

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.

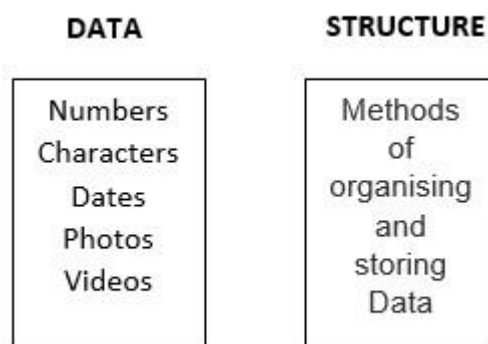
Data Structures are used to organise and store data to use it in an effective way when performing data operations.

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

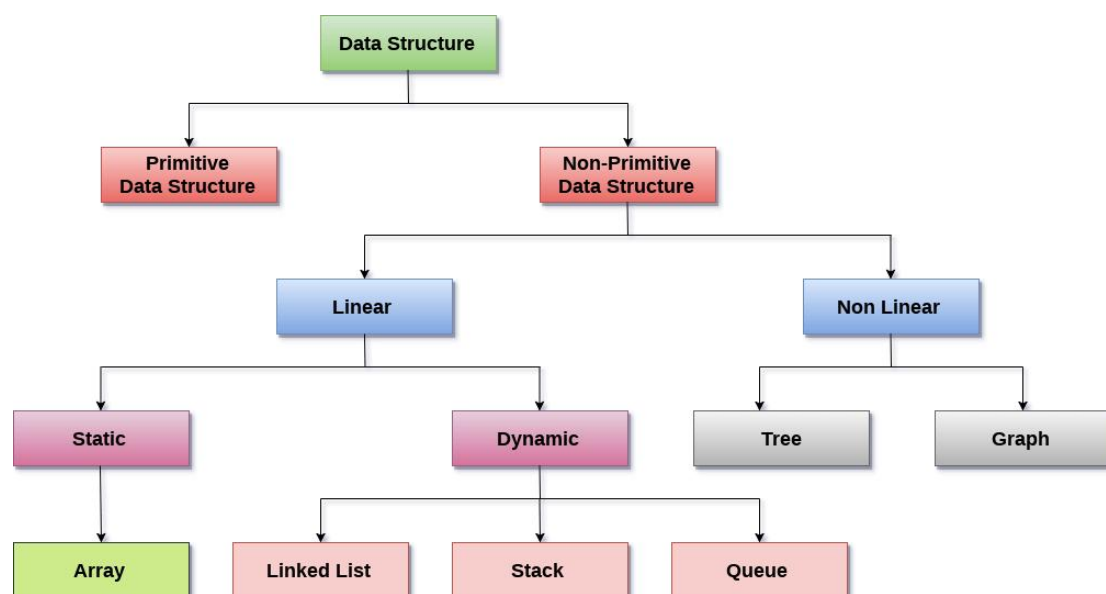
it is essential to choose the appropriate data structure for each tasks. the factor to consider when selecting a proper data structure include the following:-

- 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:-

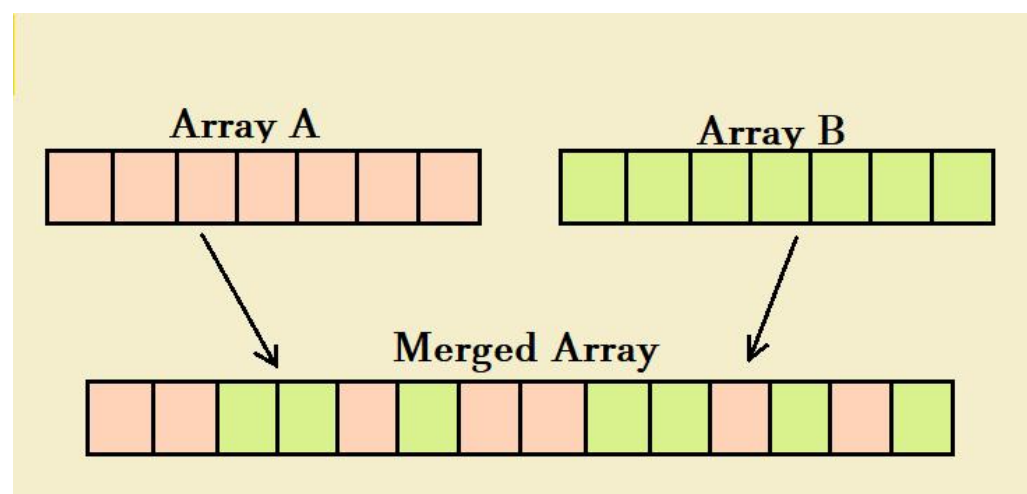
- Easier to comprehend: it is easier to define and use static data structure than the dynamic one. This is because the size and structure of static data structures are defined at compile time and do not change during the program execution.
- size: has a fixed size.
- Faster: the memory for static data 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.

- 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.
- 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.
- slow:** some algorithms are slower when done statically than when done dynamically.

for example:- how would you merge two lists?

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



- 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.

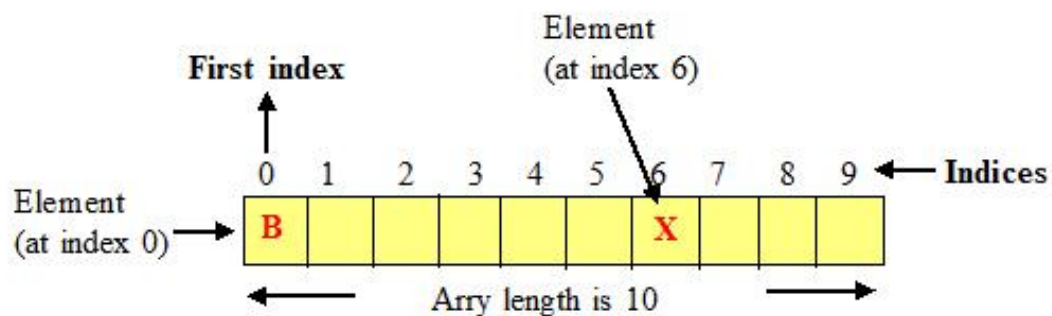
Arrays must be classified before being used in the process. they are classified into two dimensions as follows:-

- 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

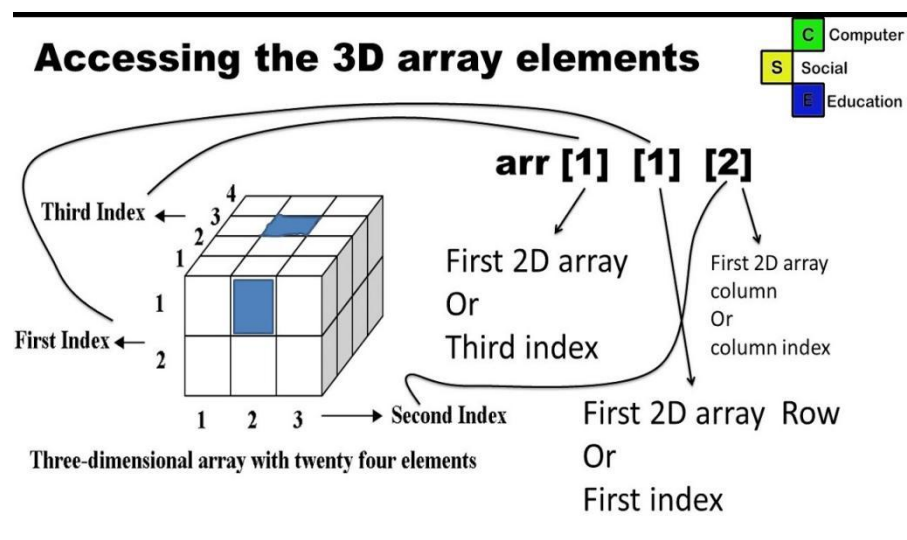


fig. 3-d arrays.

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).

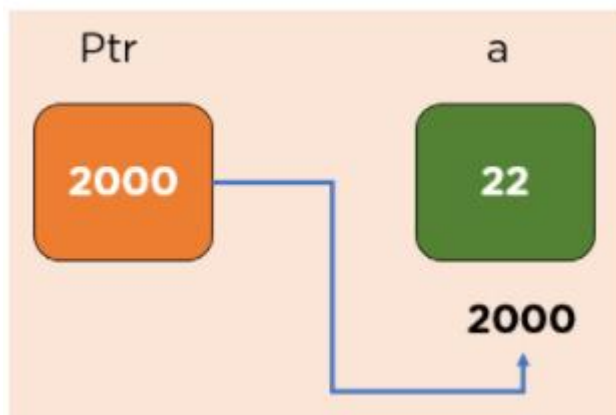


fig above. the pointer and it's variable

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.

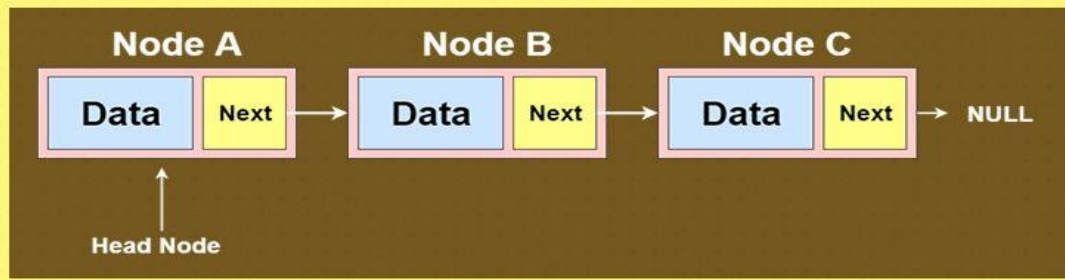
LINKED LIST.

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:-

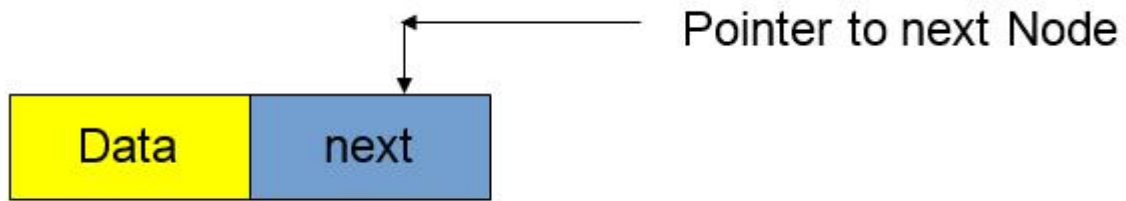
- Insertion and deletion operations can be implemented very easily and these are not costly as they do not require shifting of elements.
- They are dynamic in nature. Hence, we can change their size whenever required.
- Stacks and queues can be implemented very easily using Linked Lists.

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

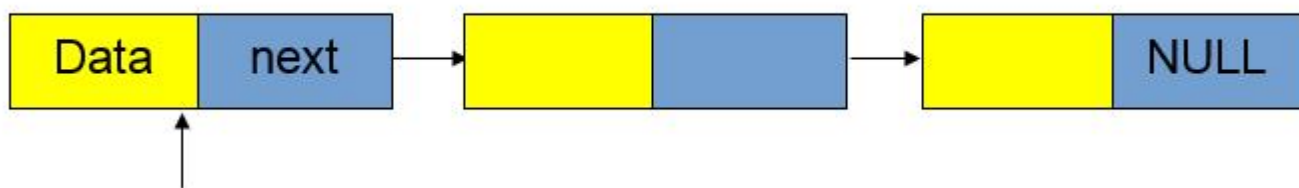


There are three types of Linked Lists:

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

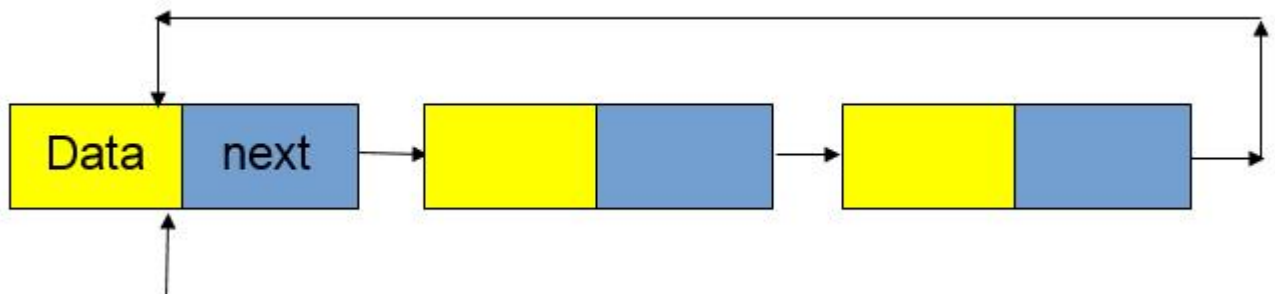
Singly Linked List

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

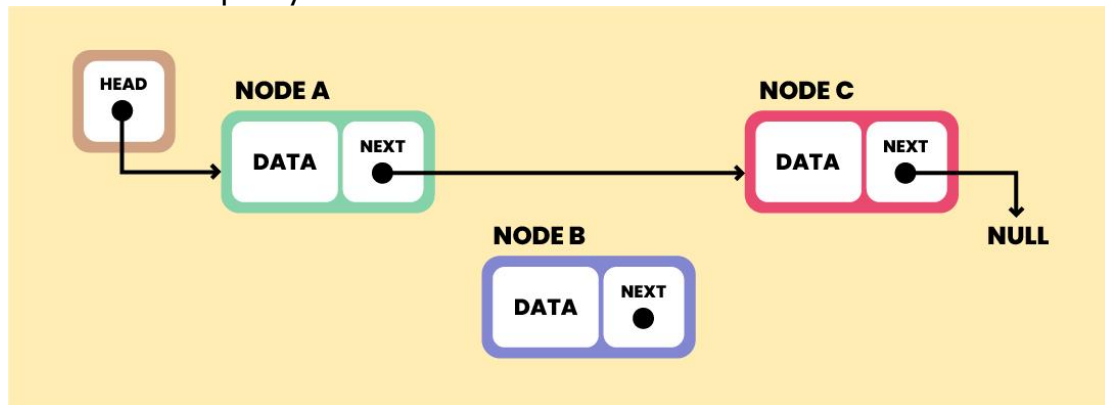
Following are the various operations to perform the required action on a linked list:

- 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

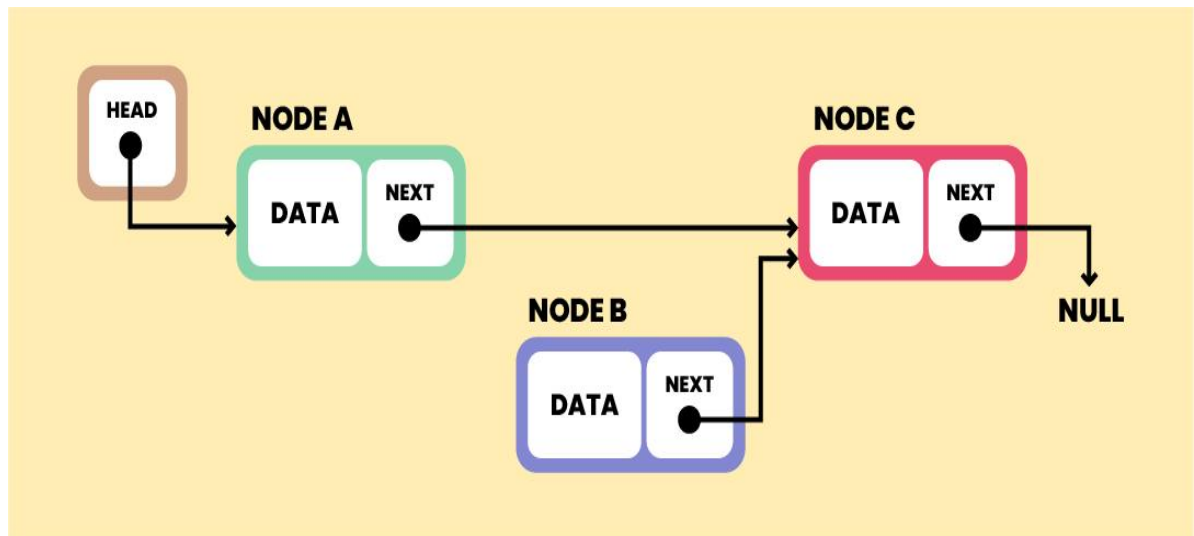
It's more than a one-step activity. First, we create a node with the same structure and specify the location where we'll insert it.



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

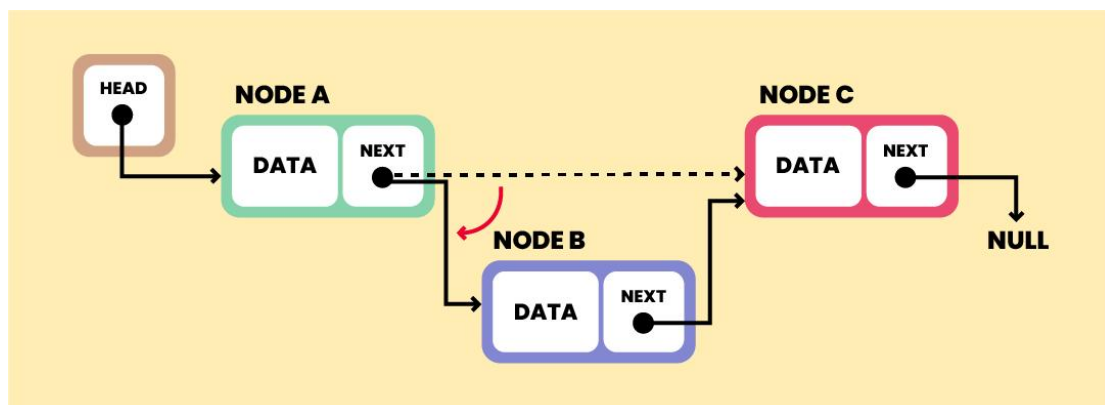
The new Node B will point to Node C.

NodeB.next - Node C

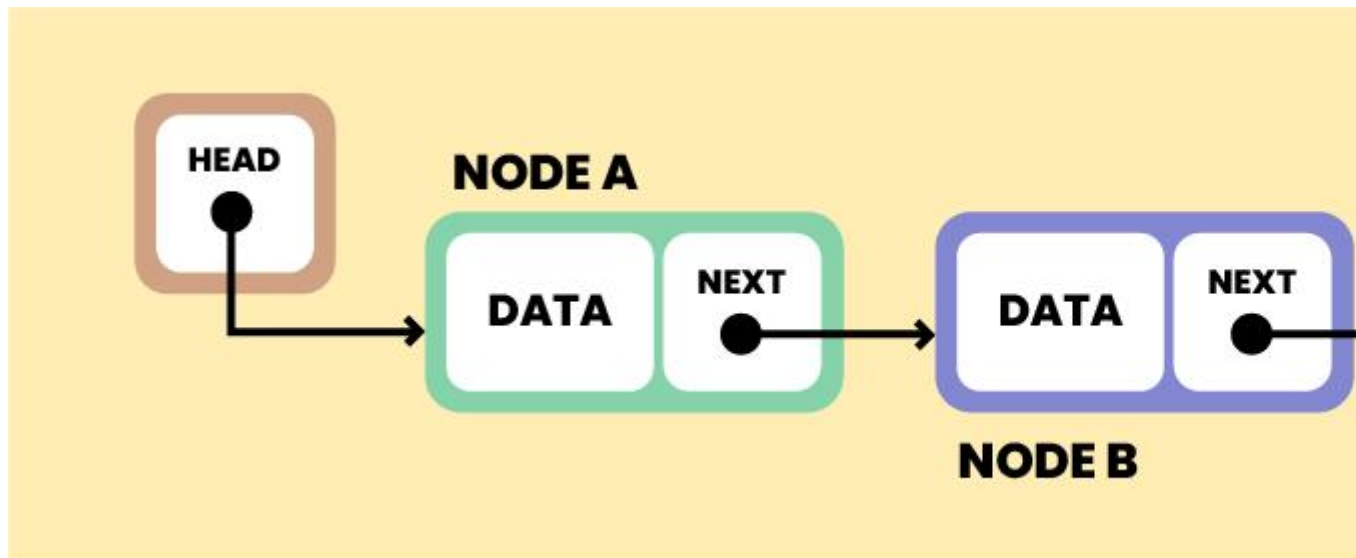


Now, Node A should point to Node B

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'.

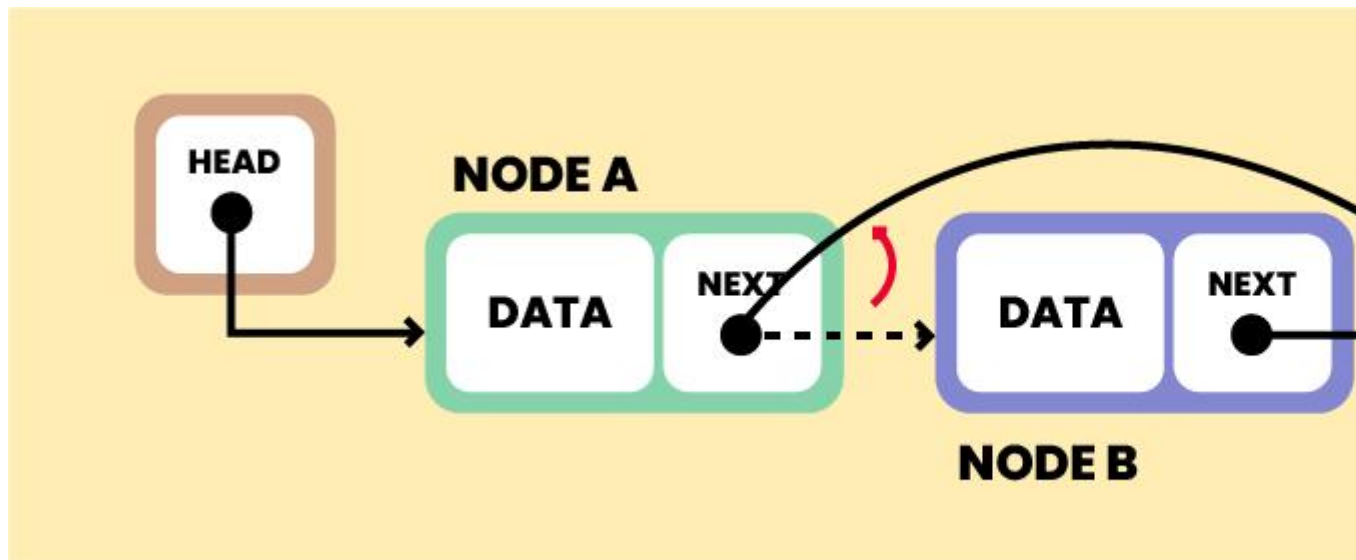
Deletion

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.

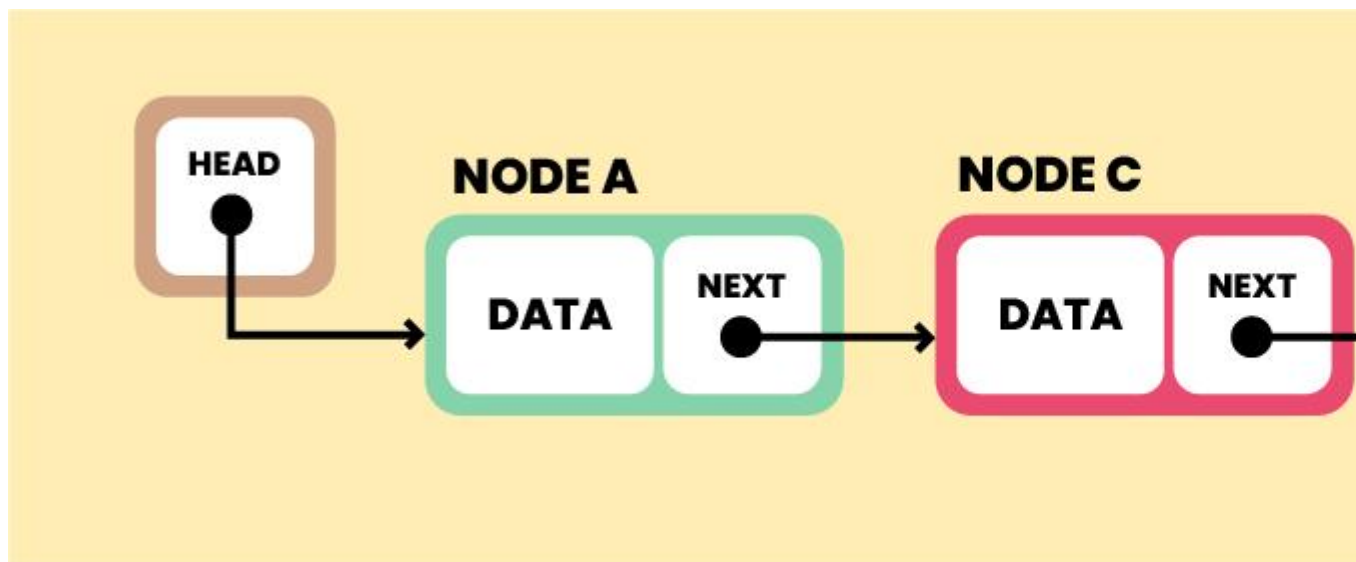


NodeA.next - NodeC

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?

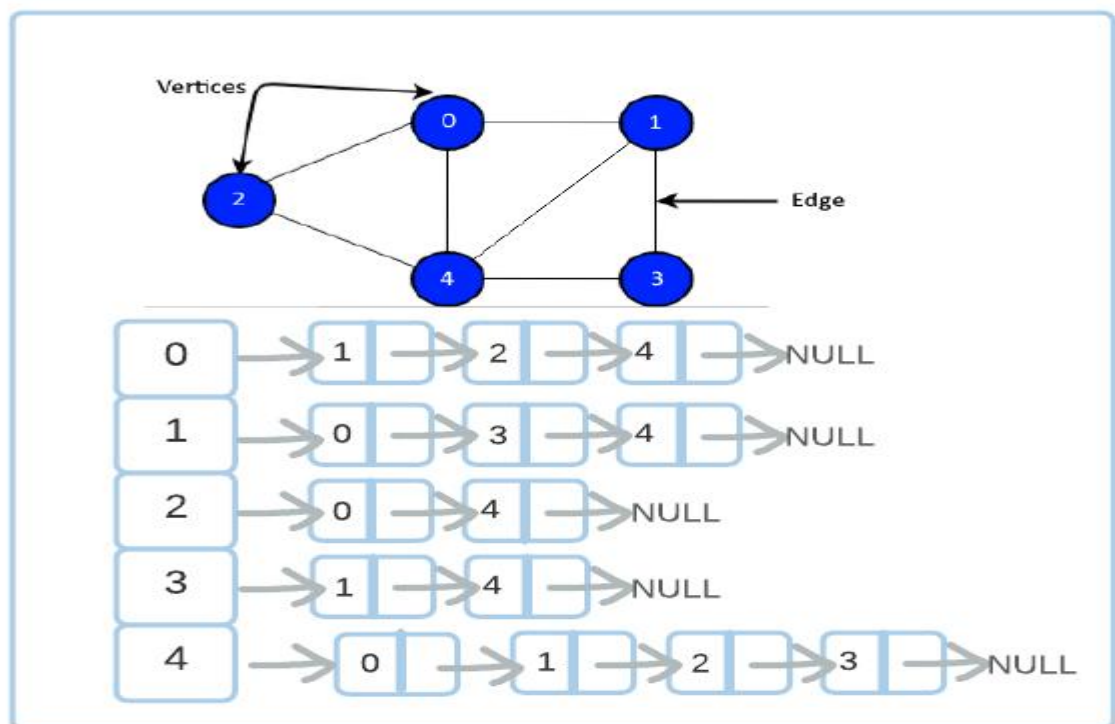
As a stack follows the Last In First Out principle, every PUSH operation will be INSERT_IN_BEGINNING and every POP operation will be DELETE_IN_BEGINNING in the linked list.

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.



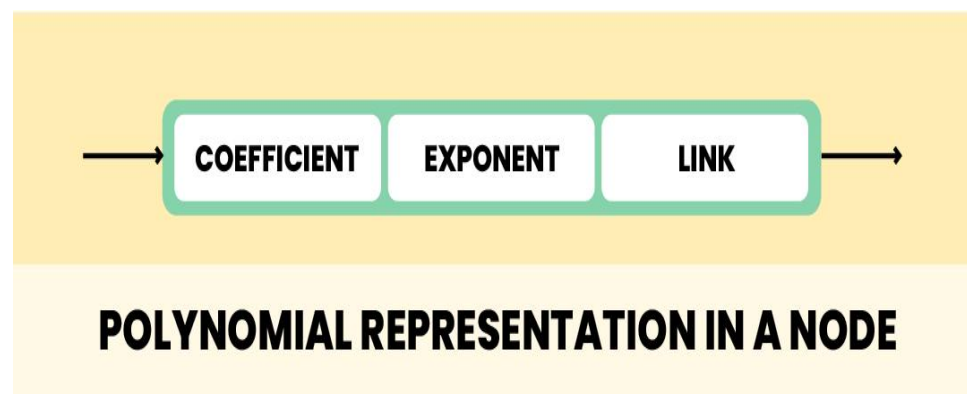
Polynomial manipulation

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?

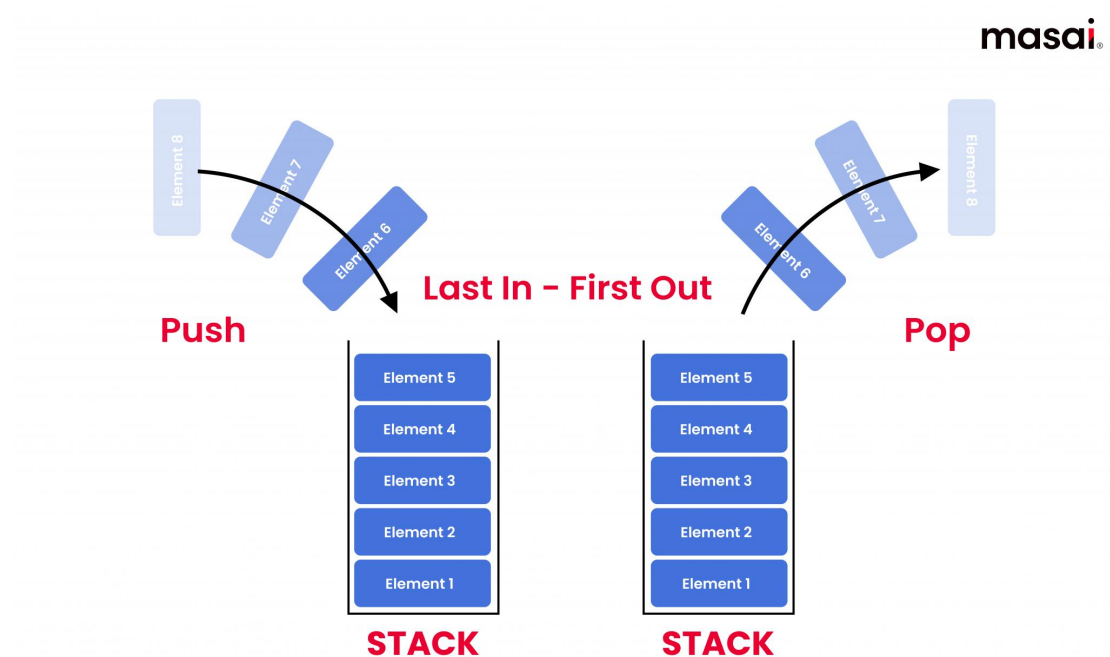
ADTs specify the data and its applications but is independent of how it's implemented. ADTs are implemented using different methods and logic in different programming languages.

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.

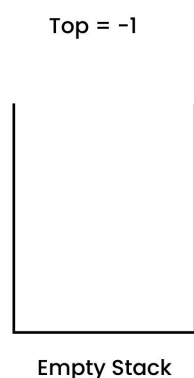
peek() – This operation allows us to view the element which is at the top. There are no changes made in the stack here.

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'

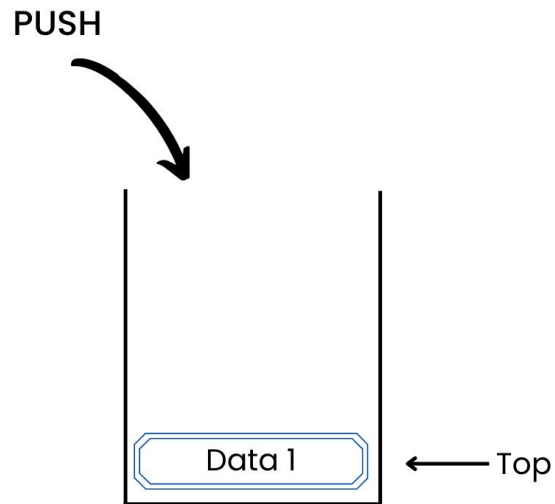
isFull() – It's the exact opposite of isEmpty operation, as it returns 'true' if the stack is full, else returns 'false'.

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

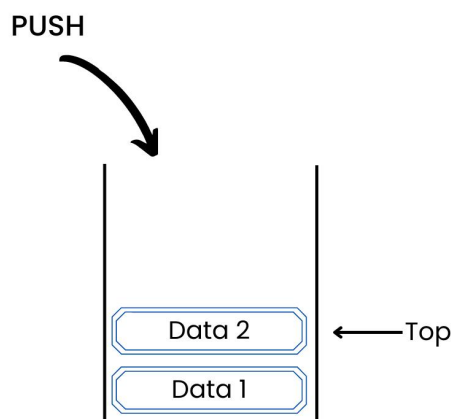
1) While initializing the stack, we set the value of the pointer 'top' = -1. This value means that the stack is empty.



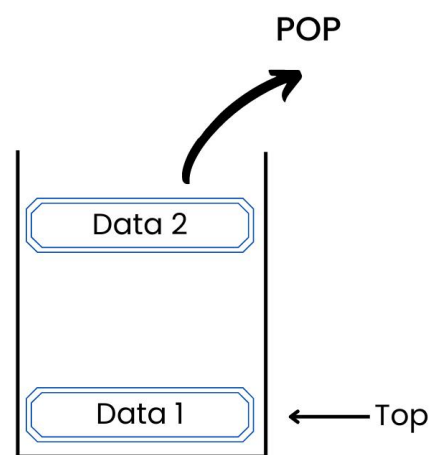
2) When we add/push an element, the value increases to 'top' = 0, and the new element is placed in the position corresponding to the 'top' pointer.



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.



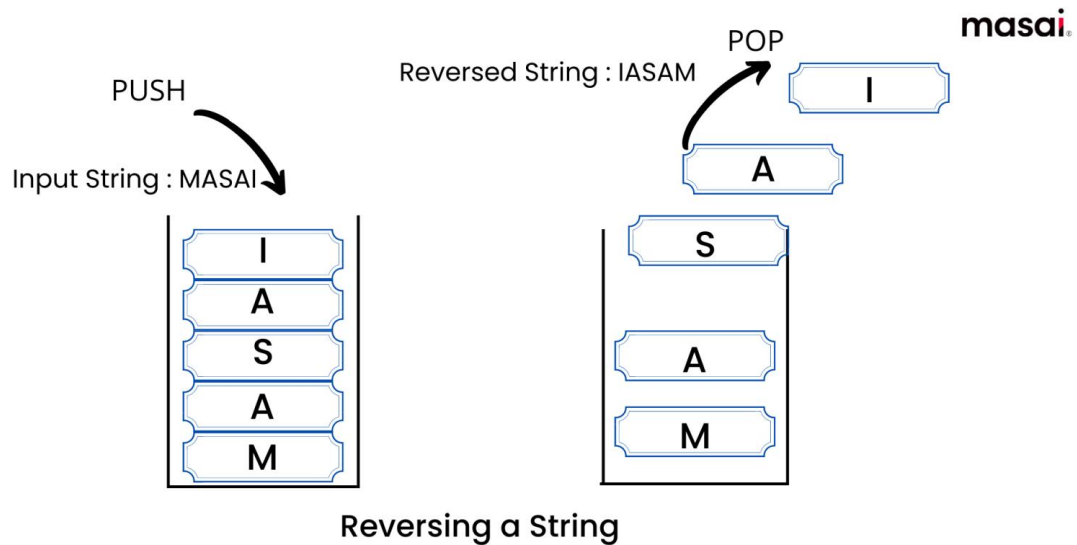
- We check if the stack is already full before pushing.
- We check if the stack is empty before implementing the pop operation.

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.