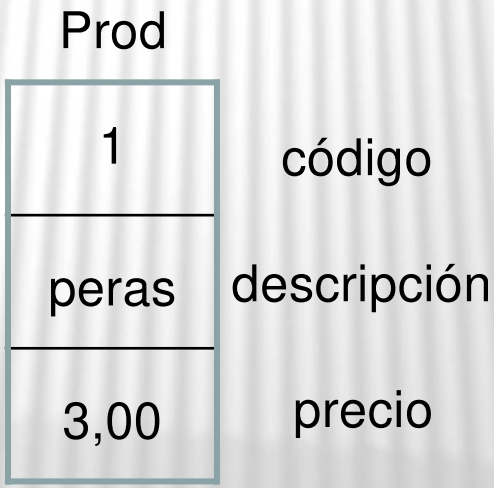
ESTRUCTURAS

Una estructura en un conjunto de variables que se citan y manejan con el mismo nombre y que permite además la utilización individual de sus elementos



ESTRUCTURAS DE DATOS

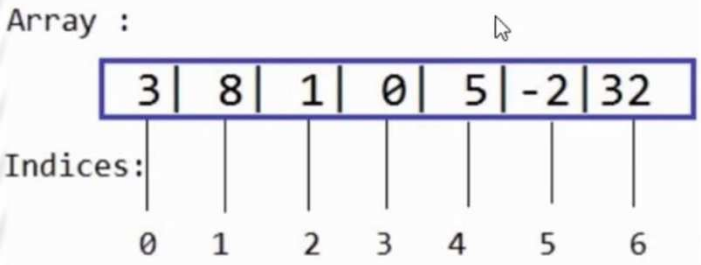
Las estructuras de datos son útiles porque nos permiten tener una batería de herramientas para solucionar ciertos tipos de problemas.

Además, nos permiten hacer un software más eficiente optimizando recursos.

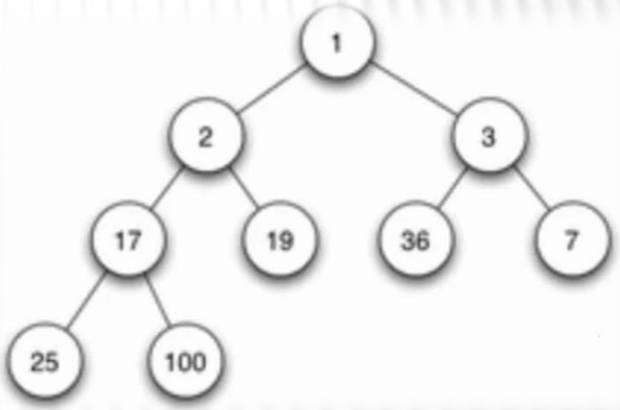
Las estructuras de datos se emplean con el objetivo principal de organizar los datos contenidos dentro de la memoria del ordenador.

ESTRUCTURAS DE DATOS MÁS CONOCIDAS

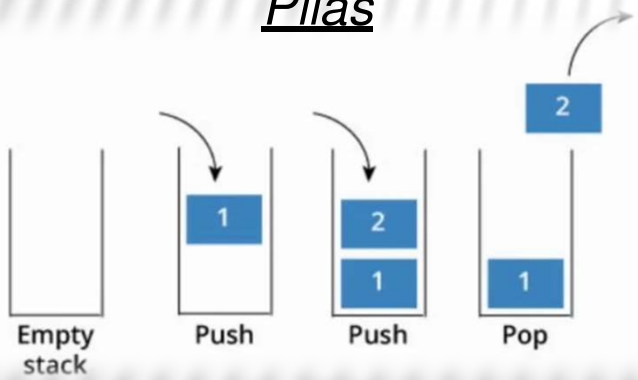
Arrays



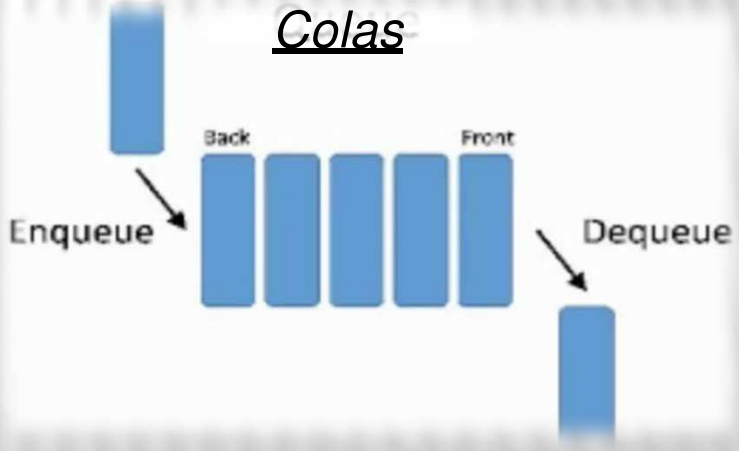
Arboles Binarios



Pilas



Colas



FORMATO DE DEFINICIÓN DE UNA VARIABLE ESTRUCTURA:

```
struct NombreEstructura
    {
        TipoVariable1   NombreVariable1;
        TipoVariable2   NombreVariable2;
        TipoVariable3 NombreVariable3;
                …
        TipoVariableN NombreVariableN;
    } NombreVarEstru1, NombreVarEstru2, …, NombVarEstruN;
```

o también:

```
struct NombreEstructura
    {
        TipoVariable1   NombreVariable1;
        TipoVariable2   NombreVariable2;
        TipoVariable3 NombreVariable3;
                …
        TipoVariableN NombreVariableN;
    };

struct NombreEstructura NombreVarEstru1, NombreVarEstru2, …, NombVarEstruN;
```

FORMATO DE DEFINICIÓN DE UNA VARIABLE ESTRUCTURA:

Si solamente se necesita una variable estructura, no es necesario poner el nombre de la estructura y quedaría así:

```
struct
     {
          TipoVariable1   NombreVariable1;
          TipoVariable2   NombreVariable2;
          TipoVariable3 NombreVariable3;
                        …
           TipoVariableN NombreVariableN;
      } NombreVarEstru1;
```

EJEMPLO DE UNA VARIABLE ESTRUCTURA:

```
struct
     {
          char nombres[50];
          char apellidos[50];
          char direccion [100];
          char telefono [7];
          float saldo;
} cuenta;
```

ESTRUCTURAS DE DATOS

- Una estructura de datos define la organización e interrelación de estos y un conjunto de ***operaciones*** que se pueden realizar sobre ellos. Las operaciones básicas son:
  - *Agregar*: adicionar un nuevo valor a la estructura.
  - *Eliminar*: borrar un valor de la estructura.
  - *Búsqueda*: encontrar un determinado valor en la estructura para realizar una operación con este valor, en forma secuencial o binario (siempre y cuando los datos estén ordenados).
- Otras operaciones que se pueden realizar son:
  - *Ordenamiento*: de los elementos pertenecientes a la estructura.
  - *Unión*: dadas dos estructuras originar una nueva ordenaday que contenga a las apareadas.

Cada estructura ofrece ventajas y desventajas en relación a la simplicidad y eficiencia para la realización de cada operación. De esta forma, la elección de la estructura de datos apropiada para cada problema depende de factores como la frecuencia y el orden en que se realiza cada operación sobre los datos.

EJERCICIO 1.

Declarar un registro que permita almacenar el código, descripción y precio de un producto. Luego definir dos variables de dicho tipo, cargarlas e imprimir el nombre del producto que tiene mayor precio.

```
struct producto {
    int codigo;
    char descripcion[41];
    float precio;
}; //obligatorio el punto y coma
```

Es común declarar el registro fuera de cualquier función para luego poder utilizarlo en muchas funciones (de todos modos la declaración del struct podría estar dentro de la función main.

Para definir una variable de tipo "producto" le antecedemos la palabra clave struct junto con el nombre del struct.

```
struct producto
pro1,pro2;
```

```
struct producto
pro1; struct
producto pro2;
```

Estamos definiendo dos variables llamadas pro1 y pro2 de tipo producto. No hay ningún problema de definir cada variable en una línea distinta.

EJERCICIO 1.

Para cargar la variable pro1 debemos cargar campo a campo:

Es decir primero indicamos el nombre de la variable (pro1) y seguidamente el campo que estamos cargando (código).

```
printf("Ingrese el código del producto:"); scanf("%i",&pro1.codigo);
fflush(stdin);
printf("Ingrese la descripción:");
gets(pro1.descripcion);
printf("Ingrese el precio:");
scanf("%f",&pro1.precio);
```

La carga del segundo registro llamado pro2 es idéntico a la carga del primer registro.

Para verificar cual de los dos productos tiene un precio mayor debemos acceder al campo precio de cada uno de los dos productos:

```
if (pro1.precio>pro2.precio)
```

Lo mismo cuando procedemos a imprimir por pantalla debemos indicar que campo debemos mostrar y no podemos pasar el registro completo:

```
printf("El producto %s tiene un precio mayor",pro1.descripcion);
```

EJERCICIO 1.

```
#INCLUDE<STDIO.H>
#INCLUDE<CONIO.H>
    struct
    producto {
    int codigo;
    char descripcion[41];
    float precio;
    }; //obligatorio el punto y coma

    int main()
    {
    struct producto pro1,pro2;
    printf("Ingrese el código del producto:");
    scanf("%i",&pro1.codigo);
    fflush(stdin);
    printf("Ingrese la descripción:");
    gets(pro1.descripcion);
    printf("Ingrese el precio:");
    scanf("%f",&pro1.precio);
    printf("Ingrese el código del producto:");
    scanf("%i",&pro2.codigo);
    fflush(stdin);
    printf("Ingrese la descripción:");
    gets(pro2.descripcion);
    printf("Ingrese el precio:");
    scanf("%f",&pro2.precio);
    if (pro1.precio>pro2.precio)
    {
        printf("El producto %s tiene un precio
mayor",pro1.descripcion);
    }
    else
    {
        if (pro2.precio>pro1.precio)
        {
            printf("El producto %s tiene un precio
mayor",pro2.descripcion);
        }
        else
        {
            printf("Tienen igual precio");
        }
    }
    getch();
    return 0;
}
```

PROBLEMAS PROPUESTOS.

Problema 1: Se tiene la siguiente declaración de registro:

```
struct producto {
     int codigo;
     char descripcion[41];
     float precio;
};
```

Plantear dos funciones una que cargue un registro de tipo producto y otra que lo imprima.

En la función main definir dos variables de tipo producto llamar a las funciones anteriores.

Problema 2: Se tiene la siguiente declaración de registro:

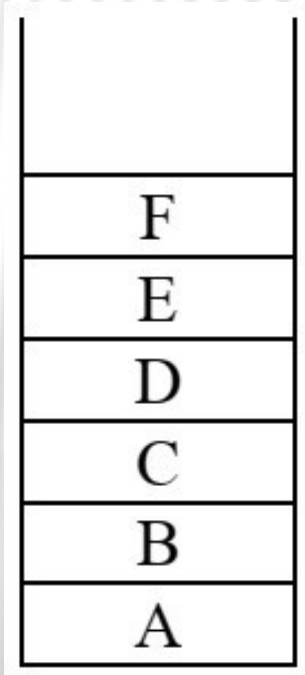
Definir un vector de 3 elementos de tipo producto, realizar su carga e impresión.

```
struct fecha
    { int dd;
    int mm;
    int aa;
};
struct producto
    { int
    codigo;
    char
    descripcion[41];
    float precio;
    struct fecha
    fechavencimiento;
};
```

ESTRUCTURA DE DATOS - PILAS.

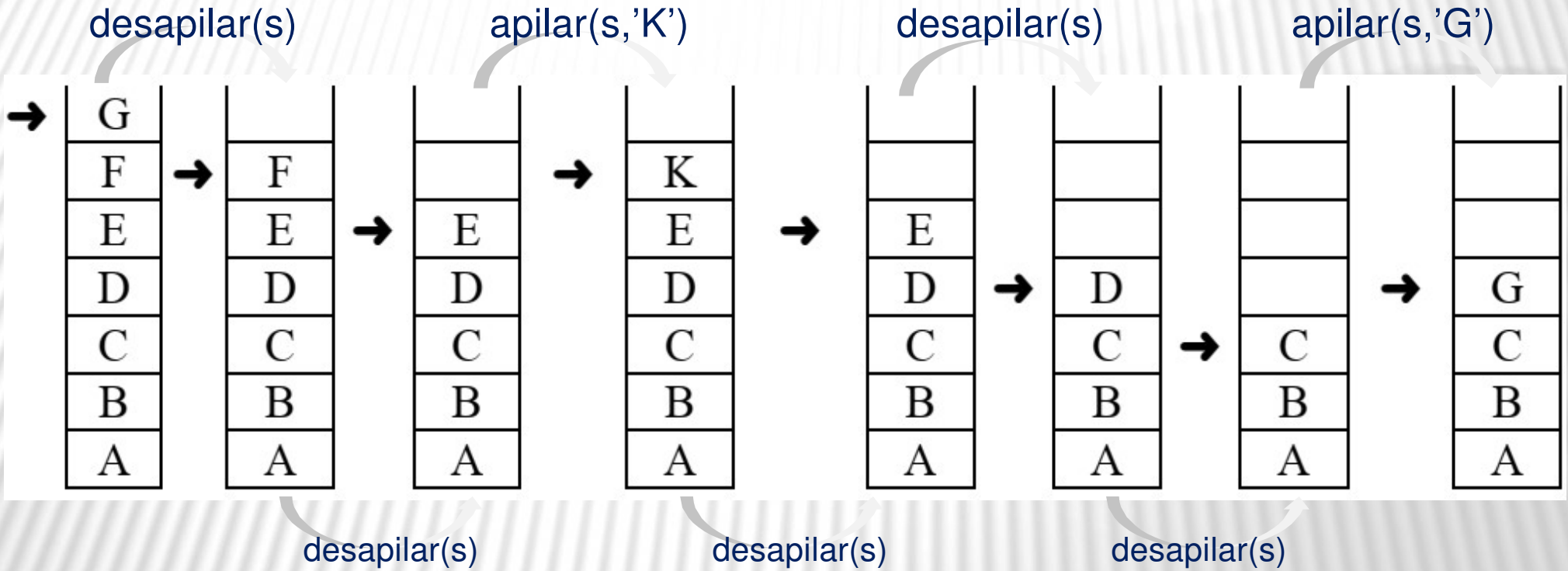
Una pila es un conjunto ordenado de elementos en el cual se pueden agregar y eliminar elementos en un extremo, que es llamado el tope de la pila.

- La definición de la pila considera la inserción y eliminación de elementos, por lo que una pila es un objeto dinámico en constante cambio.
- La definición especifica que un solo extremo de la pila se designa como el tope.
- Pueden colocarse nuevos elementos en el tope de la pila o se pueden quitar elementos.



Una pila con 6 elementos

ESTRUCTURA DE DATOS - PILAS.



La característica más importante de una pila es que el último elemento insertado en ella es el primero en suprimirse. Por esta razón, en ocasiones una pila se denomina una lista "último en entrar, primero en salir" o LIFO (last in, first out).

En la pila no se conserva un registro de los elementos intermedios que han estado en ella, si se desea conservar, debe llevarse en otra parte.

ESTRUCTURA DE DATOS - COLAS.

Una cola es un conjunto ordenado de elementos del que pueden suprimirse elementos de un extremo (llamado la parte delantera de la cola) y en el que pueden insertarse elementos del otro extremo (llamado la parte posterior de la cola).

- El primer elemento insertado en una cola es el primer elemento que se suprime.
- Por esta razón, una cola se denomina una lista fifo (el primero en entrar, el primero en salir) que es lo contrario de una pila, la cual es una lista lifo (el último en entrar, el primero en salir).

ESTRUCTURA DE DATOS - COLAS.

Se aplican tres operaciones primitivas a una cola:

1. La operación *insertar(q,x)* inserta el elemento *x* en la parte posterior de la cola *q*.
2. La operación *x = remover(q)* suprime el elemento delantero de la cola *q* y establece su contenido en *x*.
3. La operación, *vacia(q)* retorna *false* o *true*, dependiendo si la cola contiene elementos o no.

Podría considerarse una operación adicional que indique si la cola se encuentra llena.

La operación *insert* puede ejecutarse siempre, pues no hay un límite -teórico- en la cantidad de elementos que puede contener una cola. Sin embargo, la operación *remove* sólo puede aplicarse si la cola no está vacía; no hay forma de remover un elemento de una cola que no contiene elementos. El resultado de un intento no válido de remover un elemento de una cola vacía se denomina **subdesbordamiento**. La operación *empty* siempre es aplicable.

PROBLEMAS PROPUESTOS.

1. Pilas. Representar e implementar una Pila de 50 ítems de cualesquiera de las estructuras presentadas mas conveniente para el caso, incluyendo las funciones: apilar, desapilar, cantidad de elementos, pila vacía, pila llena, se debe utilizar arreglos.

2. Colas. Representar e implementar una Cola de 100 ítems de cualesquiera de las estructuras presentadas mas conveniente para el caso, incluyendo las funciones: insertar, remover, cantidad de elementos, cola vacía, cola llena, se debe utilizar arreglos.

A

```
struct producto {
     int codigo;
     char descripcion[41];
     float precio;
};
```

B

```
struct persona {
     int cedula;
     char nombre[50];
     char apellido[50];
     char sexo[1];
     Fecha fecha_nacimiento;
     long teléfono;
};
```

D

```
struct vehiculo {
     char placa[8];
     char marca[100];
     char modelo[100]
     int año;
     enum tipo {Moto, Auto, Camioneta, Bus}
};
```

C

```
struct libro {
     char isbn[20]
     char titulo [60];
     char autor [30];
     char editorial [30];
     int anio_publicacion;
};
```

Nota: según sea el caso con la estructura a utilizar, mencionar brevemente el contexto para su uso.