

## **-Presentación**

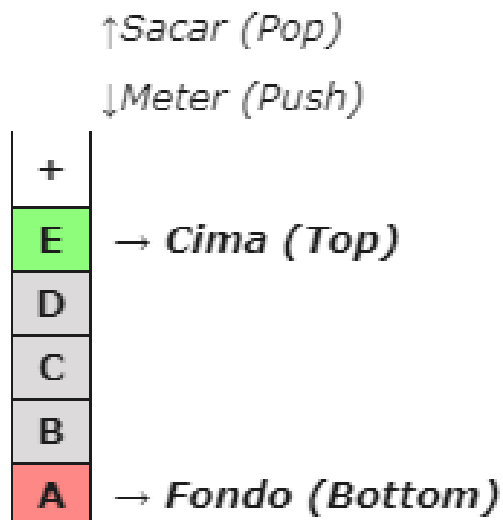
En términos muy generales, el área de estudio de estructuras de datos como disciplina de las ciencias computacionales (CS) es el almacenamiento y la manipulación de la información. Las estructuras de datos, siendo una disciplina dentro de la categoría de la computación teórica, estudian de manera más abstracta la relación entre una colección de datos y las operaciones que pueden ser aplicadas a dicha colección. A continuación, se hablará acerca de una de las estructuras de datos fundamentales en el manejo de colecciones: las pilas (stacks).

## **-Introducción**

Pero, ¿qué es una pila?

En programación, una pila es una estructura de datos dinámica formada por un grupo ordenado de elementos homogéneos (todos del mismo tipo) en la que los elementos sólo pueden ser añadidos o eliminados por un extremo llamado cabecera o cima (top) de la pila.

De esta forma, los elementos se extraen en orden inverso al seguido en el proceso de inserción. Por ello, una pila es considerada una estructura de datos LIFO (Last In First Out), esto es, que el último elemento que entra es el primero que sale.



## -Desarrollo

Las pilas pueden ser entendidas con la analogía de una montaña de objetos, por ejemplo una montaña de hotcakes, donde agregar un elemento significa agregar un nuevo hotcake encima del último (el de la cima) y sacar un elemento significa quitar el hotcake que está en la cima. La operación de agregar un elemento a una pila se conoce como push, mientras que la operación que *saca* elementos se conoce como pop. Además, cuando tratamos de agregar elementos a una pila llena (porque las definimos como finitas) ocurre un error conocido como stack overflow, mientras que en el caso opuesto, es decir, sacar elementos de una pila vacía, genera un error con un nombre menos popular conocido como stack underflow.



- **Operaciones sobre una pila**

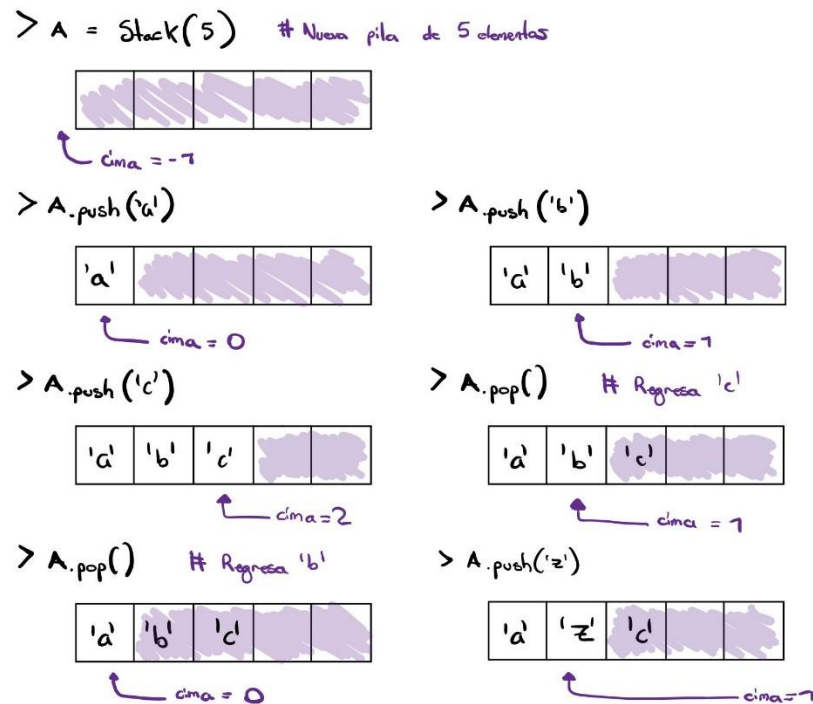
Las operaciones básicas que podemos realizar sobre una pila son dos: almacenamiento (push) y recuperación (pop). Para manejar una estructura de datos tipo pila, un programador debe definir un conjunto de operaciones que permitan al usuario acceder y manipular los elementos en ella almacenados.

Generalmente, cada una de estas operaciones suele recibir un nombre fijo e identificativo de la operación que realiza, siendo la terminología utilizada más común la mostrada a continuación:

- **Meter o apilar:** se denomina así a la operación que añade un elemento a una pila.
- **Sacar o desapilar:** operación que saca un elemento de la pila.
- **Inicializar Pila:** una vez creada la pila y antes de usarla, esta debe quedar inicialmente vacía.
- **Pila vacía:** operación que devuelve un valor de tipo lógico para comprobar si la pila se encuentra vacía antes de sacar un elemento.
- **Pila llena:** la usaremos en aquellos casos en que sea necesario saber si la pila se encuentra llena antes de añadir un nuevo elemento.
- **Cima:** útil para obtener el último elemento que fue añadido a la pila, este es, el primero en salir de ella.

## -Implementación

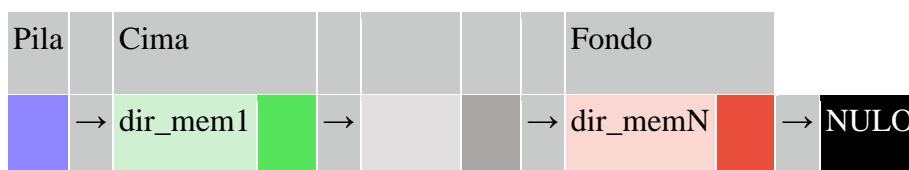
Una pila puede ser fácilmente implementada utilizando arreglos y manteniendo siempre una referencia a la cima, es decir, un índice que indique la posición del último elemento agregado. Cada que un elemento se agrega a la pila incrementamos en 1 dicho índice y agregamos el nuevo elemento al arreglo en la posición que la cima indica. En el caso de un pop realizamos lo opuesto: regresamos el elemento que se encuentra en la posición de la cima y decrementamos su valor en 1. Se podría también eliminar (volver nulo) el valor en dicha posición antes de regresarlo, pero de todas formas estaremos ocupando la misma cantidad de memoria por lo que la supuesta optimización no tendría sentido.



También pila puede ser implementada de dos formas diferentes:

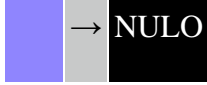
- Usando una tabla unidimensional (estructura de datos estática).
- Mediante el uso de punteros (estructura de datos dinámica).

El primer caso, tabla, es más sencillo de implementar, sin embargo, y dado que una pila es una estructura de datos dinámica, el uso de punteros es más lógico. Gráficamente la implementación de una pila usando punteros sería:



Cuando se crea la pila el puntero tiene valor Nulo, esto es:

Pila vacía



## Ejemplo de Pila en C++

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

```
using namespace std;
```

```
class nodo {
public:
    nodo(char v, nodo *sig = NULL) { //funcion nodo que tiene un dato(char) y un puntero que al inicio apunta a null
        dato = v; //el valor de retorno se guarda en dato
        siguiente = sig; // el valor de sig se guarda en la variable siguiente
    }
private:
    char dato; //los protagonista de la clase el dato y el puntero nodo
    nodo *siguiente; // puntero siguiente de tipo nodo
    friend class pila; // acceso a esta clase desde pila
};
typedef nodo *pnodo; //se define el tipo nodo del puntero
```

```
class pila {
public:
    pila() : ultimo(NULL) {}
    ~pila();

    void Push(char v);
    char Pop();

private:
    pnodo ultimo;
};
```

```
// función de caso particular de pila que verifica si la pila esta vacía
```

```
pila::~~pila() {
    while(ultimo) Pop();
}
```

```
//función de insertar valor
```

```
void pila::Push(char v) {
    pnodo nuevo; // declaracion de nuevo
```

```
nuevo = new nodo(v, ultimo); // Se crea un nuevo nodo
ultimo = nuevo; // el comienzo de esta pila es en nuevo nodo
}
```

```
char pila::Pop() {
    pnodo nodo; // variable que manipula el nodo
```

```

char v;    // variable que retorna el valor

if(!ultimo) return '0';    //si no hay nodos retorna 0
    nodo = ultimo;    // nodo apunta al primer elemento de la pila
    ultimo = nodo->siguiente; //asignamos a pila toda la pila menos el primer elemento
    v = nodo->dato;    //guardamos el valor de retorno
    delete (nodo);    //Se borra el nodo
return v;
}

int main(int argc, char** argv) {
    pila Pila;

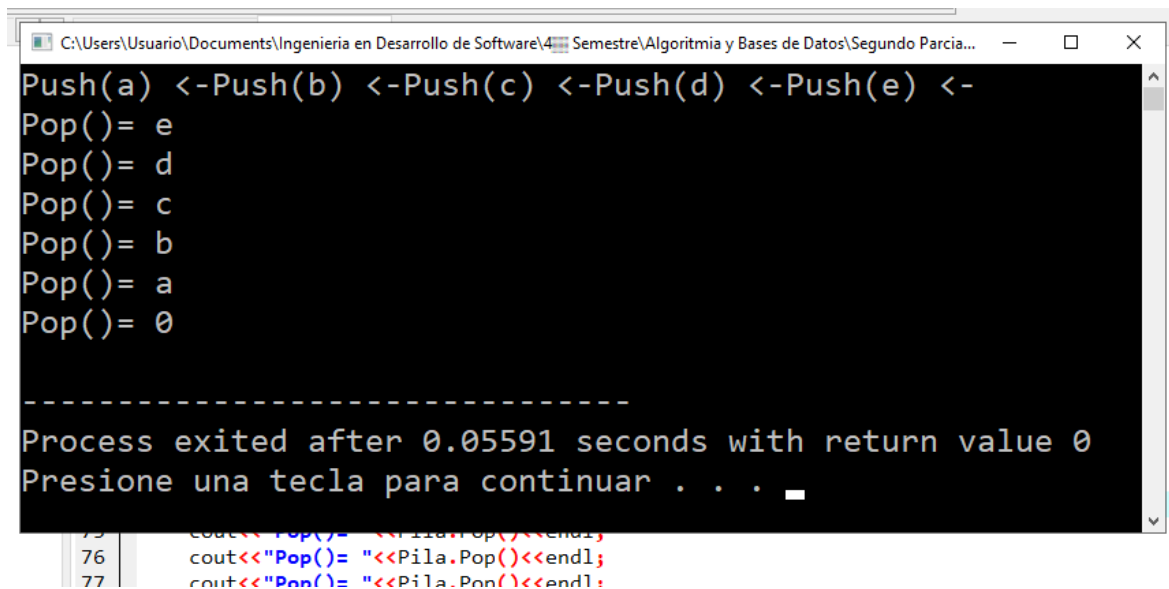
    Pila.Push('a');
    cout<<"Push(a) <-"<<endl;
    Pila.Push('b');
    cout<<"Push(b) <-"<<endl;
    Pila.Push('c');
    cout<<"Push(c) <-"<<endl;
    Pila.Push('d');
    cout<<"Push(d) <-"<<endl;
    Pila.Push('e');
    cout<<"Push(e) <-"<<endl;

    cout<<"Pop()= "<<Pila.Pop()<<endl;
    cout<<"Pop()= "<<Pila.Pop()<<endl;
    cout<<"Pop()= "<<Pila.Pop()<<endl;
    cout<<"Pop()= "<<Pila.Pop()<<endl;
    cout<<"Pop()= "<<Pila.Pop()<<endl;
    cout<<"Pop()= "<<Pila.Pop()<<endl;
    return 0;
}

```

## -Resultados

Finalmente al ejecutarse el programa se da el siguiente resultado:



```

C:\Users\Usuario\Documents\Ingenieria en Desarrollo de Software\4 Semestre\Algoritmia y Bases de Datos\Segundo Parcia...
Push(a) <-Push(b) <-Push(c) <-Push(d) <-Push(e) <-
Pop()= e
Pop()= d
Pop()= c
Pop()= b
Pop()= a
Pop()= 0

-----
Process exited after 0.05591 seconds with return value 0
Presione una tecla para continuar . . .

```

Como se puede apreciar en la imagen se insertan cinco datos de tipo carácter en la pila en el siguiente orden: a, b, c, d, e. Y al momento de realizar las funciones “Pop()” los datos de la pila se van retirando a partir del ultimo dato que se ingresó que en este caso es ‘e’ y al momento de vaciar la pila se realiza la verificación de que esta vacía y en este caso la pila comienza a retornar ‘0’;

## **-Conclusión**

Las pilas son una estructura de datos de tipo LIFO (Last In, First Out) las cuales tienen un comportamiento muy interesante ya que al tener dos operaciones principales (push y pop) estas actúan de una manera en la que se ingresan datos formando una especie de pila o como se mencionó anteriormente una “montaña de hot cakes” en la cual para poder retirar un dato, este actúa de manera que solo se pueden retirar los datos que llegaron al último uno por uno. Aunque las pilas son muy útiles, hay que tener cuidado al manejarlas, ya que se pueden dar problemas como puede ser un desbordamiento de pila (stack overflow) o querer seguir retirando datos cuando la pila está vacía (stack underflow). Con esta investigación y práctica me quedo mucho más claro como funcionan las pilas, debido a que este tema lo había visto anteriormente pero no lo había comprendido muy bien del todo.

## **-Referencias**

Barrera, F. J., & Bejarano Gómez, L. D. (s. f.). 3.2. *Pilas / Programación avanzada: Estructuras de datos y funciones*. IEDA. Recuperado 23 de septiembre de 2021, de [http://agrega.juntadeandalucia.es/repositorio/02122016/a5/es-an\\_2016120212\\_9131705/32\\_pilas.html](http://agrega.juntadeandalucia.es/repositorio/02122016/a5/es-an_2016120212_9131705/32_pilas.html)

Trinidad, P. (2019, 18 abril). *Estructuras de Datos[3] - Arreglos, pilas y colas*. Pablo Trinidad. <https://pablotrinidad.me/edd-3-arreglos-pilas-y-colas/>