Data Structure and Types

What are Data Structures?

Data structure is a storage that is used to store and organize data. It is a way of arranging data on a computer so that it can be accessed and updated efficiently.

Depending on your requirement and project, it is important to choose the right data structure for your project. For example, if you want to store data sequentially in the memory, then you can go for the Array data structure.

Array data Structure Representation

> **Note: Data structure and data types are slightly different. Data structure is the collection of data types arranged in a specific order.**

Types of Data Structure

Basically, data structures are divided into two categories:

- **Linear data structure**
- **Non-linear data structure**

Linear data structures

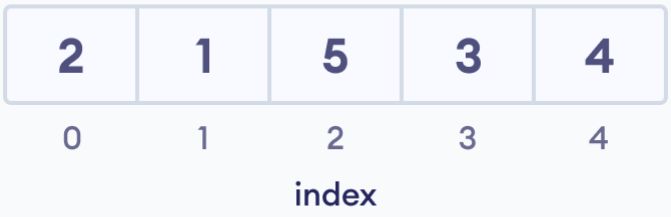
In linear data structures, the elements are arranged in sequence one after the other. Since elements are arranged in particular order, they are easy to implement.

However, when the complexity of the program increases, the linear data structures might not be the best choice because of operational complexities.

Popular linear data structures are:

1. Array Data Structure

In an array, elements in memory are arranged in continuous memory. All the elements of an array are of the same type. And, the type of elements that can be stored in the form of arrays is determined by the programming language.

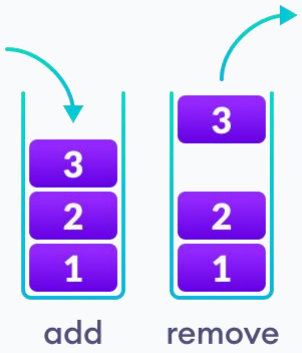


An array with each element represented by an index

2. Stack Data Structure

In stack data structure, elements are stored in the LIFO principle. That is, the last element stored in a stack will be removed first.

It works just like a pile of plates where the last plate kept on the pile will be removed first.



In a stack, operations can be perform only from one end (top here).

3. Queue Data Structure

Unlike stack, the queue data structure works in the FIFO principle where first element stored in the queue will be removed first.

It works just like a queue of people in the ticket counter where first person on the queue will get the ticket first.

In a queue, addition and removal are performed from separate ends.

4. Linked List Data Structure

In linked list data structure, data elements are connected through a series of nodes. And, each node contains the data items and address to the next node.

A linked list

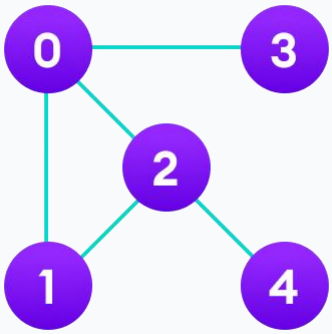
Non linear data structures

Unlike linear data structures, elements in non-linear data structures are not in any sequence. Instead they are arranged in a hierarchical manner where one element will be connected to one or more elements.

Non-linear data structures are further divided into graph and tree based data structures.

1. Graph Data Structure

In graph data structure, each node is called vertex and each vertex is connected to other vertices through edges.



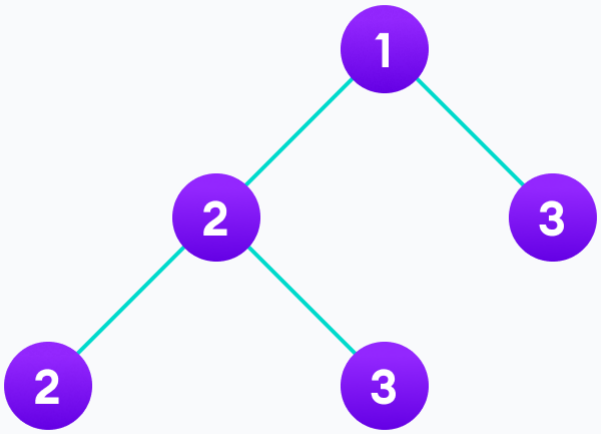
Graph data structure example

Popular Graph Based Data Structures:

- Spanning Tree and Minimum Spanning Tree
- Strongly Connected Components
- Adjacency Matrix
- Adjacency List

2. Trees Data Structure

Similar to a graph, a tree is also a collection of vertices and edges. However, in tree data structure, there can only be one edge between two vertices.



Tree data structure example

Popular Tree based Data Structure

- Binary Tree
- Binary Search Tree
- AVL Tree
- B-Tree
- B+ Tree
- Red-Black Tree

Linear Vs Non-linear Data Structures

Now that we know about linear and non-linear data structures, let's see the major differences between them.

Linear Data Structures	Non Linear Data Structures
The data items are arranged in sequential order, one after the other.	The data items are arranged in non-sequential order (hierarchical manner).

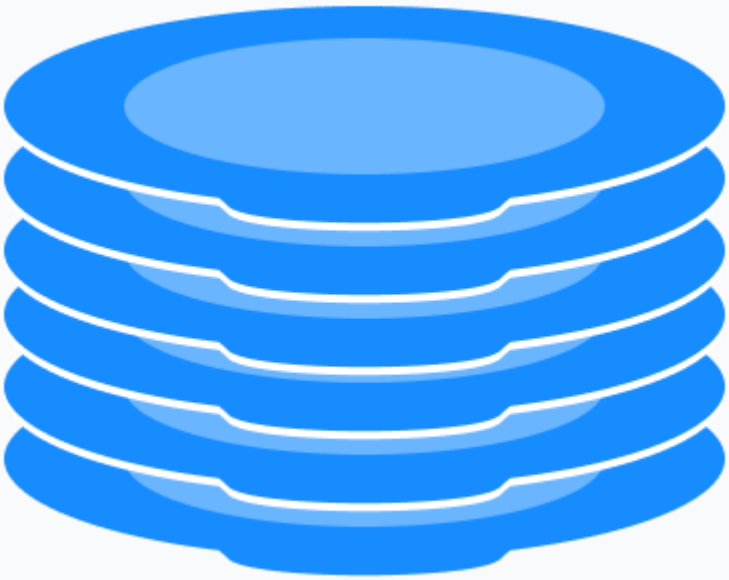
All the items are present on the single layer.	**The data items are present at different layers.**
It can be traversed on a single run. That is, if we start from the first element, we can traverse all the elements sequentially in a single pass.	**It requires multiple runs. That is, if we start from the first element it might not be possible to traverse all the elements in a single pass.**
The memory utilization is not efficient.	**Different structures utilize memory in different efficient ways depending on the need.**
The time complexity increase with the data size.	**Time complexity remains the same.**
Example: Arrays, Stack, Queue	**Example: Tree, Graph, Map**

Why Data Structure?

Knowledge about data structures help you understand the working of each data structure. And, based on that you can select the right data structures for your project.

This helps you write memory and time efficient code.

A stack is a linear data structure that follows the principle of Last In First Out (LIFO). This means the last element inserted inside the stack is removed first.
You can think of the stack data structure as the pile of plates on top of another.



Stack representation similar to a pile of plate

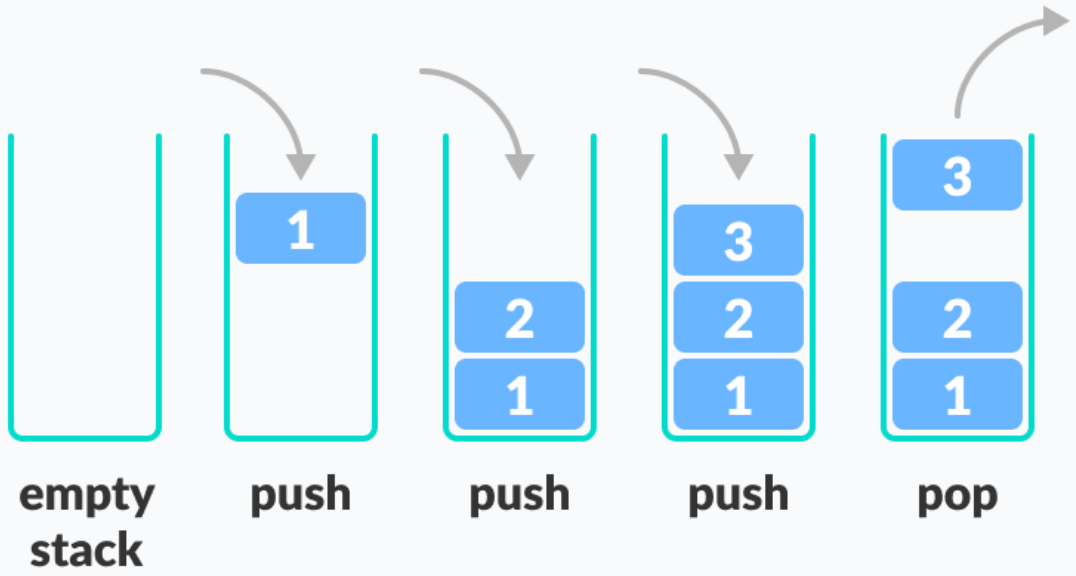
Here, you can:

- **Put a new plate on top**
- **Remove the top plate**

And, if you want the plate at the bottom, you must first remove all the plates on top. This is exactly how the stack data structure works.

LIFO Principle of Stack

In programming terms, putting an item on top of the stack is called push and removing an item is called pop.



Stack Push and Pop Operations

In the above image, although item 3 was kept last, it was removed first. This is exactly how the LIFO (Last In First Out) Principle works. We can implement a stack in any programming language like C, C++, Java, Python or C#, but the specification is pretty much the same.

Basic Operations of Stack

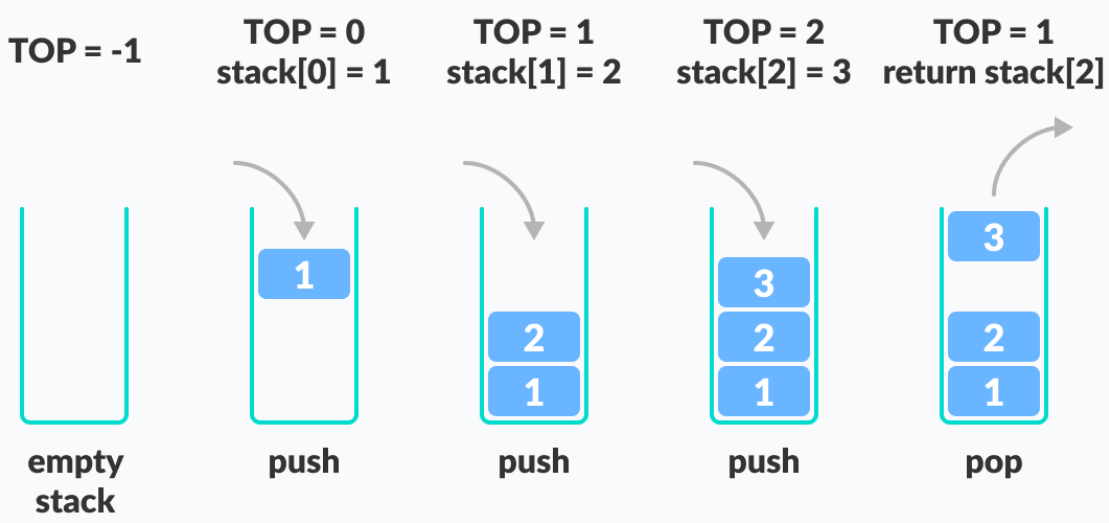
There are some basic operations that allow us to perform different actions on a stack.

- **Push: Add an element to the top of a stack**
- **Pop: Remove an element from the top of a stack**
- **IsEmpty: Check if the stack is empty**
- **IsFull: Check if the stack is full**
- **Peek: Get the value of the top element without removing it**

Working of Stack Data Structure

The operations work as follows:

1. A pointer called TOP is used to keep track of the top element in the stack.
2. When initializing the stack, we set its value to -1 so that we can check if the stack is empty by comparing TOP == -1.
3. On pushing an element, we increase the value of TOP and place the new element in the position pointed to by TOP.
4. On popping an element, we return the element pointed to by TOP and reduce its value.
5. Before pushing, we check if the stack is already full
6. Before popping, we check if the stack is already empty



Working of Stack Data Structure

Applications of Stack Data Structure

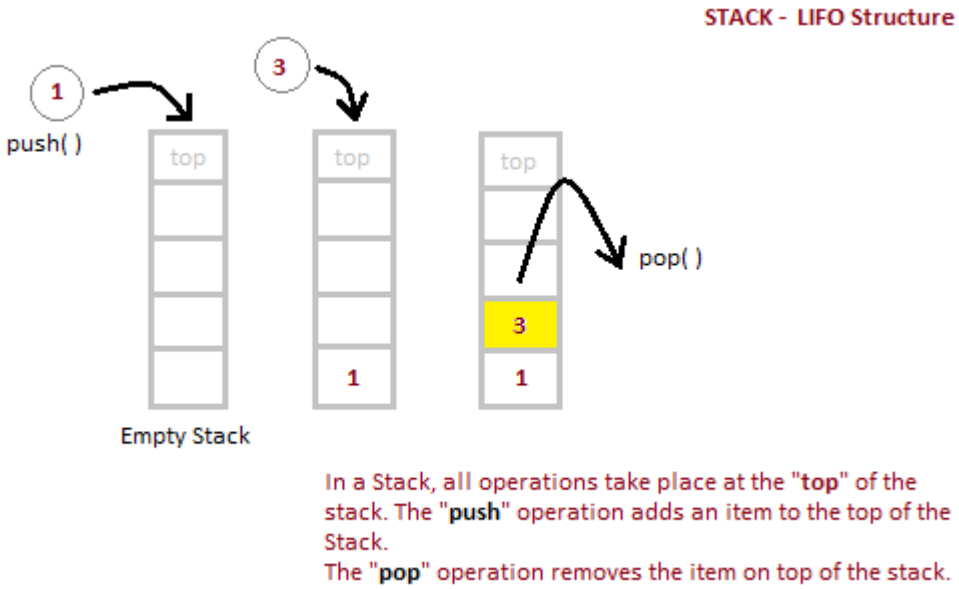
Although stack is a simple data structure to implement, it is very powerful. The most common uses of a stack are:

- To reverse a word - Put all the letters in a stack and pop them out. Because of the LIFO order of stack, you will get the letters in reverse order.

- **In compilers - Compilers use the stack to calculate the value of expressions like 2 + 4 / 5 * (7 - 9) by converting the expression to prefix or postfix form.**
- **In browsers - The back button in a browser saves all the URLs you have visited previously in a stack. Each time you visit a new page, it is added on top of the stack. When you press the back button, the current URL is removed from the stack, and the previous URL is accessed.**

Implementation of Stack Data Structure

Stack can be easily implemented using an Array or a Linked List. Arrays are quick, but are limited in size and Linked List requires overhead to allocate, link, unlink, and deallocate, but is not limited in size. Here we will implement Stack using array.



Algorithm for PUSH operation

1. **Check if the stack is full or not.**
2. **If the stack is full, then print error of overflow and exit the program.**
3. **If the stack is not full, then increment the top and add the element.**

Algorithm for POP operation

1. **Check if the stack is empty or not.**
2. **If the stack is empty, then print error of underflow and exit the program.**
3. **If the stack is not empty, then print the element at the top and decrement the top.**

Stack Operations using Array

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

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

- **Step 1 - Include all the header files which are used in the program and define a constant 'SIZE' with specific value.**
- **Step 2 - Declare all the functions used in stack implementation.**
- **Step 3 - Create a one dimensional array with fixed size (int stack[SIZE])**
- **Step 4 - Define a integer variable 'top' and initialize with '-1'. (int top = -1)**
- **Step 5 - In main method, display menu with list of operations and make suitable function calls to perform operation selected by the user on the stack.**

push(value) - Inserting value into the stack

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

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

pop() - Delete a value from the Stack

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

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

display() - Displays the elements of a Stack

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

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

Implementation of Stack using Array

```
#include<stdio.h>
#include<conio.h>
```

```
#define SIZE 10

void push(int);
void pop();
void display();

int stack[SIZE], top = -1;

void main()
{
   int value, choice;
     while(1){
      printf("\n\n***** MENU *****\n");
      printf("1. Push\n2. Pop\n3. Display\n4. Exit");
      printf("\nEnter your choice: ");
      scanf("%d",&choice);
      switch(choice){
         case 1: printf("Enter the value to be insert: ");
                 scanf("%d",&value);
                 push(value);
                 break;
         case 2: pop();
                 break;
         case 3: display();
                 break;
         case 4: exit(0);
         default: printf("\nWrong selection!!! Try again!!!");
      }
   }
}
void push(int value){
   if(top == SIZE-1)
      printf("\nStack is Full!!! Insertion is not possible!!!");
   else{
      top++;
```

```
      stack[top] = value;
      printf("\nInsertion success!!!");
   }
}
void pop(){
   if(top == -1)
      printf("\nStack is Empty!!! Deletion is not possible!!!");
   else{
      printf("\nDeleted : %d", stack[top]);
      top--;
   }
}
void display(){
   if(top == -1)
      printf("\nStack is Empty!!!");
   else{
      int i;
      printf("\nStack elements are:\n");
      for(i=top; i>=0; i--)
         printf("%d\n",stack[i]);
   }
}
```