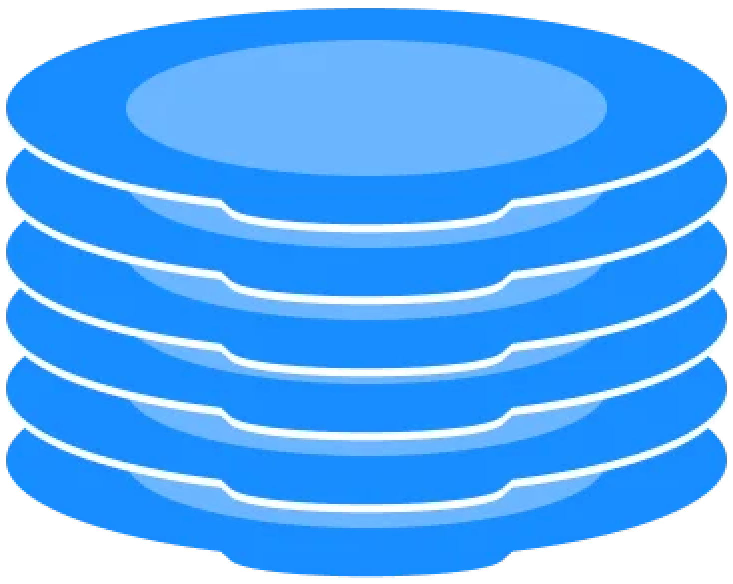
A stack is a linear data structure that follows the principle of Last In First Out (LIFO). This means the last element inserted inside the stack is removed first.

You can think of the stack data structure as the pile of plates on top of another.

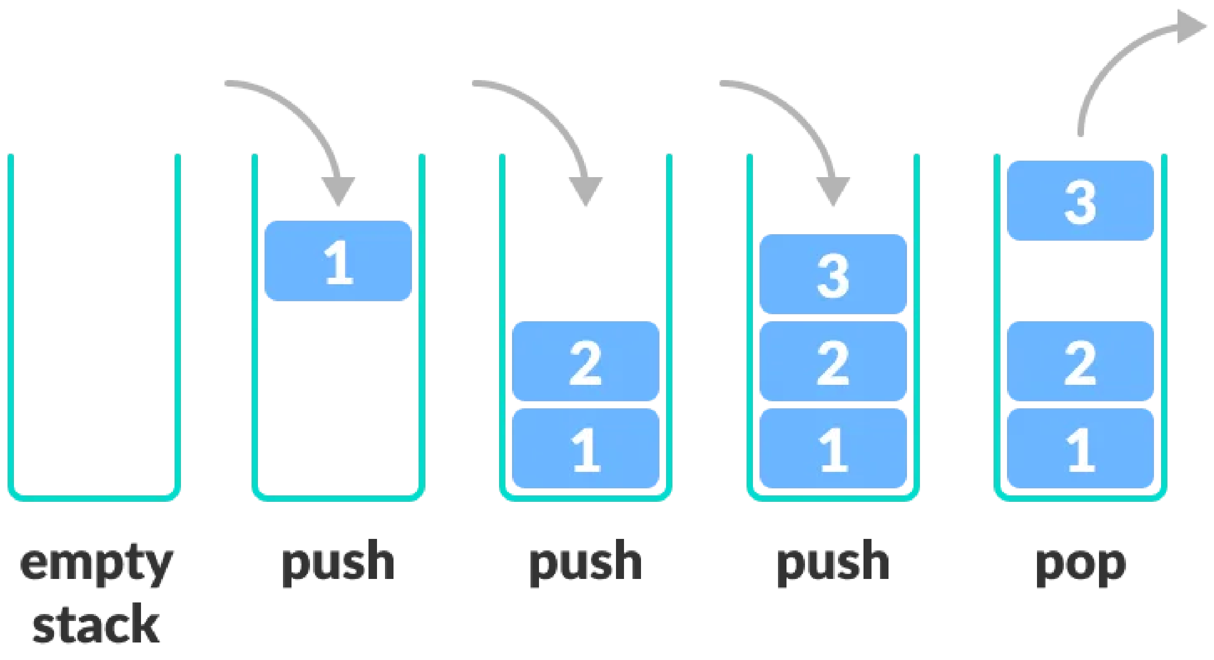


Here, you can:

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.



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.

Basic Operations of Stack

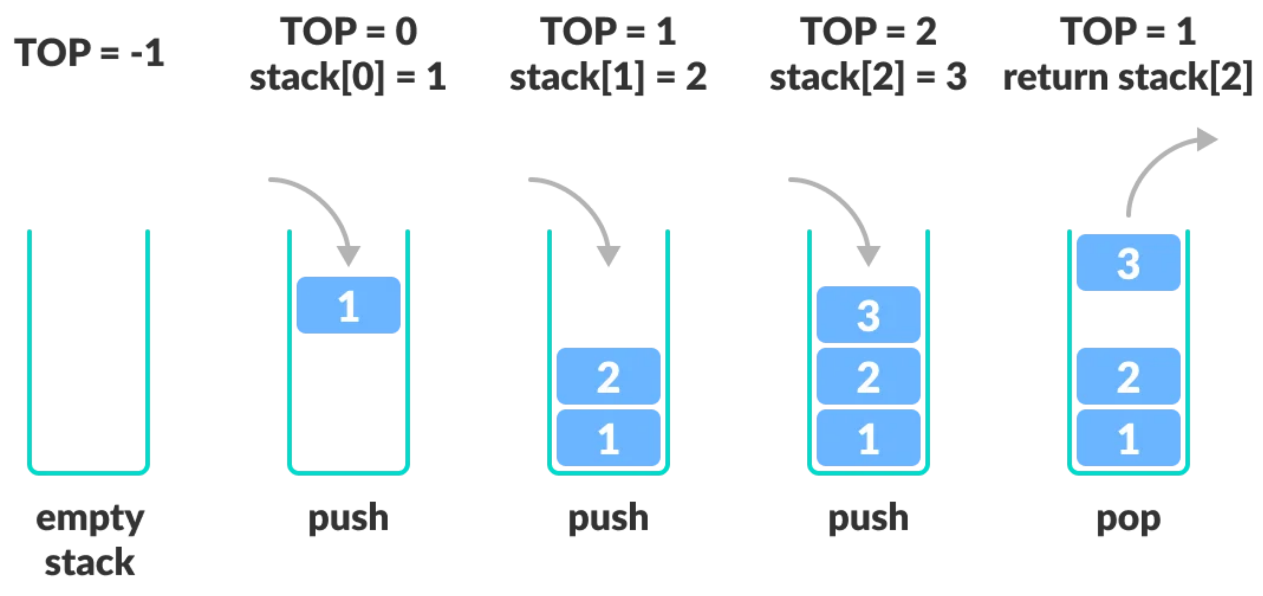
There are some basic operations that allow us to perform different actions on a stack.

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:

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



Stack Implementations in Python, Java, C, and C++

The most common stack implementation is using arrays, but it can also be implemented using lists.

Stack implementation in Java

```
class Stack {
  private int arr[];
  private int top;
  private int capacity;

  // Creating a stack
  Stack(int size) {
    arr = new int[size];
    capacity = size;
    top = -1;
  }

  // Add elements into stack
  public void push(int x) {
    if (isFull()) {
      System.out.println("OverFlow\nProgram Terminated\n");
      System.exit(1);
    }

    System.out.println("Inserting " + x);
    arr[++top] = x;
  }

  // Remove element from stack
  public int pop() {
    if (isEmpty()) {
      System.out.println("STACK EMPTY");
      System.exit(1);
    }
    return arr[top--];
  }

  // Utility function to return the size of the stack
  public int size() {
    return top + 1;
  }

  // Check if the stack is empty
  public Boolean isEmpty() {
    return top == -1;
  }

  // Check if the stack is full
  public Boolean isFull() {
    return top == capacity - 1;
  }

  public void printStack() {
    for (int i = 0; i <= top; i++) {
      System.out.println(arr[i]);
    }
  }

  public static void main(String[] args) {
    Stack stack = new Stack(5);

    stack.push(1);
    stack.push(2);
    stack.push(3);
    stack.push(4);

    stack.pop();
    System.out.println("\nAfter popping out");

    stack.printStack();

  }
}
```

Stack implementation in C++

```
#include <stdlib.h>
#include <iostream>

using namespace std;

#define MAX 10
```

```
int size = 0;

// Creating a stack
struct stack {
  int items[MAX];
  int top;
};
typedef struct stack st;

void createEmptyStack(st *s) {
  s->top = -1;
}

// Check if the stack is full
int isfull(st *s) {
  if (s->top == MAX - 1)
    return 1;
  else
    return 0;
}

// Check if the stack is empty
int isempty(st *s) {
  if (s->top == -1)
    return 1;
  else
    return 0;
}

// Add elements into stack
void push(st *s, int newitem) {
  if (isfull(s)) {
    cout << "STACK FULL";
  } else {
    s->top++;
    s->items[s->top] = newitem;
  }
  size++;
}

// Remove element from stack
void pop(st *s) {
  if (isempty(s)) {
    cout << "\n STACK EMPTY \n";
  } else {
    cout << "Item popped= " << s->items[s->top];
    s->top--;
  }
  size--;
  cout << endl;
}

// Print elements of stack
void printStack(st *s) {
  printf("Stack: ");
  for (int i = 0; i < size; i++) {
    cout << s->items[i] << " ";
  }
  cout << endl;
}

// Driver code
int main() {
  int ch;
  st *s = (st *)malloc(sizeof(st));

  createEmptyStack(s);

  push(s, 1);
  push(s, 2);
  push(s, 3);
  push(s, 4);

  printStack(s);

  pop(s);

  cout << "\nAfter popping out\n";
  printStack(s);
}
```

Stack Time Complexity

For the array-based implementation of a stack, the push and pop operations take constant time, i.e. O(1).

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.