characters one by one into the stack.

Let's take the word 'MASAI' as an example.



As stack follows the last in first out principle, the last element to go in would be the first one to come out. We'll use the pop function one by one to print the reversed string.

Expression evaluation

Stack is used for the evaluation of a given expression. Let's take the following expression as an example-

$$6 * (2 + 4) + 16 / 4$$

It wouldn't be wrong to say that this expression requires a list of operations. These operations are put in a stack in the order of high precedence.

As parenthesis (bracket) have the highest precedence, we solve the numbers inside the bracket first to get –

$$6 * 6 + 16 / 4$$

Once the parenthesis is accessed and returns a value, the division is next inline –

$$6 * 6 + 4$$

Now, we move to the multiplication part and get :

36 + 4,

The value returned after the addition is 40.

This is how arithmetic operations are kept in a stack to evaluate different expressions.

Converting Decimal to binary

We use stacks to find the equivalent binary code for a decimal number.

How?

We write a program in which – we divide any decimal value by 2 and store the remainder each time (be it 0 or 1) in a stack. Once the value reaches 0 (no digit left), we pop one digit at a time from the stack and print the binary number.

Since a stack remembers the sequence in which we call the functions, we get the correct returns.

Differences between stack and Linked List.

- A stack is an abstract data type which is basically a collection of elements like a pile of books. There are basically with two principal operations in stack, which are known as push and pop.
Where as, a linked list is a linear collection of data elements known as nodes where each node consists value of that node and a reference pointer which points to the next node. The order of the nodes is not given by their location in memory. Thus, this is the main difference between stack and linked list.
- Push, pop and peek are the main operations performed on a stack while insert, delete and traversing are the main operations performed on a linked list.
- In a stack, we can only access the topmost element of the stack.
Where as, On the other hand in a linked list, if we want to access a specific element which is present after i elements, then it is necessary to

traverse the first i elements from the beginning of the Linked List to access that particular element.

- A stack works on the principal of LIFO mechanism, in which the last element inserted will be removed first and both insertion and deletion will take place from that one end only.
whereas, in a linked list, the elements connect to each other by references. Hence, this is another difference between stack and linked list.
- Implementing a stack is less complex as compared to implementing a Linked List. The structure of stack is also less complex as compared to the linked list.

Similarities between stack and Linked List.

- Stack and Linked List both are two different linear data structures. we can implement both of these data structures using any programming language.
- Both of them are flexible in size and can grow according to requirement of input.
- we can implement both stack and Linked List using an array.

TASKS:

IMPLEMENTATION OF STACK

there are two ways of implementing stack

I. using arrays.

II. using linked list.

LECTURE 10:

QUEUES DATA STRUCTURE

What is Queue Data Structure?

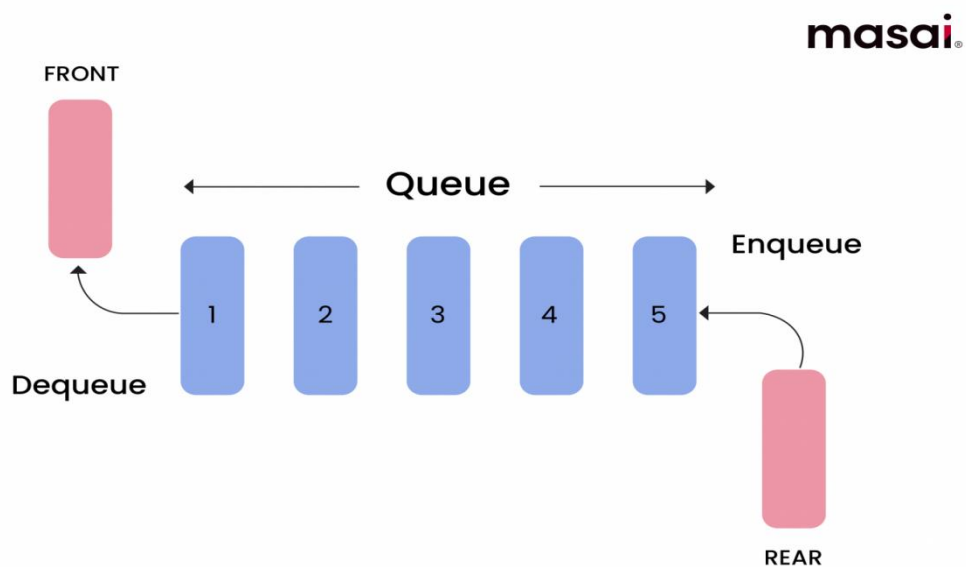
A queue is a linear data structure that contains elements in an ordered sequence. It is an abstract data type, pretty similar to a **stack**.

But unlike stacks, we can perform operations at both ends of a queue.

We insert data at one end of the queue and remove data from the other end.

Think of a queue at the cinema theatre counter. The person standing first in the queue gets the ticket and leaves the group, while a new person joins at the back end of the queue.

This is at the center of the working of a queue, known as the **First In First Out** principle.



Basic Operations of a Queue

We can perform the following operations on a queue-

- **Enqueue()** – This is the process of adding or storing an element to the rear end(back-end) of a queue.
- **Dequeue()** – It refers to removing or accessing an element from the front-end of a queue.
- **peek()** – This function brings the asked element to the front-end of the queue without removing it.
- **isEmpty()** – It checks if the queue is empty.
- **isFull()** – It checks if the queue is full.

By now you would have understood, that we need two pointers for two different functions in a queue. The front pointer is used to access or Dequeue an element, while the rear pointer points to the new element that is added.

Applications of Queue Data Structure

As a queue data structure is an ordered list and yet open at both ends, it has multifold applications in the real world. Let's look at a few of them –

Job scheduling

The most important application of a queue is **CPU scheduling**. The tasks that a computer executes are scheduled in the processor/CPU using a queue. That's how the computer executes these tasks one by one based on the order.

Other single shared resources such as disks and printers also use queues as waiting lists.

Asynchronous transfer of data

Queues are used in asynchronous data transfer. Asynchronous here means the process where data is not being transferred at the same rate between two processes. For example – IO buffers, pipes, file IO, etc.

Traffic System

In a computer-controlled traffic system, we use circular queues to switch on the different lights one by one periodically (at regular time intervals).

We also use queues in the handling of website traffic and real-time system interrupts.

In Networks:

- In routers
- Queues in mail scheduling

TASK :

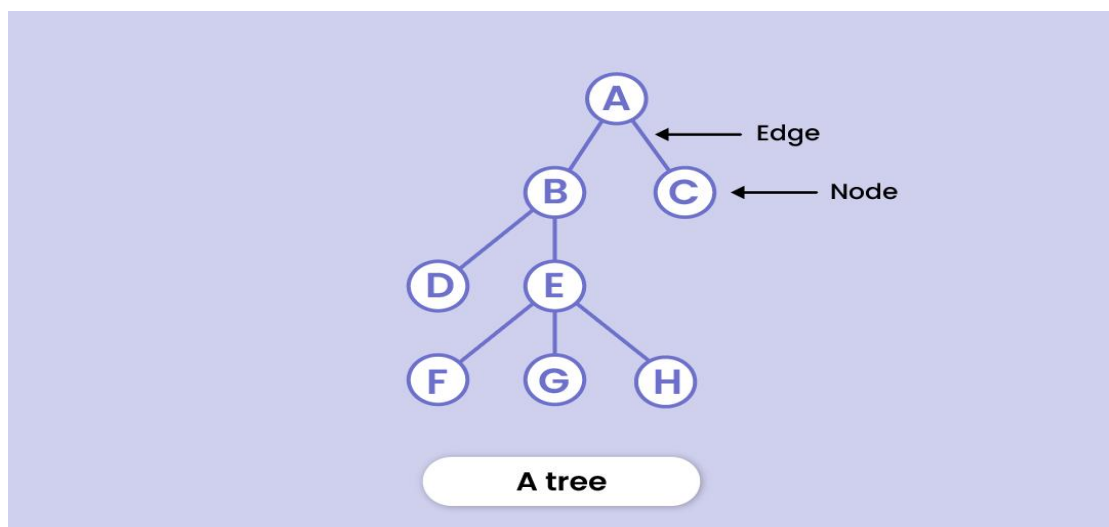
Implement queue operations using arrays.

LECTURE 11: NON LINEAR DATA STRUCTURE

TREE

What is Tree Data Structure?

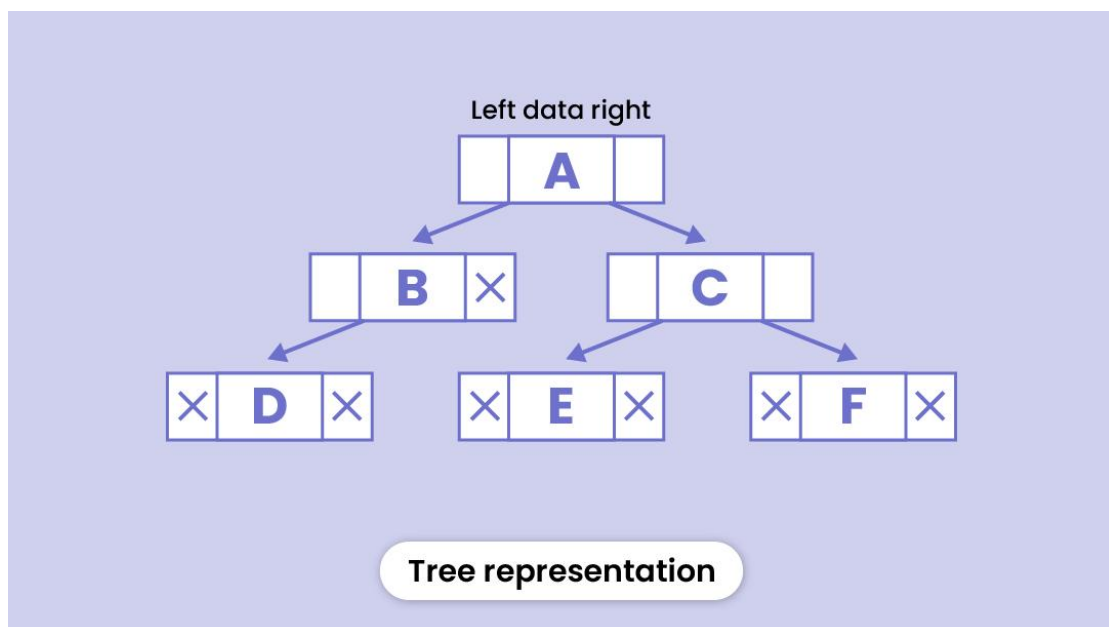
A tree is a hierarchical data structure that contains a collection of nodes connected via edges so that each node has a value and a list of pointers to other nodes. (Similar to how parents have the reference to their children in a family tree)



Following are the key points of a tree data structure:

- Trees are specialized to represent a hierarchy of data
- Trees can contain any type of data be it strings or numbers

Here's the logical representation of a tree:



Basic terms of a tree data structure

Root node

The topmost node in the tree hierarchy is known as the root node. In other words, it's the origin point of the tree that doesn't have any parent node.

Child node

The descendant of any node is said to be a child node.

Parent node

If the node has a descendant i.e. a sub-node, it's called the parent of that sub-node.

Siblings

Just like a family tree, nodes having the same parent node are called siblings.

Leaf node

It's the bottom-most node in the tree that doesn't have any child node. They are also called external nodes. The "end" of a tree branch is normally where the leaf nodes are located.

Internal node

A node having at least one child node is known as an internal node. Internal nodes in a tree structure are often located "between" other nodes.

Ancestor Node

Any node that lies between the root and the current node is considered to be an ancestor node. You may think of ancestor nodes as "parents, grandparents, etc."

Descendant

A descendant is any child, grandchild, great-grandchild, etc. of the current node. Any node in the tree structure that is "below" the current node is referred to as a descendant.

Edges

The link between any two nodes is called an edge.

Height

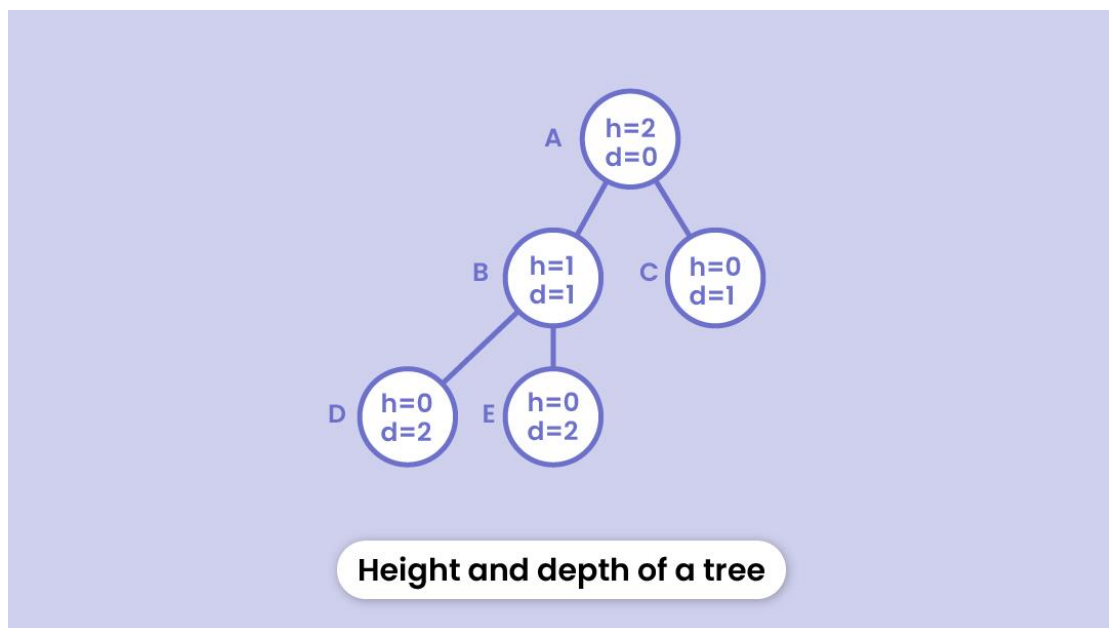
The height of a node is the number of edges from that node to the leaf node (the lowermost node in the hierarchy)

In the figure below - the height of node B will be the number of edges from B to the external node i.e. D. Thus $h = 1$)

Depth

The depth of a node is the number of edges it takes from the root node to that particular node.

In the figure below - the depth of node D will be the number of edges from A to D. Thus $d = 2$)

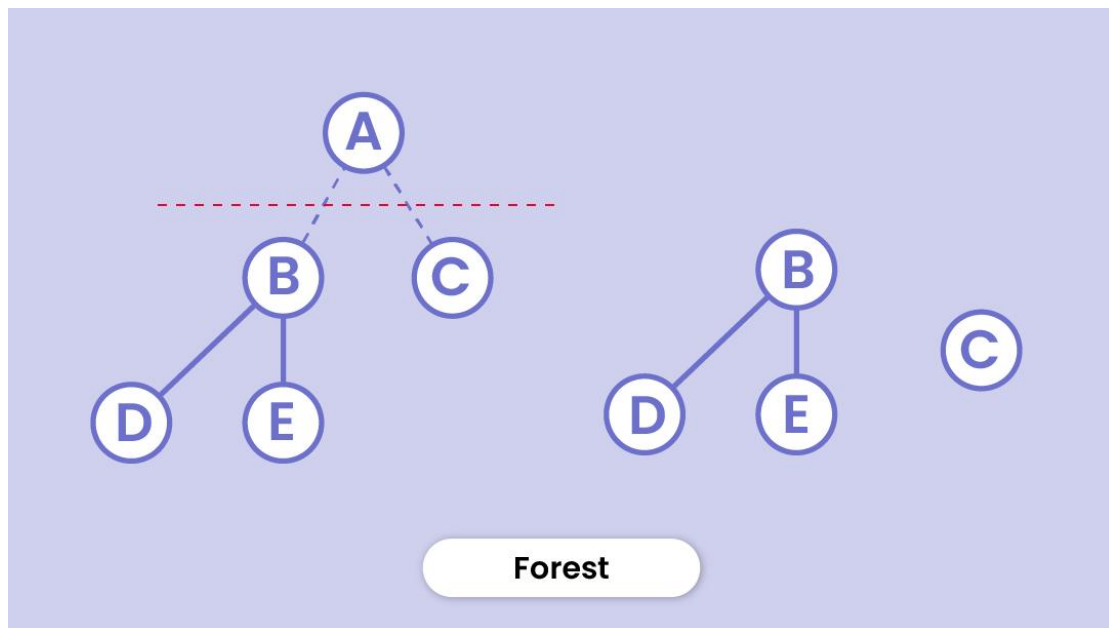


Degree

The total number of branches coming out of a node is considered to be the degree of that node.

Forest

It's not hard to guess, is it? A collection of disconnected trees is called a forest. If you cut the root of a tree, the disjoint trees hence formed make up a forest. (As shown in the figure)



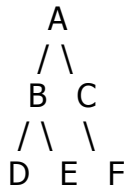
The characteristics of tree data structures

A tree, which replicates a hierarchical tree structure with a collection of connected nodes, is a common data structure in computer science. The following characteristics of a tree data structure:

- Number of edges
- Recursive data structure
- Height of node x
- Depth of node x

Continue reading to understand more about each of these tree data structure's characteristics in greater depth.

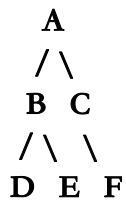
Number of edges: A tree's edge count is always one less than its node count. The reason for this is that on any path from the root to any leaf node, there is always one less edge than there are nodes.



- Nodes: A, B, C, D, E, F (6 nodes)
- Edges: AB, AC, BD, BE, CF (5 edges)

Recursive data structure: When a tree comprises a root node and nodes that have zero or more children each, a tree is a recursive data structure. The tree's offspring nodes are situated below the root node, which is the uppermost node. The term "n-aryl tree" refers to a tree whose nodes each have one or more children.

Example: The tree can be broken down recursively:

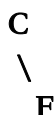


The tree rooted at B is:



B is a subtree with root B and children D and E.

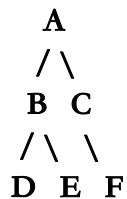
The tree rooted at C is:



C is a subtree with root C and child F.

Height of node x: The distance along the longest route leading from a node to any leaf node is used to measure a node's height. To put it another way, it is just the quantity of edges along the route leading from that leaf node to the deepest leaf node.

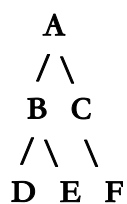
Example: Using the same tree:



- The height of node A is 2 (longest path A -> B -> D or A -> B -> E or A -> C -> F).
- The height of node B is 1 (longest path B -> D or B -> E).
- The height of node C is 1 (longest path C -> F).
- The height of nodes D, E, and F is 0 (they are leaf nodes).

Depth of node x: The distance along the shortest route from the root to a node is referred to as the node's depth. In other words, it only refers to how many edges there are on the route leading from the root to that specific node.

Example: Using the same tree:



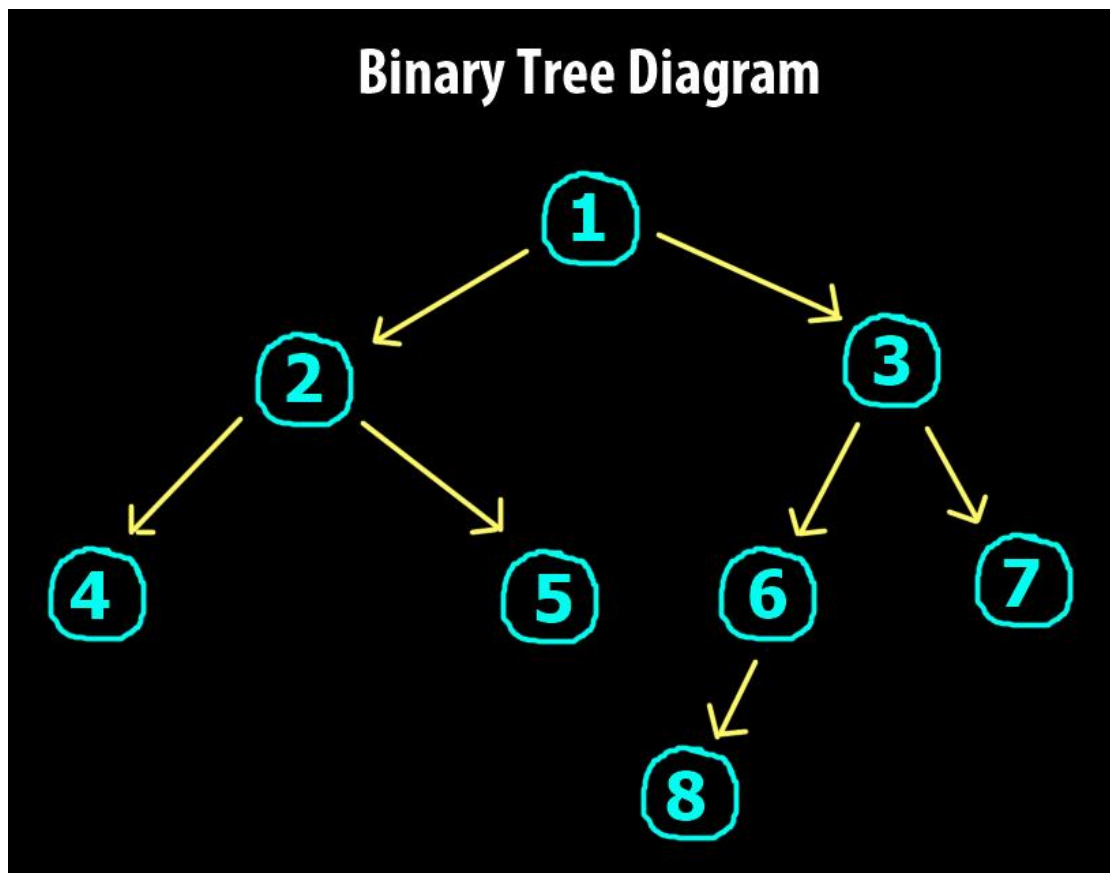
- The depth of node A (the root) is 0.
- The depth of node B is 1 (path A -> B).
- The depth of node C is 1 (path A -> C).
- The depth of node D is 2 (path A -> B -> D).
- The depth of node E is 2 (path A -> B -> E).
- The depth of node F is 2 (path A -> C -> F).

Binary Tree Data Structure

What is a Binary Tree Data Structure ?

A **binary tree** is a tree data structure in which each node has **at most two children**, which are referred to as the **left child(LC)** and the **right child(RC)**.

Binary tree (Logical Diagram)

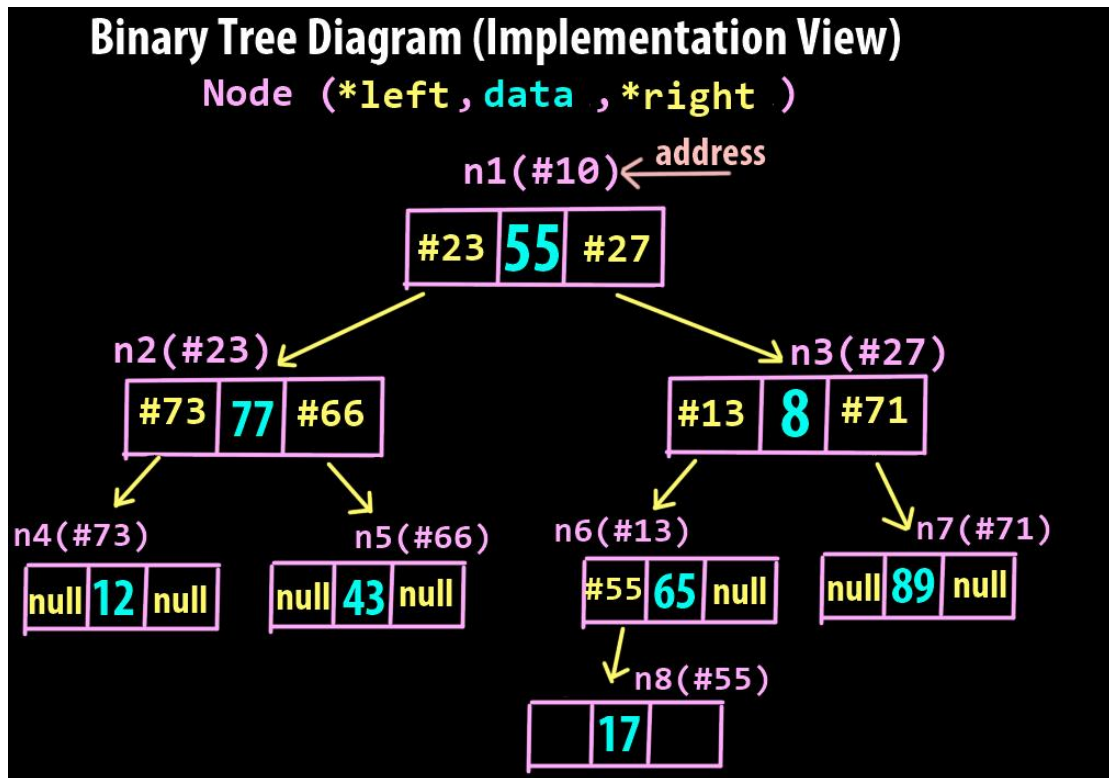


Important Binary Tree Terms & Properties —

- A binary tree is called **STRICT/PROPER** binary tree when each node has **2 or 0** children.
- A binary tree is called **COMPLETE** binary tree if all levels except the last are completely filled and all nodes are as left as possible.
- A binary tree is called **PERFECT** binary tree if all levels are completely filled with 2 children each.
- Max number of nodes at a given level ' x ' = 2^x

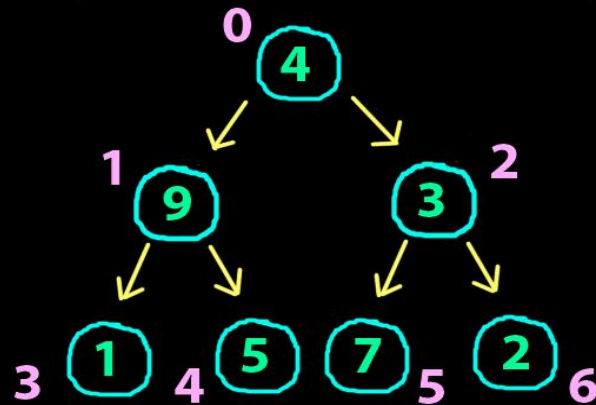
- For a binary tree, maximum number of nodes with height ' h ' = $2^{(h+1)} - 1$
- A binary tree is called **BALANCE** binary tree, if the difference between the height of left subtree and right subtree for every node is not more than k (usually $k=1$).

Binary Tree Implementation view (Dynamic Nodes in Memory) —



Binary Tree Implementation view (Array Implementation) –

2. Binary Tree *as conventional arrays*



arr =

4	9	3	1	5	7	2
0	1	2	3	4	5	6

For node at index i
Left child index = $2i+1$ Right child index = $2i+2$

BINARY TREE OPERATIONS

Binary Tree Traversal

Going through a Tree by visiting every node, one node at a time, is called **traversal**.

Since Arrays and Linked Lists are linear data structures, there is only one obvious way to traverse these: start at the first element, or node, and continue to visit the next until you have visited them all.

But since a Tree can branch out in different directions (non-linear), there are different ways of traversing Trees.

There are two main categories of Tree traversal methods:

Breadth First Search (BFS) is when the nodes on the same level are visited before going to the next level in the tree. This means that the tree is explored in a more sideways direction.

Depth First Search (DFS) is when the traversal moves down the tree all the way to the leaf nodes, exploring the tree branch by branch in a downwards direction.

There are three different types traversals:

- pre-order
- in-order
- post-order

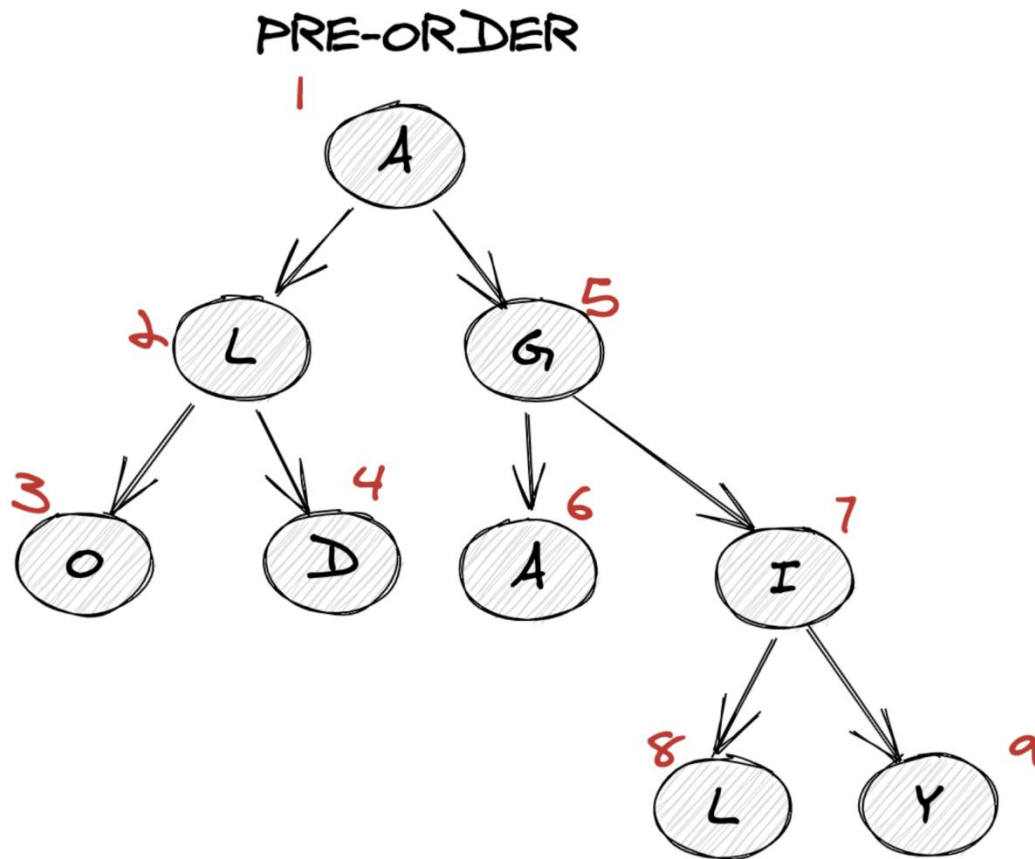
Pre-order Traversal of Binary Trees

Pre-order Traversal is done by visiting the root node first, then recursively do a pre-order traversal of the left subtree, followed by a recursive pre-order traversal of the right subtree. It's used for creating a copy of the tree, prefix notation of an expression tree, etc.

This traversal is "pre" order because the node is visited "before" the recursive pre-order traversal of the left and right subtrees.

Preorder traversal

1. Visit root node
2. Then go to all the nodes in the left subtree
3. Visit all the nodes in the right subtree



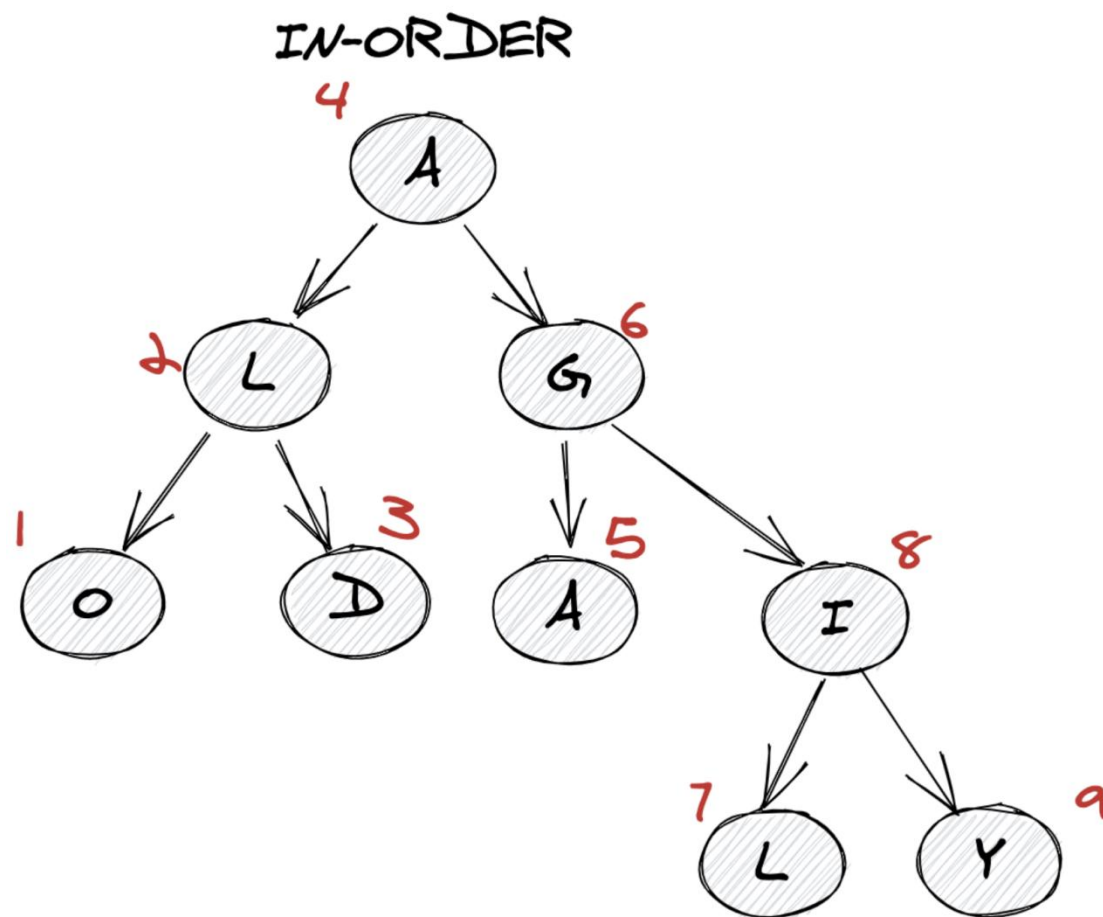
In-order Traversal of Binary Trees

In-order Traversal does a recursive In-order Traversal of the left subtree, visits the root node, and finally, does a recursive In-order Traversal of the right subtree. This traversal is mainly used for Binary Search Trees where it returns values in ascending order.

What makes this traversal "in" order, is that the node is visited in between the recursive function calls. The node is visited after the In-order Traversal of the left subtree, and before the In-order Traversal of the right subtree.

Inorder traversal

1. First, visit all the nodes in the left subtree
2. Then the root node
3. Visit all the nodes in the right subtree



Post-order Traversal of Binary Trees

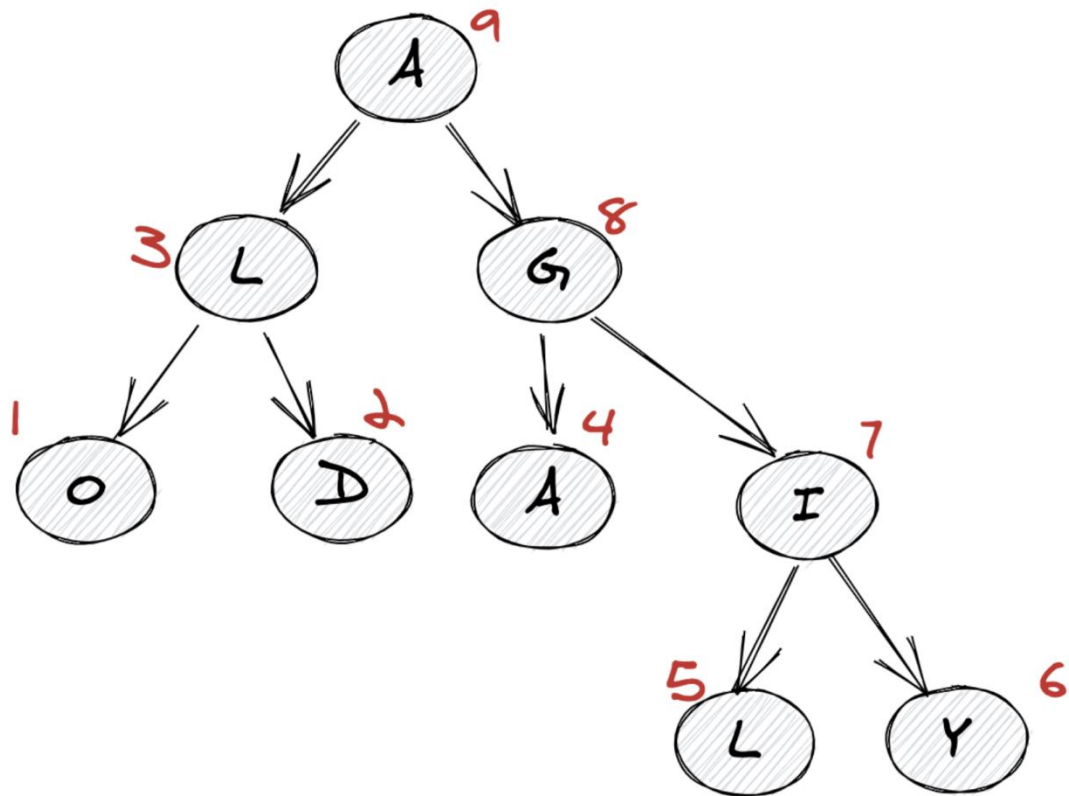
Post-order Traversal works by recursively doing a Post-order Traversal of the left subtree and the right subtree, followed by a visit to the root node. It is used for deleting a tree, post-fix notation of an expression tree, etc.

What makes this traversal "post" is that visiting a node is done "after" the left and right child nodes are called recursively.

Postorder traversal

1. Visit all the nodes in the left subtree
2. Visit all the nodes in the right subtree
3. Visit the root node

POST-ORDER



APPLICATIONS OF BINARY TREES

File Systems

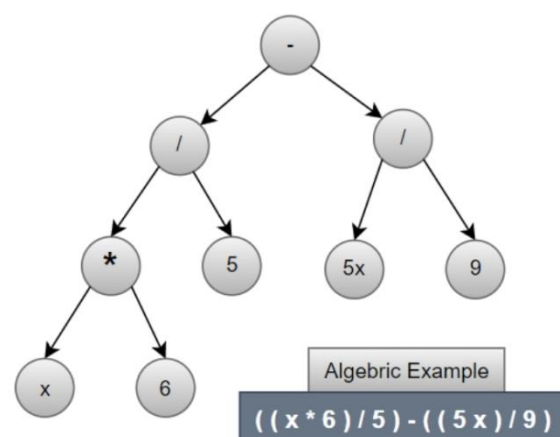
Binary trees are used in file systems to organise and store files. Each node in the tree represents a file or a directory, and the left child of a node represents a subdirectory or a file that is smaller than the node, while the right child represents a subdirectory or a file that is larger than the node.

Search Engines

Binary trees are used in search engines to organise and index web pages. Each node in the tree represents a web page, and the left child of a node represents a web page ranked lower than the node, while the right child represents a web page ranked higher than the node.

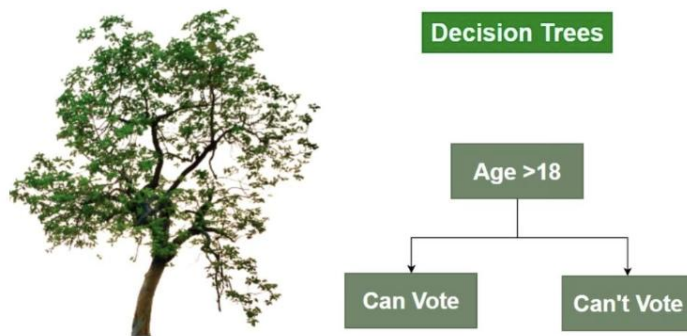
Expression Trees

Are you a maths lover? Because binary trees can also be used to represent maths expressions, such as equations. These are known as expression trees. Expression trees are basically used in evaluating and simplifying expressions and generating code.



Decision Trees

The decision tree is used in machine learning and data mining to model decisions and their consequences. In the decision tree, the inner node represents the test on an attribute, while the leaf node represents the result of the decision. Decision trees are used in various applications, such as fraud detection, customer segmentation, and medical diagnosis.



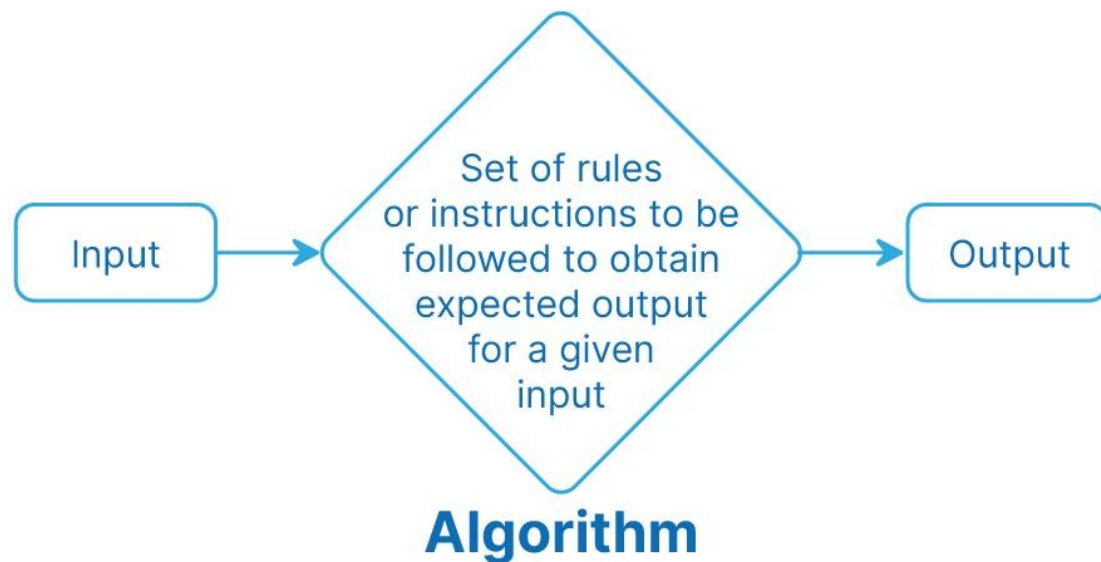
ALGORITHMS

CONCEPT OF ALGORITHMS

What is an Algorithm?

An algorithm is a set of commands that must be followed for a computer to perform calculations or other problem-solving operations. According to its formal definition, an **algorithm** is a finite set of instructions carried out in a specific order to perform a particular task. It is not the entire program or code; it is simple logic to a problem

represented as an informal description in the form of a flowchart or pseudocode.



Problem: A problem can be defined as a real-world problem or real-world instance problem for which you need to develop a program or set of instructions. An algorithm is a set of instructions.

Algorithm: An algorithm is defined as a step-by-step process that will be designed for a problem.

Input: After designing an algorithm, the algorithm is given the necessary and desired inputs.

Processing unit: The input will be passed to the processing unit, producing the desired output.

Output: The outcome or result of the program is referred to as the output.

How do Algorithms Work?

Algorithms are step-by-step procedures designed to solve specific problems and perform tasks efficiently in the realm of computer science and mathematics. These powerful sets of instructions form the backbone of modern technology and govern everything from web searches to artificial intelligence. Here's how algorithms work:

- **Input:** Algorithms take input data, which can be in various formats, such as numbers, text, or images.
- **Processing:** The algorithm processes the input data through a series of logical and mathematical operations, manipulating and transforming it as needed.
- **Output:** After the processing is complete, the algorithm produces an output, which could be a result, a decision, or some other meaningful information.
- **Efficiency:** A key aspect of algorithms is their efficiency, aiming to accomplish tasks quickly and with minimal resources.

- **Optimization:** Algorithm designers constantly seek ways to optimize their algorithms, making them faster and more reliable.
- **Implementation:** Algorithms are implemented in various programming languages, enabling computers to execute them and produce desired outcomes.

What is the Need for Algorithms?

You require algorithms for the following reasons:

Scalability

It aids in your understanding of scalability. When you have a sizable real-world problem, you must break it down into small steps to analyze it quickly.

Performance

The real world is challenging to break down into smaller steps. If a problem can be easily divided into smaller steps, it indicates that the problem is feasible.

After understanding what is an algorithm, why you need an algorithm, you will look at how to write one using an example.

Use of the Algorithms

Algorithms are essential to many disciplines because they offer organized, practical answers to challenging issues. Here are a few significant applications of algorithms in various fields:

1. Data Analysis and Machine Learning

Algorithms are used in data analysis and machine learning to find patterns in big datasets and forecast outcomes. Thanks to machine learning methods like support vector machines, decision trees, and neural networks, computers can learn from data and improve over time. These techniques are essential for applications such as

recommendation systems, natural language processing, and picture recognition.

2. Cryptography & Security

Cryptography algorithms safeguard data by using encryption and decryption techniques, guaranteeing safe data storage and communication. Algorithms such as SHA-256, AES, and RSA are commonly employed in data integrity maintenance, user authentication, and sensitive information security. These algorithms comprise the foundation of cybersecurity measures used in secure communications, data encryption, and online transactions.

3. Information Retrieval and Search Engines

Search engines can efficiently index and retrieve pertinent information thanks to search algorithms such as PageRank and Hummingbird. By prioritizing web pages according to their significance and relevancy, these algorithms assist users in locating the most relevant information available online. Effective search algorithms are necessary to manage the enormous volume of online information.

4. Optimization problems

Optimization methods are utilized to select the optimal answer from various options. In various industries, including banking, engineering, logistics, and artificial intelligence, sophisticated issues are resolved using methods like gradient descent, linear programming, and genetic algorithms. These algorithms increase productivity, reduce expenses, and optimize resources for operations and decision-making.

5. Genomics and medical diagnostics

Due to their ability to analyze medical images, forecast disease outbreaks, and recognize genetic changes, algorithms are indispensable in medical diagnostics. Personalized medicine has been transformed by machine learning algorithms, in particular, which enable the creation of customized treatment regimens based on each patient's unique genetic profile. Additionally, algorithms help to speed up the sequencing and interpretation of genomic data, improving biotechnology and genomics research.

Characteristics of an Algorithm

An algorithm is a methodical process used to solve a task or solve a problem. Several important factors impact an algorithm's effectiveness:

1. Finiteness

An algorithm must always have a finite number of steps before it ends. When the operation is finished, it must have a defined endpoint or output and not enter an endless loop.

2. Definiteness

An algorithm needs to have exact definitions for each step. Clear and straightforward directions ensure that every step is understood and can be taken easily.

3. Input

An algorithm requires one or more inputs. The values that are first supplied to the algorithm

before its processing are known as inputs. These inputs come from a predetermined range of acceptable values.

4. Output

One or more outputs must be produced by an algorithm. The output is the outcome of the algorithm after every step has been completed. The relationship between the input and the result should be clear.

5. Effectiveness

An algorithm's stages must be sufficiently straightforward to be carried out in a finite time utilizing fundamental operations. With the resources at hand, every operation in the algorithm should be doable and practicable.

6. Generality

Rather than being limited to a single particular case, an algorithm should be able to solve a group of issues. It should offer a generic fix that manages a

variety of inputs inside a predetermined range or domain.

Qualities of a Good Algorithm

- **Efficiency:** A good algorithm should perform its task quickly and use minimal resources.
- **Correctness:** It must produce the correct and accurate output for all valid inputs.
- **Clarity:** The algorithm should be easy to understand and comprehend, making it maintainable and modifiable.
- **Scalability:** It should handle larger data sets and problem sizes without a significant decrease in performance.
- **Reliability:** The algorithm should consistently deliver correct results under different conditions and environments.
- **Optimality:** Striving for the most efficient solution within the given problem constraints.
- **Robustness:** Capable of handling unexpected inputs or errors gracefully without crashing.

- **Adaptability:** Ideally, it can be applied to a range of related problems with minimal adjustments.
- **Simplicity:** Keeping the algorithm as simple as possible while meeting its requirements, avoiding unnecessary complexity.

Advantage and Disadvantages of Algorithms

Advantages of Algorithms:

- **Efficiency:** Algorithms streamline processes, leading to faster and more optimized solutions.
- **Reproducibility:** They yield consistent results when provided with the same inputs.
- **Problem-solving:** Algorithms offer systematic approaches to tackle complex problems effectively.
- **Scalability:** Many algorithms can handle larger datasets and scale with increasing input sizes.
- **Automation:** They enable automation of tasks, reducing the need for manual intervention.

Disadvantages of Algorithms:

- **Complexity:** Developing sophisticated algorithms can be challenging and time-consuming.
- **Limitations:** Some problems may not have efficient algorithms, leading to suboptimal solutions.
- **Resource Intensive:** Certain algorithms may require significant computational resources.
- **Inaccuracy:** Inappropriate algorithm design or implementation can result in incorrect outputs.
- **Maintenance:** As technology evolves, algorithms may require updates to stay relevant and effective.

HOW TO WRITE AN ALGORITHMS

as we all know, an algorithms is a basic tool and very useful for understanding a problem through it's step by step analysis . However there are no well-defined standards for writting algorithms;- althought it depends on the problem and available resources.

the following are the steps:-

- a. determine the input and output of the problem.
- b. find the correct data structure to present the problem.
- c. try to reduce the problem to a variation of a well-known one.
- d. decide whether you look for a recursive or imperative one or mixed algorithm.
- e. write an algorithm.

example:- Algorithm for multiplication of two numbers.

solution:-

step 1: Start

step2: Enter the first number.

step 3: Enter the second number.

step 4: Multiply two numbers and store the result.

step 5: Display the result.

step 6: Stop.

TASK:

Write the following algorithms

1. algorithm to find the largest number between two numbers.
2. write an algorithm to find the area of a square.
3. write an algorithms to find simple interest.
4. write an algorithm to find compound interest.

TYPES OF ALGORITHMS.

There are many algorithms, but for your level we are going to learn two fundamental types :-

i. Recursive Algorithms.

ii. Iterative Algorithms.

RECURSIVE ALGORITHMS

This is an algorithm that calls itself repeatedly with a smaller value as inputs generated after solving the current inputs until the problem is solved.

Recursion is a programming technique where a function calls itself to solve smaller instances of a problem until a base case is met, at which point the recursive calls stop.

Key Concepts:

- **Base Case:** The condition under which the recursion ends. Without a base case, the recursion would continue indefinitely, leading to a stack overflow.

- **Recursive Case:** The part of the function where it calls itself with a modified argument, moving towards the base case.

Analogy: Think of recursion as standing between two mirrors: each reflection is a smaller version of the original image, continuing until it becomes too small to see (the base case).

Why Use Recursion? Recursion is often used to simplify the code for problems that have a natural recursive structure, like tree traversals, the Fibonacci sequence, and factorial calculations.

Implementation of Recursive Algorithms

Algorithm: Recursive Factorial Calculation

Input:

- A non-negative integer n .

Output:

- The factorial of n , denoted as $n!$.

Steps:

1. Base Case:

- If n is 0 or 1, then:
 - Return 1.
 - (Reason: By definition, $0! = 1$ and $1! = 1$).

2. Recursive Case:

- If n is greater than 1, then:
 - Calculate the factorial of $(n - 1)$ by calling the algorithm recursively.
 - Multiply n by the result of the recursive call.
 - Return the product.

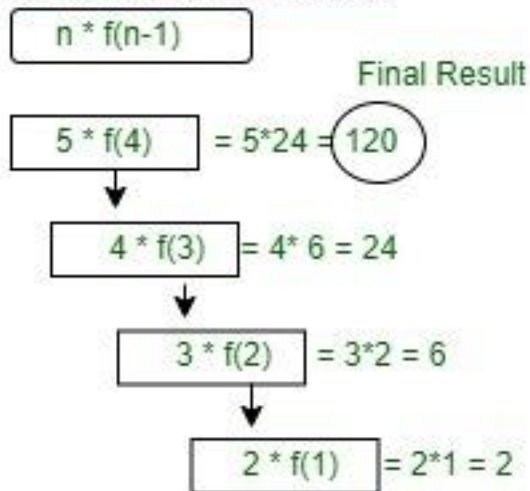
3. End of Algorithm.

Example:

- If $n = 5$:
 - The algorithm calculates $5 * (4 * (3 * (2 * (1 * 1)))) = 120$.

For user input : 5

Factorial Recursion Function



Advantages and Disadvantages of Recursive Algorithms

Advantages:

- it reduces the number of lines of code in a program.
- it is clear and straight for the programmer to understand.
- recursive algorithms simply solve complex problems.

Disadvantages:

- **Performance Overhead:** Recursive calls can lead to significant overhead due to function call stack management, especially for deep recursions.

- **Stack Overflow:** If the recursion depth is too high (e.g., no proper base case or a large input), it can cause a stack overflow.
- **Difficult to Debug:** Recursive functions can be harder to trace and debug compared to iterative solutions, especially for complex problems.

TASK:

1. Write a recursive algorithm to find the sum of n-integers.
2. differentiate btn algorithm and program.
3. explain which effect will be arisen when the base case is not defined in a recursive function?

ITERATIVE ALGORITHMS

Definition: Iterative algorithms repeatedly execute a set of instructions until a specific condition is met. Unlike recursion, which solves a problem by breaking it down into smaller sub-problems, iteration solves a problem by repeatedly applying the same operations.

Key Concepts:

- **Loop:** The primary structure used in iterative algorithms, such as for, while, or do-while loops. A loop allows you to execute a block of code multiple times.
- **Condition:** A boolean expression that determines when the loop should stop. It's checked before each iteration.
- **Iteration:** Each execution of the loop's body is called an iteration.

Why Use Iteration? Iteration is often more efficient than recursion in terms of memory usage because it doesn't involve the overhead of multiple function calls. It's ideal for problems where you need to repeat an action a known number of times or until a condition is met.

Implementation of Iterative Algorithms

Basic Structure:

1. **Initialize variables:** Set up any variables that control the loop or store results.
2. **Loop condition:** Define the condition under which the loop will continue executing.
3. **Loop body:** Write the code that will be executed during each iteration.

4. **Update:** Modify the loop control variable to ensure that the loop progresses toward termination.
5. **Exit the loop:** When the loop condition is no longer true, the loop stops, and the algorithm proceeds to the next step.

Examples of Iterative Algorithms

Example 1: Factorial Calculation (Iterative Approach)

Problem: Compute the factorial of a number n , denoted as $n! = n * (n-1) * (n-2) * \dots * 1$.

Algorithm:

Input:

A non-negative integer n .

Output:

The factorial of n .

Steps:

1. Initialize result to 1.
2. For each integer i from 1 to n :
 - Multiply result by i .
3. After the loop ends, result holds the value of $n!$.

4. Return result.

Example:

1. If $n = 5$, the loop calculates $1 * 1 * 2 * 3 * 4 * 5 = 120$.

Advantages and Disadvantages of Iterative Algorithms

Advantages:

- **Efficiency:** Iterative algorithms generally use less memory than recursive algorithms because they don't involve the overhead of multiple function calls.
- **Simplicity:** They are easier to understand and debug for problems that involve simple repetition.
- **Avoidance of Stack Overflow:** Since they don't involve deep recursion, iterative algorithms are less likely to cause stack overflow.

Disadvantages:

- **Code Complexity:** For problems that naturally fit a recursive pattern, iterative solutions can be more complex and harder to write.
- **Less Intuitive:** Some problems, like tree traversals, are less intuitive when solved iteratively, leading to more complicated code.

TASK:

1. a.) design an algorithm to find the sum of the digits made from a positive number.

b.)create a c++ program to implement the algorithms.
2. what are the advantages of using iterative over recursive algorithms?explain.

SEARCHING ALGORITHMS

Searching algorithms are essential tools in computer science used to locate specific items within a collection of data. These algorithms are designed to efficiently navigate through data structures to find the desired information, making them fundamental in various applications such as databases, web search engines, and more.

What is Searching?

Searching is the fundamental process of locating a specific element or item within a collection of data. This collection of data can take various forms, such as arrays, lists, trees, or other structured representations. The primary objective of searching is to determine whether the desired element exists within the data, and if so, to identify its precise location or retrieve it. It plays an important role in various computational tasks and real-world applications, including information retrieval, data analysis, decision-making processes, and more.

Searching terminologies:

Target Element:

In searching, there is always a specific target element or item that you want to find within the data collection. This target could be a value, a record, a key, or any other data entity of interest.

Search Space:

The search space refers to the entire collection of data within which you are looking for the target element. Depending on the data structure used, the search space may vary in size and organization.

Complexity:

Searching can have different levels of complexity depending on the data structure and the algorithm used. The complexity is often measured in terms of time and space requirements.

Applications of Searching:

Searching algorithms have numerous applications across various fields. Here are some common applications:

- **Information Retrieval:** Search engines like Google, Bing, and Yahoo use sophisticated searching algorithms to retrieve relevant information from vast amounts of data on the web.
- **Database Systems:** Searching is fundamental in database systems for retrieving specific data records based on user queries, improving efficiency in data retrieval.
- **E-commerce:** Searching is crucial in e-commerce platforms for users to find products quickly based on their preferences, specifications, or keywords.
- **Networking:** In networking, searching algorithms are used for routing packets

efficiently through networks, finding optimal paths, and managing network resources.

- **Artificial Intelligence:** Searching algorithms play a vital role in AI applications, such as problem-solving, game playing (e.g., chess), and decision-making processes
- **Pattern Recognition:** Searching algorithms are used in pattern matching tasks, such as image recognition, speech recognition, and handwriting recognition.

Linear Search Algorithm

The linear search algorithm is defined as a sequential search algorithm that starts at one end and goes through each element of a list until the desired element is found; otherwise, the search continues till the end of the dataset. In this article, we will learn about the basics of the linear search algorithm, its applications, advantages, disadvantages, and more to provide a deep understanding of linear search.

What is Linear Search Algorithm?

Linear search is a method for searching for an element in a collection of elements. In linear search, each element of the collection is visited one by one in a sequential fashion to find the desired element. Linear search is also known as sequential search.

Algorithm for Linear Search Algorithm:

The algorithm for linear search can be broken down into the following steps:

- a. **Start:** Begin at the first element of the collection of elements.
- b. **Compare:** Compare the current element with the desired element.
- c. **Found:** If the current element is equal to the desired element, return true or index to the current element.
- d. **Move:** Otherwise, move to the next element in the collection.

- e. **Repeat:** Repeat steps 2-4 until we have reached the end of collection.
- f. **Not found:** If the end of the collection is reached without finding the desired element, return that the desired element is not in the array.

How Does Linear Search Algorithm Work?

In Linear Search Algorithm,

Every element is considered as a potential match for the key and checked for the same.

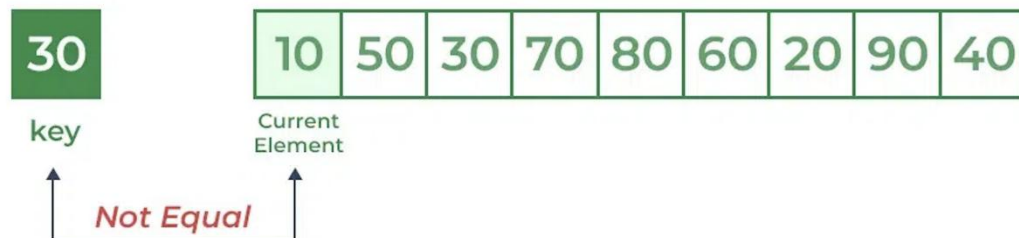
If any element is found equal to the key, the search is successful and the index of that element is returned.

If no element is found equal to the key, the search yields "No match found".

For example: Consider the array `arr[] = {10, 50, 30, 70, 80, 20, 90, 40}` and `key = 30`

Step 1: Start from the first element (index 0) and compare key with each element (`arr[i]`).

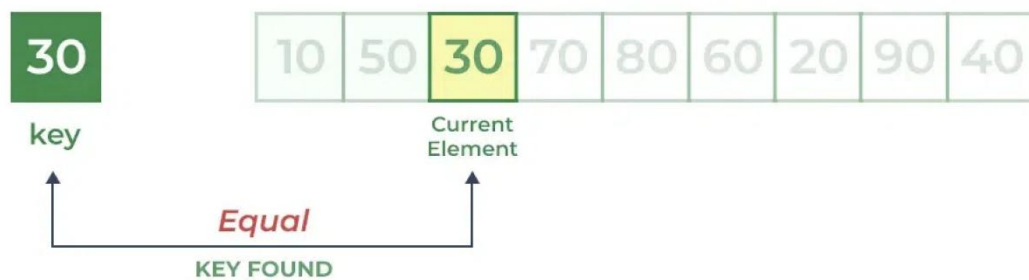
Comparing key with first element $\text{arr}[0]$. Since not equal, the iterator moves to the next element as a potential match.



Comparing key with next element $\text{arr}[1]$. Since not equal, the iterator moves to the next element as a potential match.



Step 2: Now when comparing $\text{arr}[2]$ with key, the value matches. So the Linear Search Algorithm will yield a successful message and return the index of the element when key is found (here 2).



Time and Space Complexity of Linear Search Algorithm:

Time Complexity:

Best Case: In the best case, the key might be present at the first index. So the best case complexity is $O(1)$

Worst Case: In the worst case, the key might be present at the last index i.e., opposite to the end from which the search has started in the list. So the worst-case complexity is $O(N)$ where N is the size of the list.

Average Case: $O(N)$

Auxiliary Space: $O(1)$ as except for the variable to iterate through the list, no other variable is used.

Applications of Linear Search Algorithm:

- **Unsorted Lists:** When we have an unsorted array or list, linear search is most commonly used to find any element in the collection.
- **Small Data Sets:** Linear Search is preferred over binary search when we have small data sets with
- **Searching Linked Lists:** In linked list implementations, linear search is commonly used to find elements within the list. Each node is checked sequentially until the desired element is found.
- **Simple Implementation:** Linear Search is much easier to understand and implement as compared to Binary Search or Ternary Search.

Advantages of Linear Search Algorithm:

- Linear search can be used irrespective of whether the array is sorted or not. It can be used on arrays of any data type.
- Does not require any additional memory.

- It is a well-suited algorithm for small datasets.

Disadvantages of Linear Search Algorithm:

- Linear search has a time complexity of $O(N)$, which in turn makes it slow for large datasets.
- Not suitable for large arrays.

When to use Linear Search Algorithm?

- When we are dealing with a small dataset.
- When you are searching for a dataset stored in contiguous memory.

BINARY SEARCHING ALGORITHM

What is Binary Search Algorithm?

Binary search is a search algorithm used to find the position of a target value within a sorted array. It works by repeatedly dividing the search interval in half until the target value is found or the interval is empty. The search interval is halved by comparing the target element with the middle value of the search space.

Conditions to apply Binary Search Algorithm in a Data Structure:

Binary Search is a highly efficient algorithm for finding an element in a sorted data structure, like an array. However, to successfully apply the Binary Search algorithm, certain conditions must be met:

1. The Data Structure Must Be Sorted

- **Condition:** The array or data structure must be sorted in ascending or descending order.
- **Reason:** Binary Search works by repeatedly dividing the search interval in half. If the data is not sorted, this division won't correctly narrow down the search area. In an unsorted array, the algorithm has no reliable way of

determining which half of the data to discard, making the search ineffective.

- **Example:** If you have an array [10, 20, 30, 40, 50], you can apply Binary Search to find any element because the array is sorted. However, with an unsorted array like [30, 10, 50, 20, 40], Binary Search would fail.

2. Access to Any Element of the Data Structure Should Take Constant Time ($O(1)$)

- **Condition:** You should be able to access any element in the data structure in constant time, denoted as $O(1)$.
- **Reason:** Binary Search requires direct access to elements at specific indices (mid-points) to function efficiently. For instance, if you're searching for an element, Binary Search calculates the mid-point and immediately accesses that element. If accessing an element takes more than constant time, the efficiency of Binary Search decreases significantly.
- **Example:** Arrays allow $O(1)$ access to elements because they are contiguous blocks of memory. However, in a linked list, access to elements is sequential, not constant time, so Binary Search is not practical.

Binary Search Algorithm:

Below is the step-by-step algorithm for Binary Search:

- Divide the search space into two halves by finding the middle index "mid".
- Compare the middle element of the search space with the key.
- If the key is found at middle element, the process is terminated.
- If the key is not found at middle element, choose which half will be used as the next search space.
- If the key is smaller than the middle element, then the left side is used for next search.
- If the key is larger than the middle element, then the right side is used for next search.
- This process is continued until the key is found or the total search space is exhausted.

Applications of Binary Search Algorithm:

- Binary search can be used as a building block for more complex algorithms used in machine learning, such as algorithms for training neural networks or finding the optimal hyperparameters for a model.
- It can be used for searching in computer graphics such as algorithms for ray tracing or texture mapping.
- It can be used for searching a database.

Advantages of Binary Search:

- Binary search is faster than linear search, especially for large arrays.
- More efficient than other searching algorithms with a similar time complexity, such as interpolation search or exponential search.
- Binary search is well-suited for searching large datasets that are stored in external memory, such as on a hard drive or in the cloud.

Disadvantages of Binary Search:

- The array should be sorted.
- Binary search requires that the data structure being searched be stored in contiguous memory locations.
- Binary search requires that the elements of the array be comparable, meaning that they must be able to be ordered.

TASKS:

Question: Explain the time complexity of both linear and binary search algorithms. Why is binary search more efficient than linear search in a sorted array?

Question: Write a c++ program to implement the binary search algorithms.

Question: Write a C++ program to implement a linear search algorithm that searches for a given element in an array. The program should output the index of the element if found, or -1 if the element is not found.

SORTING ALGORITHMS.

Introduction to Sorting Algorithms

What is Sorting?

Sorting is the process of arranging data in a particular order, typically in ascending or descending order. Sorting is a fundamental operation in computer science, widely used in various applications such as searching, data analysis, and database management.

Why Sorting is Important?

Sorting is essential because:

- **Improved Search Efficiency:** Sorted data allows for faster search operations, especially when using algorithms like binary search.
- **Data Organization:** Sorting helps in organizing data in a meaningful way, making it easier to analyze and understand.
- **Optimization:** Many algorithms, such as those in data processing or networking, work more efficiently on sorted data.
- **Real-World Applications:** Sorting is used in various real-world scenarios like arranging names alphabetically, prioritizing tasks, or ranking search engine results.

Types of Sorting Algorithms

Sorting algorithms can be broadly classified into different kinds including:-

- Bubble Sort
- Selection Sort
- Insertion Sort
- Merge Sort
- Quick Sort
- Heap Sort

SELECTION SORTING.

Selection Sort

- Selection sort is a sorting algorithm, specifically an **in-place comparison sort**.
- It has $O(n^2)$ time complexity, making it inefficient on large lists.

- The algorithm divides the input list into two parts: the sublist of items already sorted, which is built up from left to right at the front (left) of the list, and the sublist of items remaining to be sorted that occupy the rest of the list.
- Initially, the sorted sublist is empty and the unsorted sublist is the entire input list. The algorithm proceeds by finding the smallest (or largest, depending on sorting order) element in the unsorted sublist, exchanging (swapping) it with the leftmost unsorted element (putting it in sorted order), and moving the sublist boundaries one element to the right.

Selection Sort Algorithm

```
for i=0 to n-1 // outer for loop
  min = i // set min value to i
  for j=i+1 to n // inner loop
    if arr[j] less than arr[min] then // check which element is smaller
      min=j // store index of smallest element to min
  end for // inner loop
  if min not equal to i then // swap if min does not match to i
    swap arr[min] with arr[i] // swapping
  end for // outer for loop
```

TASK: C++ Program to Implement Selection Sort

INSERTION SORTING.

- Insertion sort is a simple sorting algorithm that works the way we sort playing cards in our hands.
- Insertion sort is a simple sorting algorithm that builds the final sorted array (or list) one item at a time.
- Time Complexity: $O(n^2)$
- Efficient for (quite) small data sets, much like other quadratic sorting algorithms
- More efficient in practice than most other simple quadratic (i.e., $O(n^2)$) algorithms such as selection sort or bubble sort

Working

null

- Step 1 – Pick next element
- Step 2 – Compare with all elements in the sorted sub-list on the left
- Step 3 – Shift all the elements in the sorted sub-list that is greater than the value to be sorted.
- Step 4 – Insert the value
- Step 5 – Repeat until list is sorted

Insertion Sort Algorithm

```
declare variables – i, key, j
loop : i = 1 to n – 1 // outer loop
  key = a[i] //pick the next element
  j = i – 1; // decrement j value
  loop : (j >= 0 && a[j] > key) // inner loop
    arr[j+1] = arr[j]
    j = j – 1
  end loop // outer loop
  arr[j+1] = key
end loop // outer loop
```

TASK: C++ Program to Implement Insertion Sort

BUBBLE SORTING.

- Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.
- This algorithm is suitable for small data sets
- Its average and worst case complexity are of (n^2) where n is the number of items.
- It is known as bubble sort, because with every complete iteration the largest element bubbles up towards the last place or the highest index just like a water bubble rises up to the water surface.

Working

- Step 1 — Starting with the first element(index = 0), compare the current element with the next element of the array.
- Step 2 — If the current element is greater than the next element of the array, swap them.

- Step 3 — If the current element is less than the next element, move to the next element.
- Step 4 — Repeat Step 1 till the list is sorted.

Bubble Sort Algorithm

```
declare variables — i, j
loop : i = 0 to n - 1 // outer loop
  loop : j = 0 to n - i - 1 // inner loop
    if ( a[j] > a[j+1] ) then
      swap a[j] & a[j+1]
  end loop // inner loop
end loop // outer loop
```

TASK: C++ Program to Implement Bubble Sort

Quick Sort Algorithm

Quick Sort Algorithm is a **Divide & Conquer** algorithm. It divides input array in two partitions, calls itself for the two partitions(recursively) and performs in-place sorting while doing so. A separate **partition()** function is used for performing

this in-place sorting at every iteration. Quick sort is one of the most efficient sorting algorithms.

- **Time Complexity: $\theta(n\log(n))$**
- **Space Complexity: $O(\log(n))$**

Working

There are 2 Phases (3 major steps) in the Quick Sort Algorithm

- Division Phase** — Divide the array(list) into 2 halves by finding the pivot point to perform the partition of the array(list).
 - The in-place sorting happens in this partition process itself.
- Recursion Phase**
 - Call Quick Sort on the left partition (sub-list)
 - Call Quick Sort on the right partition (sub-list)

Quick Sort Algorithm(Pseudo Code)

```
QuickSort(arr[], s, e)
{
    if(s < e)
    {
        p = Partition(arr[], s, e)
        QuickSort(arr[], s, (p-1))
        QuickSort(arr[], (p+1), e)
    }
}
```

Quick Sort Partition Function(Pseudo Code)

```
Partition(arr[], s, e)  
{  
    pivot = arr[e]  
    pIndex = s  
    for (i=s to e-1)  
    {  
        if(arr[i]<=pivot)  
        {  
            swap(arr[i], arr[pIndex])  
            pIndex++  
        }  
    }  
    swap(arr[e], arr[pIndex])  
    return pIndex  
}
```

TASK: C++ Program to Implement Quick Sort

Merge Sort Algorithm

Merge Sort Algorithm is a **Divide & Conquer** algorithm. It divides input array in two halves, calls itself for the two halves(recursively) and then merges the two sorted halves. A separate **merge()** function is used for merging two halves. Merge sort is one of the **most efficient sorting algorithms**.

Time Complexity: $O(n\log(n))$

Working

There are 3 Phases (4 major steps) in the Merge Sort Algorithm

- a) Division Phase — Divide the array(list) into 2 halves by finding the midpoint of the array(list).
 - $\text{Midpoint (m)} = (\text{left} + \text{right}) / 2$
 - *Here left is the starting index & right is the last index of the array(list)*
- b) Recursion Phase
 - Call Merge Sort on the left sub-array (sub-list)
 - Call Merge Sort on the right sub-array (sub-list)
- c) Merge Phase
 - Call merge function to merge the divided sub-arrays back to the original array.

- Perform sorting of these smaller sub arrays before merging them back.

Merge Sort Algorithm(Pseudo Code)

```
mergeSort(arr[],l,r) //arr is array, l is left, r is right
```

```
{
```

```
    if(l<r)
```

```
    {
```

```
        midpoint = (l+r)/2
```

```
        mergeSort(arr,l,m)
```

```
        mergeSort(arr,m+1,r)
```

```
        merge(arr,l,m,r)
```

```
    }
```

```
}
```

```
void merge(int a[], int low, int mid, int high)
```

```
{
```

```
    int i = low;
```

```
    int j = mid+1;
```

```
    int k = low;
```

```
    while(i<=mid && j<=high)
```

```
    {
```

```
        if(a[i] < a[j])
```

```
        {
```

```
            b[k] = a[i];
```

```
            i++,k++;
```

```
        }else{
```

```
            b[k] = a[j];
```

```
            j++,k++;
```

```
        }
```

```
    }
```

```
    if(i>mid)
```

```
    {
```

```
        while(j<=high)
```

```
        {
```

```
            b[k] = arr[j];
```

```
            j++,k++;
```

```
        }
```

```
    }else{  
        while(i<=mid)  
        {  
            b[k] = a[i];  
            i++,k++;  
        }  
    }  
  
    for(int p = low; p<=high; p++)  
    {  
        a[p] = b[p];  
    }  
  
}
```

TASK: C++ Program to Implement Merge Sort.

HEAP SORTING

Heap Data Structure:

A **heap** is a specialized tree-based data structure.

It's a complete binary tree where every level is fully filled except possibly the last, which is filled from left to right.

There are two types of heaps:

- **Max-Heap**: In this, the parent node is always greater than or equal to its child nodes.
- **Min-Heap**: In this, the parent node is always smaller than or equal to its child nodes.

The most common implementation of heaps is in the form of an **array**, where:

- For any element at index i , the parent is at index $(i-1)/2$.
- The left child is at index $2*i + 1$ and the right child at $2*i + 2$.

Heap Sort:

Heap Sort is a comparison-based sorting algorithm that uses a binary heap data structure to sort elements.

Max-Heap is generally used for sorting in ascending order. The largest element is placed at the root of the heap, which is then swapped with the last element and removed. This process is repeated for the remaining elements.

The time complexity of Heap Sort is **$O(n \log n)$** in all cases (best, average, and worst).

Heap Sort Algorithm and Pseudo Code

Algorithm Steps:

- **Build a Max-Heap** from the input data.
- **Swap** the root of the max-heap (largest value) with the last element in the array.
- **Reduce the heap size** by one and **heapify** the root element to maintain the max-heap property.
- **Repeat** the process until the entire array is sorted

function heapSort(arr):

 n = length of arr

 # Step 1: Build the Max-Heap

 for i = n/2 - 1 down to 0:

 heapify(arr, n, i)

 # Step 2: Extract elements from heap one by one

 for i = n-1 down to 1:

 swap(arr[0], arr[i]) # Swap current root to the
end

 heapify(arr, i, 0) # Heapify the reduced heap

```

function heapify(arr, n, i):

    largest = i      # Initialize largest as root

    left = 2*i + 1   # left child
    right = 2*i + 2  # right child


    # If left child is larger than root
    if left < n and arr[left] > arr[largest]:
        largest = left


    # If right child is larger than largest so far
    if right < n and arr[right] > arr[largest]:
        largest = right


    # If largest is not root
    if largest != i:
        swap(arr[i], arr[largest]) # Swap root with
largest

        heapify(arr, n, largest)  # Heapify the affected
sub-tree

```

