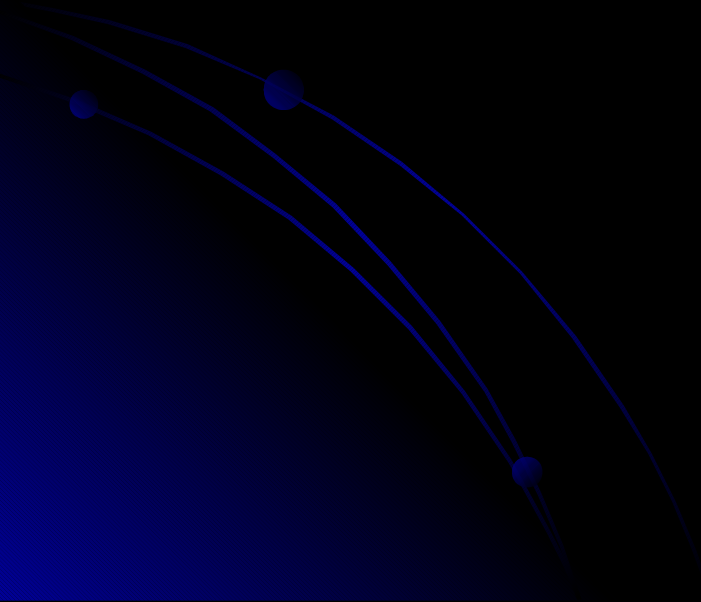


Árbol binario



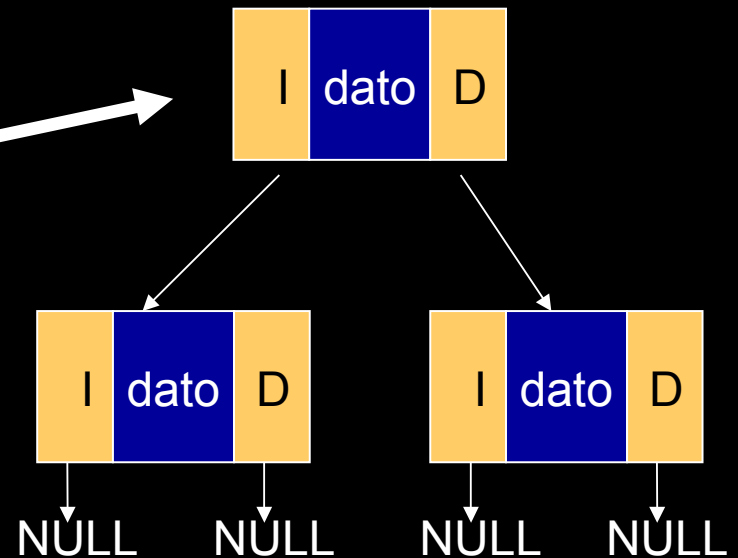
Definición

Un Árbol Binario es un conjunto finito de Elementos, de nombre Nodos de forma que:

- El Árbol Binario es Vacio si no tiene ningún elemento en el.
- El Árbol Binario contiene un Nodo Raíz y dos que parten de él, llamados Nodo Izquierdo y Nodo Derecho (que también son árboles).

Estructura de datos

```
struct nodoArbol  
{  
    int dato;  
    nodoArbol *izq;  
    nodoArbol *der;  
};
```

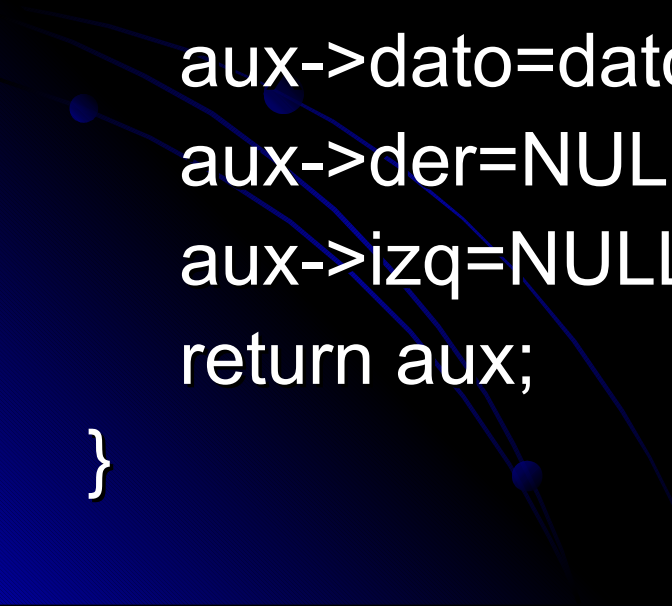


Árbol formado por un nodo raíz y dos subárboles


```
nodoArbol * inicArbol()
{
    return NULL;
}

nodoArbol * crearNodoArbol(int dato)
{
    nodoArbol * aux = (nodoArbol *) malloc(sizeof
                                                ( nodoArbol ) );

    aux->dato=dato;
    aux->der=NULL;
    aux->izq=NULL;
    return aux;
}
```

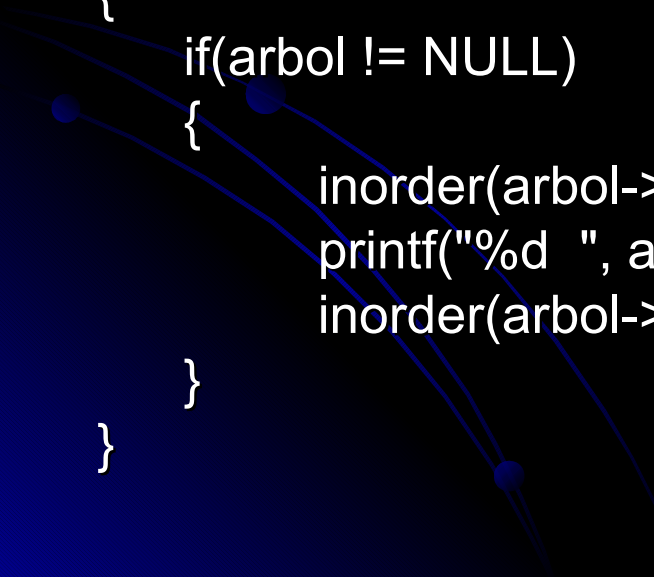


```
nodoArbol * insertar(nodoArbol * arbol, int dato)
{
    if(arbol==NULL)
        arbol = crearNodoArbol(dato);
    else
    {
        if(dato>arbol->dato)
            arbol->der = insertar(arbol->der, dato);
        else
            arbol->izq = insertar(arbol->izq, dato);
    }
    return arbol;
}
```

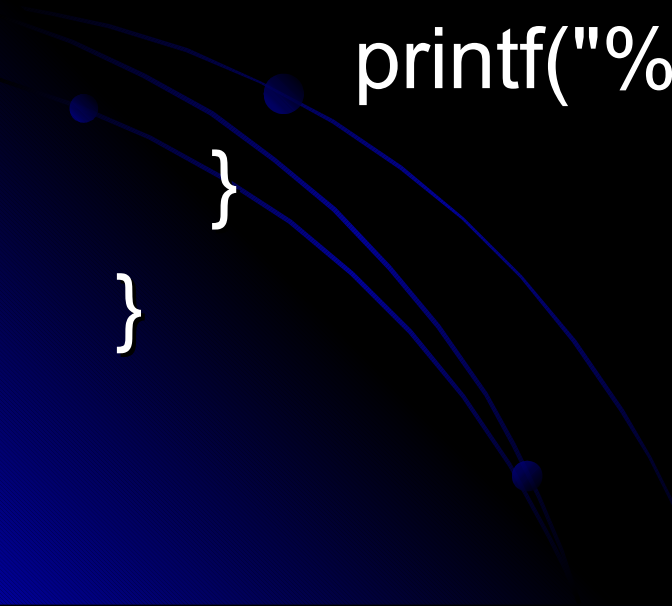


```
void preorder(nodoArbol * arbol)
{
    if(arbol != NULL)
    {
        printf("%d ", arbol->dato);
        preorder(arbol->izq);
        preorder(arbol->der);
    }
}
```

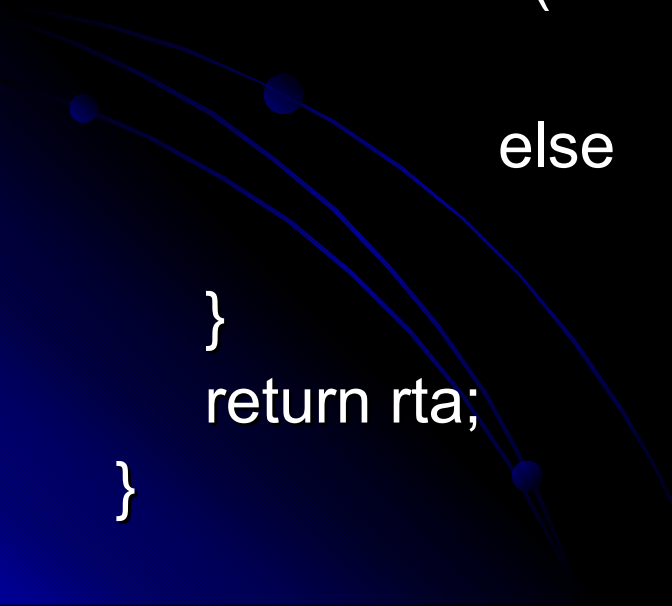
```
void inorder(nodoArbol * arbol)
{
    if(arbol != NULL)
    {
        inorder(arbol->izq);
        printf("%d ", arbol->dato);
        inorder(arbol->der);
    }
}
```



```
void postorder(nodoArbol * arbol)
{
    if(arbol != NULL)
    {
        postorder(arbol->izq);
        postorder(arbol->der);
        printf("%d ", arbol->dato);
    }
}
```



```
nodoArbol * buscar(nodoArbol * arbol, int dato)
{
    nodoArbol * rta=NULL;
    if(arbol!=NULL)
    {
        if(dato == arbol->dato)
            rta = arbol;
        else
            if(dato>arbol->dato)
                rta = buscar(arbol->der, dato);
            else
                rta = buscar(arbol->izq, dato);
    }
    return rta;
}
```




```
void main()
{
    int a[10]= {1,10,2,45,9,15,46,33,25,20};
    nodoArbol * arbol = inicArbol();
    for (int i=0; i<10; i++)
    {
        arbol = insertar(arbol, a[i]);
    }
    printf("PREORDER :");
    preorder(arbol);
    printf("\nINORDER :");
    inorder(arbol);
    printf("\nPOSTORDER :");
    postorder(arbol);
}
```

Borrar un nodo de un árbol binario:

Supongamos un árbol binario con la siguiente estructura:

```
typedef struct
{
    int r,
    struct nodoArbol * i;
    struct nodoArbol * d;
}nodoArbol;

nodoArbol * A;
```

Sea la cabecera de la función borrar: *nodoArbol * borrar (nodoArbol * A, int dato)*

Para borrar un nodo hacemos esto:

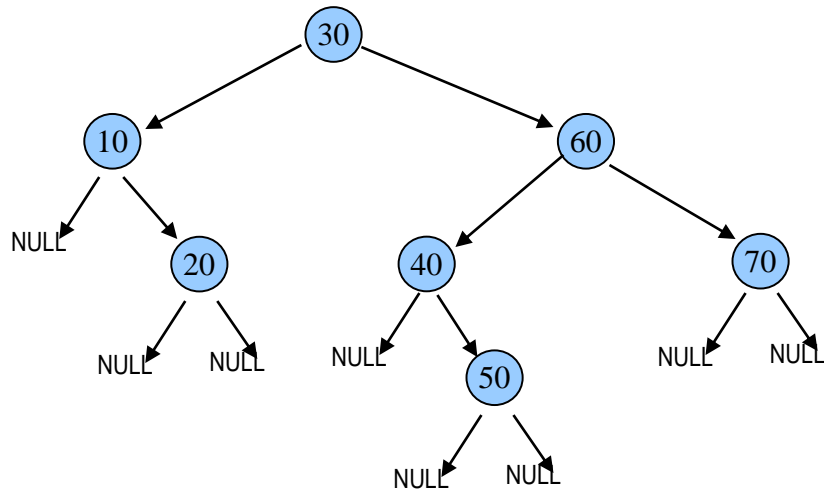
A = borrar (A, dato);

donde borrar(A, dato) es igual a:

Si A != NULL	Si dato == A->r	Si A->i != NULL	nodoArbol * masDer = NMD(A->i); A->r = masDer->r; A->i = borrar(A->i, masDer->r);	Al encontrar el nodo con el dato buscado, sobrescribo este dato con el dato del “nodo más derecho” del subárbol izquierdo (si existe nodo izquierdo)
		Sino si A->d != NULL	nodoArbol * masIzq = NMI(A->d); A->r = masIzq->r; A->d = borrar(A->d, masIzq->r);	Similar al caso anterior, pero considerando el otro subárbol.
		Sino (cuando A es hoja)	Free (A); A = NULL;	Cuando A es hoja, simplemente se libera la memoria, se la establece en NULL y se la retorna.
	Si dato > A->r		A->der = borrar (A->d, dato);	Se busca el dato recursivamente en el subarbol derecho
	Si dato < A->r		A->izq = borrar (A->i, dato);	Se busca el dato recursivamente en el subarbol izquierdo.
Si A == NULL				El elemento buscado no está en el árbol. No hacer nada, no hay else
		Al final de la función	return A	Siempre retornar A;

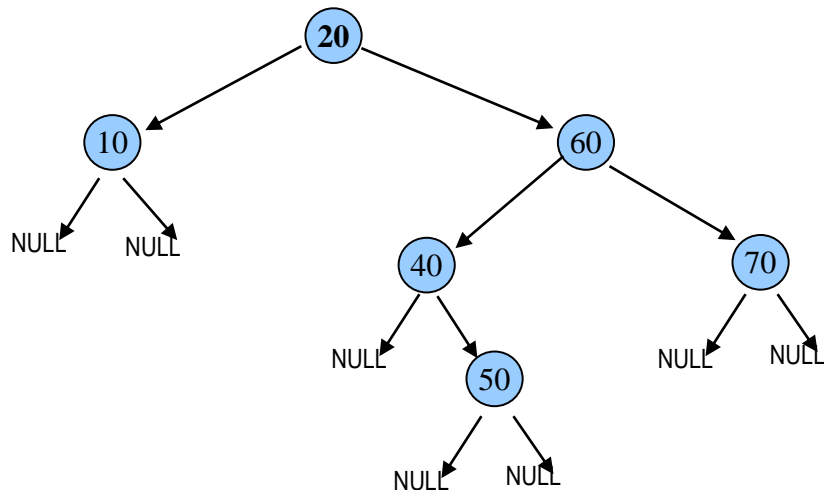
Al encontrar el elemento, si es hoja, simplemente se lo elimina; si no es hoja, debemos reemplazar el dato del nodo encontrado, con el dato siguiente en orden (el menor o el mayor). Este dato lo encontramos en el NMI(D) (nodo más izquierdo del subárbol derecho) o en el NMD(I) (nodo más derecho del subárbol izquierdo).

Pensar cómo se hacen estas dos funciones recursivas.



Por ejemplo, si se borra el nodo 30, el proceso será:

1. buscar el nodo 20 (nodo más derecho del subárbol izquierdo)
2. asignar al nodo 30 un dato (raíz) igual a 20.
3. asignar al hijo izquierdo del viejo nodo 30 (nuevo nodo 20) el resultado de borrar al subárbol izquierdo del viejo 30 el nodo con valor 20, o sea:
 - a. $A \rightarrow \text{izq} = \text{borrar}(A \rightarrow \text{izq}, \text{NMD}(A \rightarrow \text{izq}))$;
 - b. Al ejecutar esta entrada en recursión, se ubica el nodo 20, que como es un nodo hoja (no tiene hijos), directamente se lo elimina del árbol.




alumno la codificación del algoritmo. Igualmente se puede preguntar en clase.



Borrar un nodo



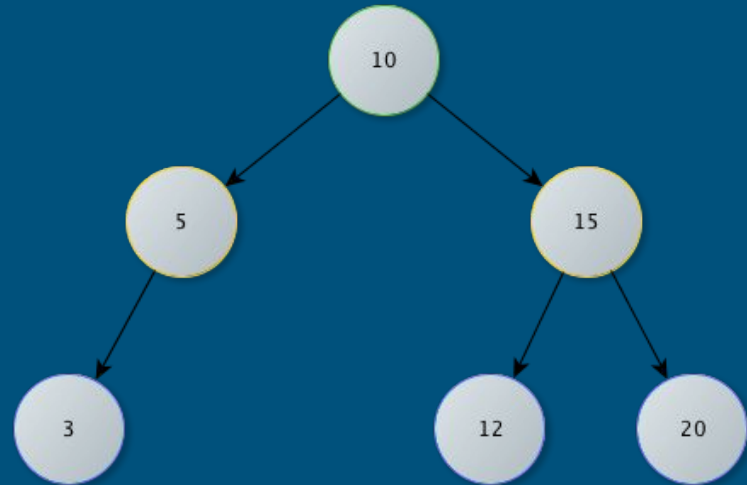
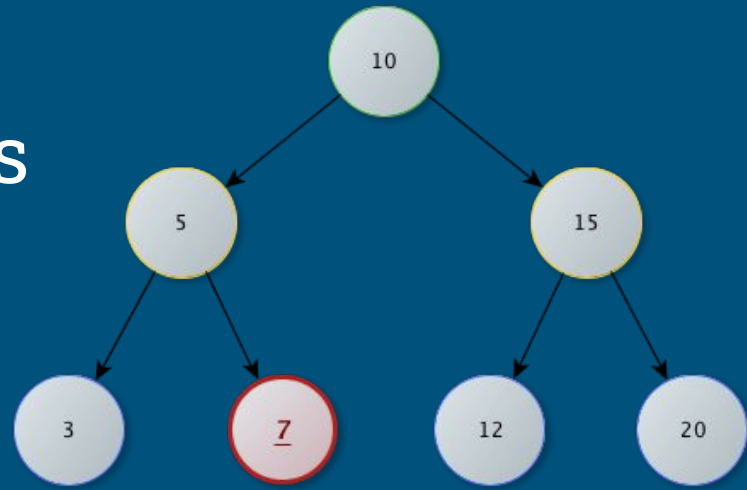
Al mal tiempo, buena cara. O eso me decían cuando me enseñaron a borrar un nodo



Casos a contemplar

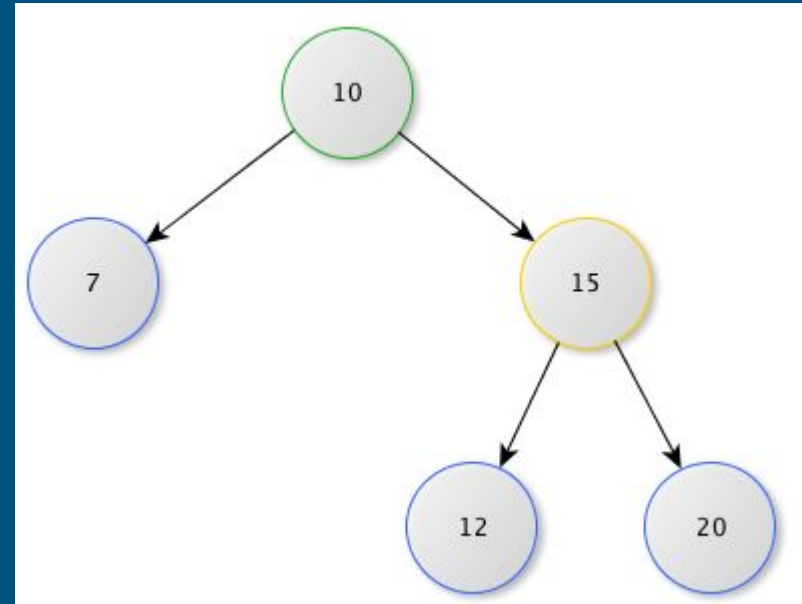
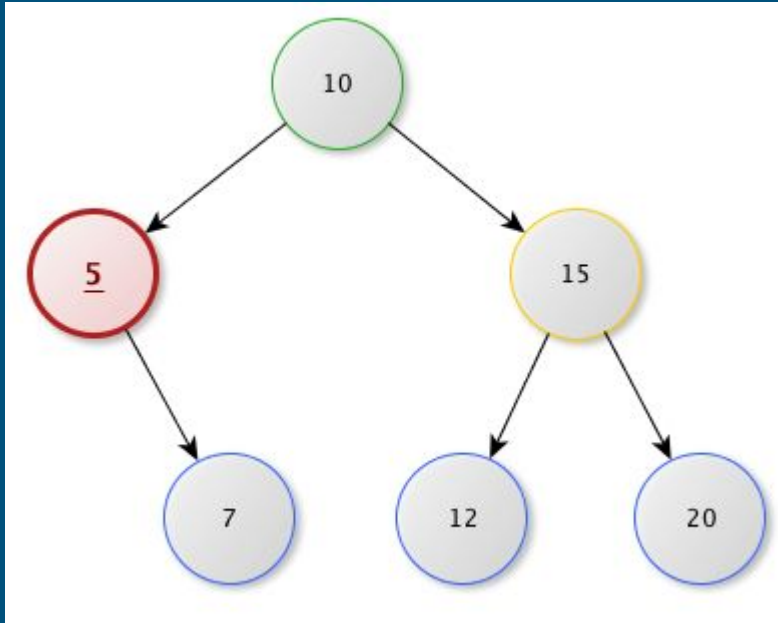
- Borrar un Nodo sin hijos
- Borrar un Nodo con un subárbol hijo
- Borrar un Nodo con dos subárboles hijos

Borrar un Nodo sin hijos



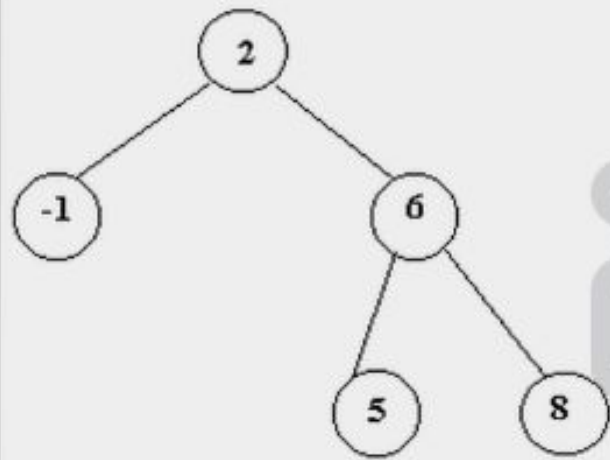
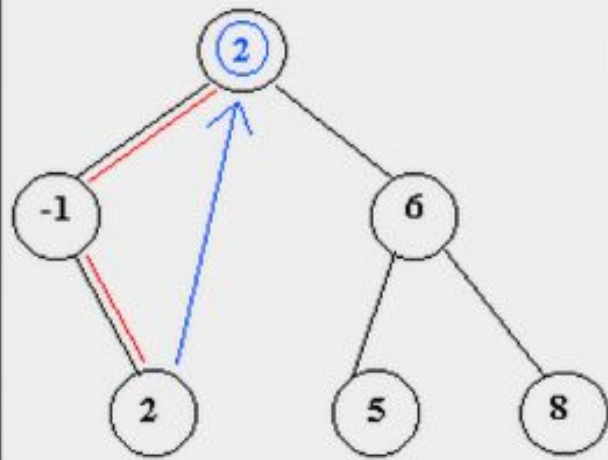
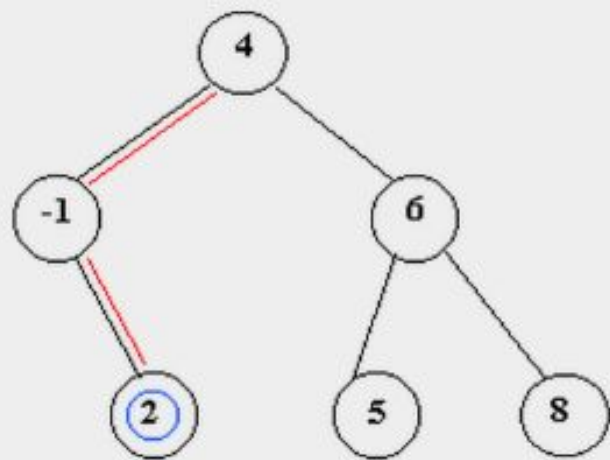
Borrar un Nodo con un subárbol hijo

Tenemos que borrar el Nodo y el subárbol que tenía pasa a ocupar su lugar



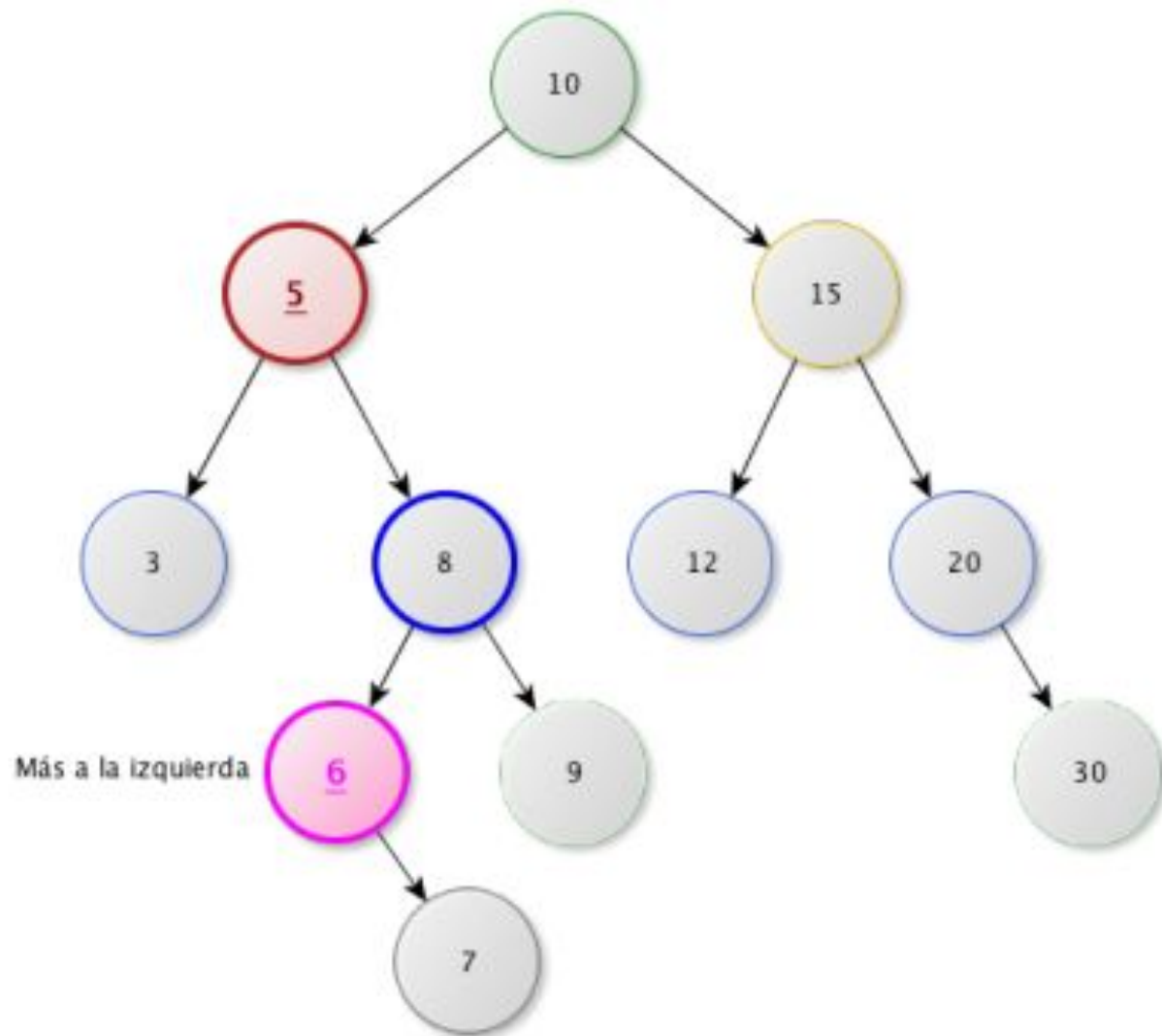
Borrar un Nodo con dos subárboles hijos

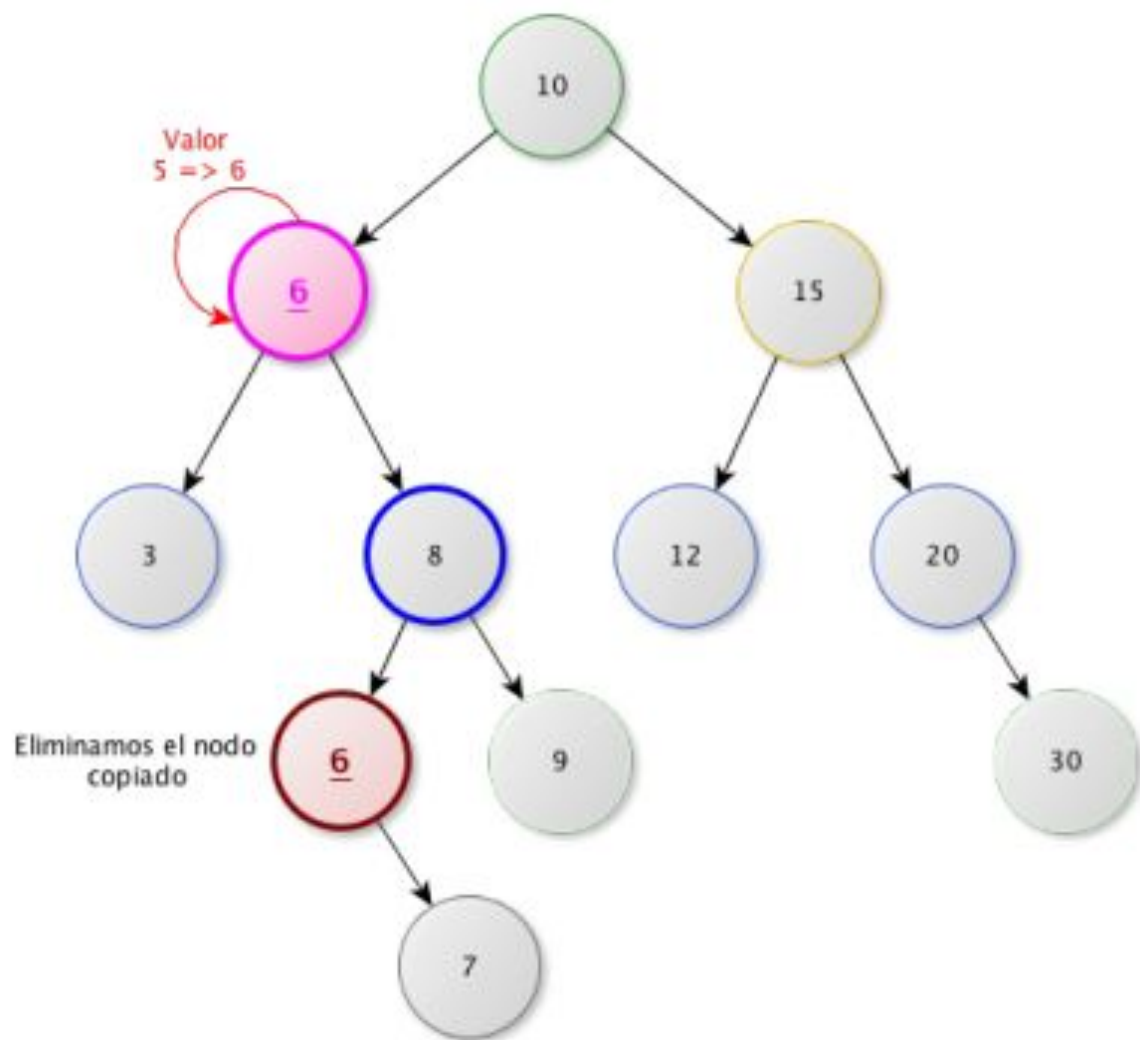
- Mantener la coherencia de los enlaces.
- Seguir la estructura como árbol de búsqueda binario.
- Sustituir la información del nodo borrado por una de sus hojas y borrar dicha hoja. Pero cual?
 - La mayor de las claves menores. La más derecha de la izquierda.
 - La menor de las claves mayores. La más izquierda de la derecha.

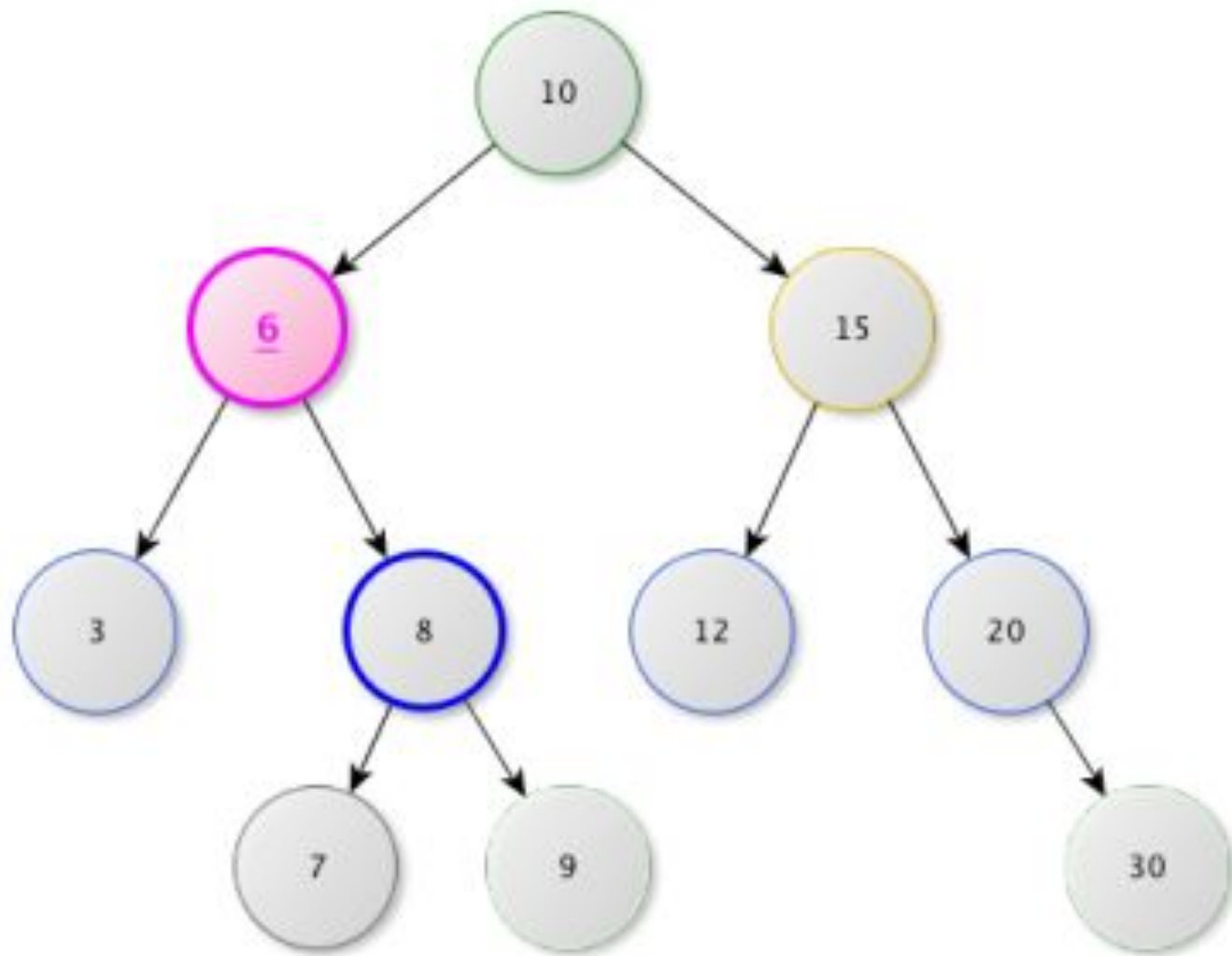


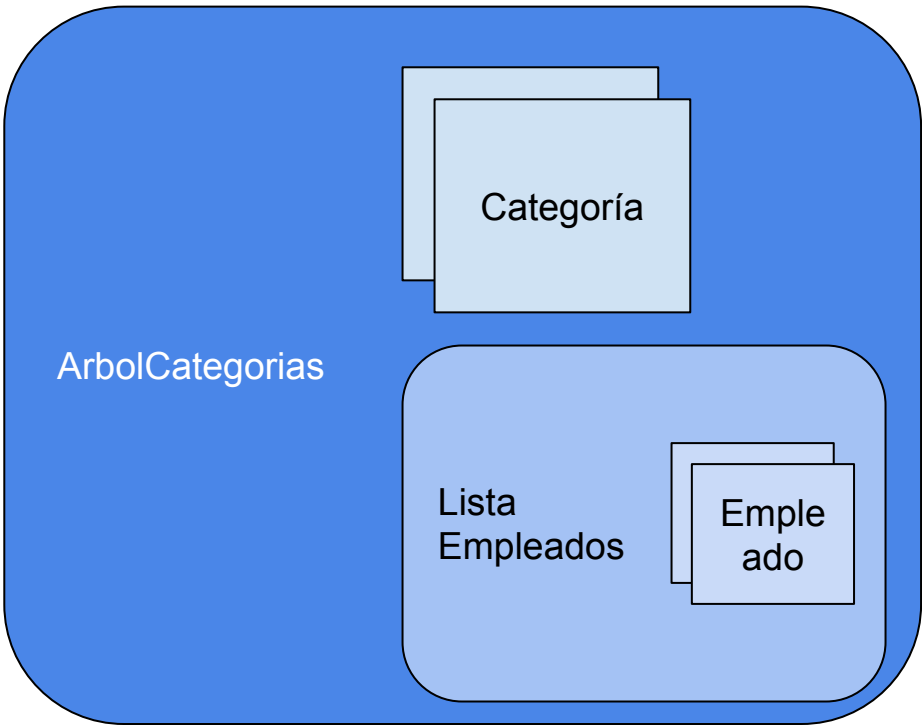
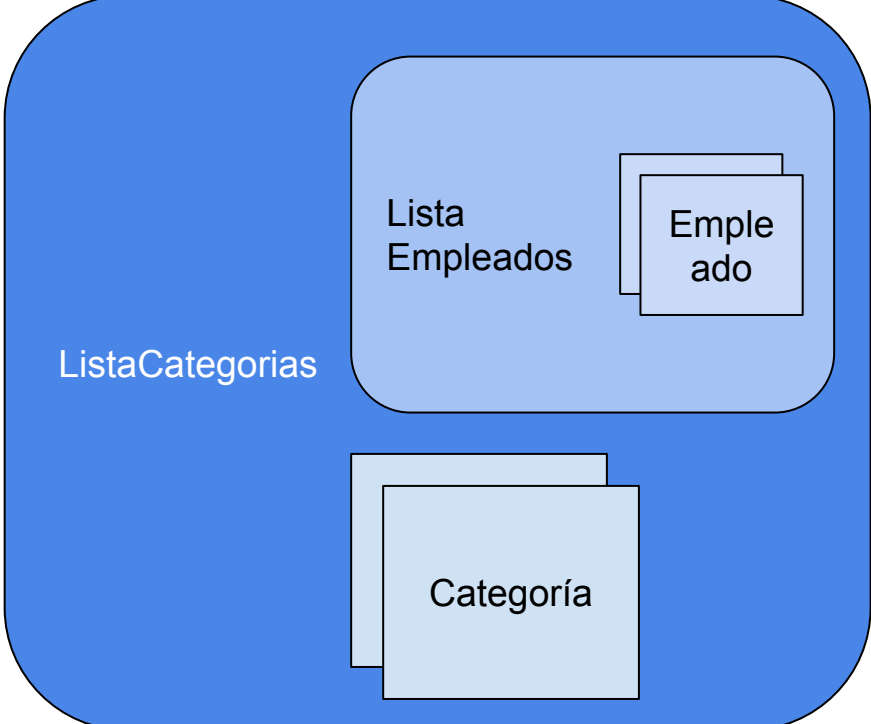
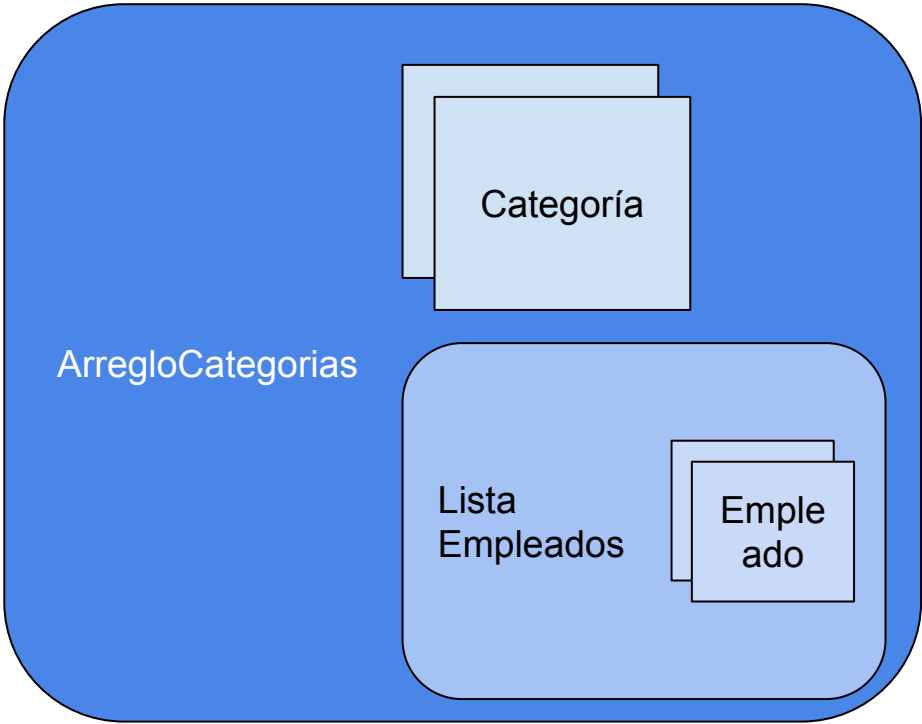
Borrar un Nodo con dos subárboles hijos

Tenemos que tomar el hijo derecho del Nodo que queremos eliminar y recorrer hasta el hijo más a la izquierda (hijo izquierdo y si este tiene hijo izquierdo repetir hasta llegar al último nodo a la izquierda), reemplazamos el valor del nodo que queremos eliminar por el nodo que encontramos (el hijo más a la izquierda), el nodo que encontramos por ser el más a la izquierda es imposible que tenga nodos a su izquierda pero si que es posible que tenga un subárbol a la derecha, para terminar solo nos queda proceder a eliminar este nodo de las formas que conocemos (caso 1, caso 2) y tendremos la eliminación completa.









Categorías

Categoria InicializarCategoria()

Categoria NuevaCategoria(char categoria, int escala, float porcen)

void CargarCategoria(Categoria *categoria, char cat, int escala, float porcen)

void ImprimirCategoria(Categoria cat)

Empleados

Empleado NuevoEmpleado(char cat, int legajo, char * nombre, float sueldo)

void ImprimirEmpleado(Empleado e)

Lista de Empleados

ListaEmpleados * CrearNodo(Empleado e)

ListaEmpleados * AgregarAlPpio(ListaEmpleados * lista, ListaEmpleados * nuevoNodo)

void MostrarListaEmpleados(ListaEmpleados * lista)

Lista de Categorías

ListaCategorias * CrearNodoCategoria(Categoria cat)

ListaCategorias * AgregarAlPpioListaCat(ListaCategorias * lista, ListaCategorias * nuevoNodo)

void MostrarListaCategorias(ListaCategorias * lista)

??? CargarListaEmpleadosACategoria (????)

Estructuras Compuestas

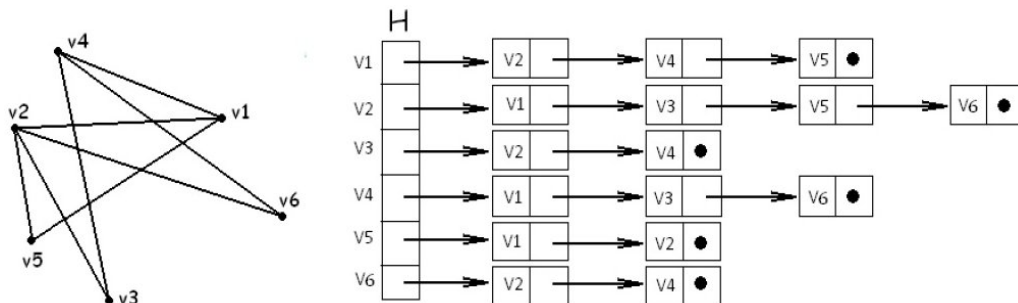
Son estructuras que permiten representar datos que resultan de la composición de estructuras más simples. Son de gran utilidad cuando se deben resolver problemas complejos que involucran la utilización de una gran variedad de tipos.

Las composiciones más frecuentes son:

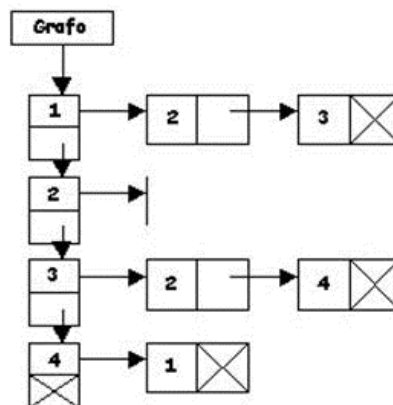
1. Arreglo de Listas.
2. Arreglo de Árboles.
3. Listas de Listas.
4. Listas de Árboles.
5. Árboles de Listas.

Ejemplos:

Un uso típico de un *arreglo de listas* es la representación de la lista de adyacencia de cada vértice de un grafo. La lista de adyacencia de un vértice dado me permite conocer de forma directa los vértices a los cuales puedo llegar. Si esta lista se ordena de menor a mayor por distancia (en el caso de un grafo de rutas) puedo obtener rápidamente el destino más cercano.



Las *listas de listas* pueden usarse cuando la estructura principal requiere la flexibilidad de una estructura dinámica (poder crecer, inserciones en orden y borrados rápidos, etc.). Un uso posible es para la representación de grafos o estructuras de datos con dos jerarquías claramente definidas (Categoría y Empleados).



Las estructuras que combinan árboles binarios son aquellas en donde se requiere una búsqueda eficiente de los elementos por una clave en particular, en especial si él o los árboles se mantienen balanceados.

Código de Ejemplo:

```
/****** TIPOS *****/
typedef struct RegCategoria {
    char categoria;
    int escala;
    float porcentaje;
} Categoria;

typedef struct RegEmpleado {
    char categoria;
    int legajo;
    char * nombre;
    float sueldo;
} Empleado;

typedef struct NodoListaEmpleados {
    Empleado empleado;
    struct NodoListaEmpleados * sig;
} ListaEmpleados;

typedef struct RegArregloCategorias {
    Categoria categoria;
    ListaEmpleados * empleados;
} ArregloCategorias;

typedef struct NodoListaCategorias {
    Categoria categoria;
    ListaEmpleados * empleados;
    struct NodoListaCategorias * sig;
} ListaCategorias;

typedef struct NodoArbolCategorias {
    Categoria categoria;
    ListaEmpleados * empleados;
    struct NodoArbolCategorias * izq;
    struct NodoArbolCategorias * der;
} ArbolCategorias;

/** TDA CATEGORIAS *****/
Categoria InicializarCategoria() {
    Categoria cat;
    cat.categoria = ' ';
    cat.escala = 0;
    cat.porcentaje = 0;
    return cat;
}

Categoria NuevaCategoria(char categoria, int escala, float porcen) {
    Categoria cat;
    cat.categoria = categoria;
    cat.escala = escala;
    cat.porcentaje = porcen;
    return cat;
}

void CargarCategoria(Categoria *categoria, char cat, int escala, float porcen) {
    (*categoria).categoria = cat;
    (*categoria).escala = escala;
    (*categoria).porcentaje = porcen;
}

void ImprimirCategoria(Categoria cat) {
    printf("\nCategoria: %c", cat.categoria);
    printf(" | Escala: %d", cat.escala);
    printf(" | Porcentaje: %.2f", cat.porcentaje);
}
```



```

/** TDA EMPLEADOS *****/
Empleado NuevoEmpleado(char cat, int legajo, char * nombre, float sueldo) {
    Empleado e;
    e.categoria = cat;
    e.legajo = legajo;
    e.nombre = nombre;
    e.sueldo = sueldo;
    return e;
}

void ImprimirEmpleado(Empleado e) {
    printf("\nEmpleado: %s", e.nombre);
    printf(" | Categoria: %c", e.categoria);
    printf(" | Legajo: %d", e.legajo);
    printf(" | Sueldo: %.2f", e.sueldo);
}

/** TDA LISTA de EMPLEADOS *****/
ListaEmpleados * CrearNodo(Empleado e) {
    ListaEmpleados * nodo = (ListaEmpleados *) malloc(sizeof(ListaEmpleados));
    nodo->empleado = e;
    nodo->sig = NULL;
    return nodo;
}

ListaEmpleados * AgregarAlPpio(ListaEmpleados * lista, ListaEmpleados * nuevoNodo) {
    if (lista == NULL) {
        lista = nuevoNodo;
    }
    else {
        nuevoNodo->sig = lista;
        lista = nuevoNodo;
    }
    return lista;
}

void MostrarListaEmpleados(ListaEmpleados * lista) {
    ListaEmpleados * aux = lista;
    if (aux == NULL) {
        printf("\nSin Empleados");
    }
    else {
        while (aux != NULL) {
            ImprimirEmpleado(aux->empleado);
            aux = aux->sig;
        }
    }
}

/** TDA LISTA de CATEGORIAS *****/
ListaCategorias * CrearNodoCategoria(Categoria cat) {
    ListaCategorias * nodo = (ListaCategorias *) malloc(sizeof(ListaCategorias));
    nodo->categoria = cat;
    nodo->empleados = NULL;
    nodo->sig = NULL;
    return nodo;
}

ListaCategorias * AgregarAlPpioListaCat(ListaCategorias * lista, ListaCategorias * nuevoNodo) {
    if (lista == NULL) {
        lista = nuevoNodo;
    }
    else {
        nuevoNodo->sig = lista;
        lista = nuevoNodo;
    }
    return lista;
}

```

```

void MostrarListaCategorias(ListaCategorias * lista) {
    ListaCategorias * aux = lista;
    if (aux == NULL) {
        printf("\nSin Categorias");
    }
    else {
        while (aux != NULL) {
            ImprimirCategoria(aux->categoria);
            MostrarListaEmpleados(aux->empleados);
            aux = aux->sig;
            printf("\n"); // paso a otra categoría
        }
    }
}

/**/
int main()
{
    ArregloCategorias a[100];
    a[0].categoria = InicializarCategoria();
    ImprimirCategoria(a[0].categoria);
    CargarCategoria(&a[0].categoria, 'A', 1, 100);
    ImprimirCategoria(a[0].categoria);
    printf("\n");

    Empleado e = NuevoEmpleado('A', 1, "Ramiro", 5000);

    ListaEmpleados * empleados;
    empleados = InicListaEmpleados();
    empleados = AgregarAlPpio(empleados, CrearNodo(e));
    empleados = AgregarAlPpio(empleados, CrearNodo(NuevoEmpleado('A', 2, "Jorge", 3000)));
    MostrarListaEmpleados(empleados);
    printf("\n");

    ListaCategorias * categorias;
    categorias = InicListaCategorias();
    categorias = AgregarAlPpioListaCat(categorias, CrearNodoCategoria(NuevaCategoria('B', 2, 80)));
    categorias = AgregarAlPpioListaCat(categorias, CrearNodoCategoria(a[0].categoria));
    // la siguiente asignación se utiliza por la simplicidad del ejemplo (el TDA debería proveer
    // un método para cargar empleados o una lista completa de empleados validando que se ingresen
    // en la categoría correspondiente)
    categorias->empleados = empleados;
    MostrarListaCategorias(categorias);

    printf("\n\nFin del programa!\n");
    return 0;
}

```

El presente ejemplo ilustra, brevemente, la combinación de las principales estructuras de datos (estáticas y dinámicas) estudiadas. Para esto se definen dos **tipos** principales “**Categoría**” y “**Empleado**” que representan: las categorías y los datos básicos de cada empleado de una empresa. A partir del tipo Empleado se define una estructura dinámica definida por el tipo **ListaEmpleados** cuyo TDA debe proveer todas funciones para su administración.

A partir de estos tipos básicos se definen las siguientes estructuras compuestas:

1. **ArregloCategorias**: un arreglo de listas en el cual cada celda contiene un registro de tipo Categoría y una lista de todos los Empleados de la misma.
2. **ListaCategorias**: una lista de listas en dónde cada nodo contiene un registro de tipo Categoría y una lista de todos los Empleados de la misma.
3. **ArbolCategorias**: una árbol de categorías en dónde cada nodo contiene un registro de tipo Categoría y una lista de todos los Empleados de la misma.

En todos los casos cada TDA debe proveer todas las funciones necesarias con sus respectivas validaciones. Por ejemplo, proveer la inserción de un Empleado en la categoría correcta.

La elección de una estructura u otra dependerá de los requerimientos particulares del problema a resolver. En cada caso se deberá analizar los mismos y elegir la mejor opción en base a, por ejemplo: tamaño a ocupar en memoria (datos fijos cuyo tamaño se conoce de antemano o dinámicos), eficiencia requerida para las búsquedas, cantidad altas y bajas esperadas, etc. Todos estos criterios podrán lograrse en menor o mayor medida con cada una de las combinaciones planteadas.

```

#include <stdio.h>
#include <stdlib.h>
struct nodo
{
    int valor;
    //int dato;
    struct nodo *std;
};

struct celda
{
    int id;
    struct nodo *lista;
};

int agregar(celda a[100], int cant, celda nueva)    // int id
{
    a[cant]=nueva;
    cant++;
    return cant;
}

int existe(celda a[100], int cant, int id)
{
    int rta = -1;
    int i = 0;
    while (i<cant && rta == -1)
    {
        if (a[i].id == id)
        {
            rta = i;
        }
        i++;
    }
    return rta;
}

int agregartodo(celda a[100],int cant, int id, int valor)
{
    nodo *aux = crearnodo(valor);
    int i = existe(a, cant, id);
    if (i == -1)
    {
        celda una;
        una.id=id;
        una.lista=null;
        cant = agregar(a, cant, una);
        i = cant - 1;
    }
    a[i].lista = agregarppio(a[i].lista, aux);
    return cant;
}

```

```

nodo *buscar (celda a[100], int cant, int id, int valor)
{
    nodo *aux = null;
    int pos = existe(a, cant, id);
    if (pos != -1)
    {
        aux = buscarnodo(a[pos].lista, valor);
    }
    return aux;
}

```

```

main ()
{
    celda a[100];
}

```

```

struct registro
{
    mat;
    alu;
    nota;
};

```

```

int pasar (char nombre_archivo[30], celda a[100], int cant)
{
    file *archi=fopen (.....);
    registro unreg;
    while (!feof(archi))
    {
        fread(&unreg,.....);
        cant = agregartodo(a, cant, unreg.mat, unreg.nombre, unreg.nota);
    }
    fclose(archi);
    return cant;
}

void pasar_de_arreglo_a_archivo(char nombre[30],celda a[100], int cant)
{
    nodo *seg;
    registro unreg;
    file *archi = fopen(      );
    for (i = 0; i < cant; i++)
    {
        seg = a[i].lista;
        unreg.mat = a[i].mat;
        while (seg != null)
        {
            strcpy(unreg.alu, seg->alu);
            unreg.nota = seg->nota;
            fwrite(&unreg,.....);
            seg = seg->ste;
        }
    }
    fclose(archi);
}

```

Evolución en la Programación

- 2) Lenguaje ensamblador (instrucciones con nombre): podían realizarse programas más largos y complejos usando una representación simbólica en vez de instrucciones de máquina.
- 3) Lenguajes de alto nivel: incorporan más herramientas para gestionar la creciente complejidad de los programas. El primero fue Fortran.
- 4) Programación estructurada (C y Pascal): Se pueden escribir de una forma sencilla programas moderadamente complejos.
- 5) Programación Orientada a Objetos: toma las mejores ideas de la programación estructurada y las combina con nuevos y poderosos conceptos que alientan a una nueva visión de las tareas de la programación.

Programación estructurada: ventajas

Con la programación estructurada, elaborar programas de computador sigue siendo una labor que demanda esfuerzo, creatividad, habilidad y cuidado. Sin embargo, con este estilo podemos obtener las siguientes ventajas:

1. Los programas son más fáciles de entender, ya que pueden ser leído de forma secuencial, sin necesidad de hacer seguimiento a saltos de línea (GOTO) dentro bloques de código para entender la lógica.
2. La estructura del programa es más clara puesto que las instrucciones están más ligadas o relacionadas entre sí.
3. Reducción del esfuerzo en las pruebas. El seguimiento de las fallas ("debugging") se facilita debido a la lógica más visible, por lo que los errores se pueden detectar y corregir más fácilmente.
4. Reducción de los costos de mantenimiento.
5. Programas más sencillos y más rápidos.
6. Los bloques de código son auto explicativos, lo que apoya a la documentación.

Programación estructurada: desventajas

- El principal inconveniente de este método de programación, es que se obtiene un único bloque de programa, que cuando se hace demasiado grande puede resultar problemático su manejo, esto se resuelve empleando la programación modular, definiendo módulos interdependientes programados y compilados por separado.
- La forma de resolver los problemas y organizar la información para realizar un sistema informático (datos por un lado y algoritmos por otro) está poco relacionada con la forma natural de observar y resolver el resto de las situaciones de la vida diaria.

Programación Orientada a Objetos

La programación orientada a objetos, plantea una nueva manera de pensar a la hora de resolver un sistema. Ya no se subdividen tareas complejas como un conjunto de tareas simples (sin tener en cuenta los datos), sino que se observa el problema y se lo subdivide en partes relacionadas, que contienen datos y código.

Se buscan (o diseñan) entidades independientes que colaboren en la resolución de un problema, cada una de estas entidades tendrá un conjunto de habilidades propias y responsabilidades designadas (cada entidad es responsable de algunas tareas).

Para resolver un sistema, se toma un conjunto de estas entidades y se las pone a interactuar entre si, enviando mensajes a cada una de ellas (entre ellas), proporcionando en cada mensaje la información necesaria para que la entidad pueda resolver la tarea y devolver una respuesta a quien le invocó.

Estas entidades, desde ahora “objetos”, colaboran entre si, comunicándose (enviándose mensajes) para poder resolver un problema complejo. Haciendo cada una de ellas una parte de la resolución, según sean sus habilidades o responsabilidades.

Estructurado vs. Objetos

Esto difiere de la programación estructurada tradicional, en la que los datos y los procedimientos están separados y sin relación, ya que lo único que se busca es el **procesamiento de unos datos de entrada para obtener otros de salida**. La programación estructurada anima al programador a pensar sobre todo en términos de procedimientos o funciones, y en segundo lugar en las estructuras de datos que esos procedimientos manejan. En la programación estructurada se escriben funciones y después les pasan datos.

Los programadores que emplean lenguajes orientados a objetos definen **objetos con datos y métodos** y después envían mensajes a los objetos diciendo que realicen esos métodos en sí mismos.

Mirando la realidad como POO.

Ejemplo

Enviando flores a mi tía que vive en España:

En este ejemplo el usuario del Sistema sería yo (actor) y el Sistema estaría formado por los siguientes objetos:

Florista en Mar del Plata.
Florista en España.
Transportista en España.
Mi tía en España.

Cómo se realiza un pedido:

- 1.- El actor (yo) envía el siguiente mensaje a la Florista de Mar del Plata: **enviarFlores**. Junto con el mensaje también envía la dirección en España, y una descripción del ramo.
- 2.- La florista de Mar del Plata se comunica con la de España y le envía a su vez el siguiente mensaje: **enviarFlores**. Junto con el mensaje también envía la dirección en España y una descripción del ramo. En definitiva delega el trabajo a otra persona.
- 3.- La florista de España prepara el pedido (ramo de flores), llama al transportista y le da el mensaje: **enviarPedido**. Junto con el mensaje le pasa el ramo de flores y la dirección. En definitiva, realiza parte de la tarea y delega otra parte al transportista.
- 4.- El transportista realiza la tarea de llevar las flores a mi tía de España.

Mirando la realidad como POO.

Ejemplo

Cada uno de los objetos conoce su trabajo y lo realiza al recibir un mensaje. En nuestro ejemplo, **enviarFlores** es el mensaje enviado y también es el nombre de una función (método) propia del objeto **Florista de España**. Se envía un mensaje invocando a un método propio del objeto que recibe el mensaje. Los datos que recibe el mensaje (descripción del ramo y dirección) son los parámetros que se le pasan al método para que pueda ejecutar su tarea.

Desde el punto de vista del actor (quien inicia las acciones) solamente se ha comunicado con la Florista de Mar del Plata y desconoce la forma en que ésta solucionará su pedido.

Desde el punto de vista de la Florista de Mar del Plata, solamente se ha comunicado con la Florista de España y desconoce la forma en que ésta solucionará su pedido.

Desde el punto de vista de la Florista de España, solamente confecciona el pedido, se comunica con el Transportista y desconoce la forma en que el transportista solucionará el envío de su pedido.

El transportista, lleva su encomienda a destino, desconociendo su origen (pedido generado desde Mar del Plata) y tal vez hasta su

Mirando la realidad como POO.

Ejemplo

El transportista, lleva su encomienda a destino, desconociendo su origen (pedido generado desde Mar del Plata) y tal vez hasta su contenido.

Ambas Floristas tienen algo en común, su actividad. O sea, su comportamiento es similar, son “objetos” que pertenecen a una misma clase (la clase de los Floristas).

El Transportista pertenece a otra clase distinta.

La forma en que cada uno de estos objetos lleva a cabo su actividad, es desconocida por los otros objetos, por ejemplo: cuántas flores posee en stock cada florista, la cantidad de móviles del transportista, la ruta empleada por el transportista, etc. A estos datos o actividades se los denomina **privados**, y a los demás **públicos**. Por ejemplo, el mensaje **enviarFlores** es de dominio público, ya que es un método que se usa para pasar mensajes entre objetos distintos (el actor a la florista de Mdp y a su vez, la florista de Mdp a la de España).

Este mismo tipo de enfoque y análisis de un ejemplo de la vida real, se pretende trasladar al análisis y resolución de un problema computacional.

Un ejemplo simple en C++

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

```
class cola
```

```
{
```

```
    int c[100];
```

```
    int ppio, fin;
```

```
public:
```

```
    void ini();
```

```
    void meter(int i);
```

```
    int sacar(void);
```

```
};
```

```
void cola::ini()
{
    fin = ppio = 0;
}
```

```
void cola::meter(int i)
{
    if(ppio==100)
    {
        cout<< "la cola esta llena";
    }
    else
    {
        ppio++;
        c[ppio] = i;
    }
}
```

Los Objetos

- **¿Por qué Orientación a Objetos (OO)?**
 - Se parece más al mundo real
 - Permite representar modelos complejos
 - Muy apropiada para aplicaciones de negocios
 - Las empresas ahora sí aceptan la OO
 - Las nuevas plataformas de desarrollo la han adoptado (Java / .NET)

¿Qué es un Objeto?

- Informalmente, un objeto representa una entidad del mundo real
- **Entidades Físicas**
 - (Ej.: Vehículo, Casa, Producto)
- **Entidades Conceptuales**
 - (Ej.: Proceso Químico, Transacción Bancaria)
- **Entidades de Software**
 - (Ej.: Lista Enlazada, Interfaz Gráfica)

¿Qué es un Objeto?

- **Definición Formal (Rumbaugh):**
 - “Un objeto es un concepto, abstracción o cosa con un significado y límites claros en el problema en cuestión”
- **Un objeto posee (Booch):**
 - Estado
 - Comportamiento
 - Identidad

Un objeto posee Estado

Lo que el objeto sabe

- El estado de un objeto es una de las posibles condiciones en que el objeto puede existir
- El estado normalmente cambia en el transcurso del tiempo
- El estado de un objeto es implementado por un conjunto de propiedades (atributos), además de las conexiones que puede tener con otros objetos

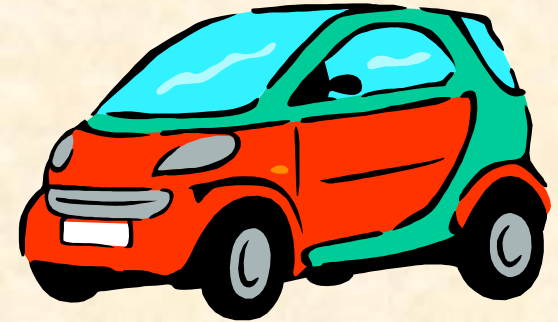
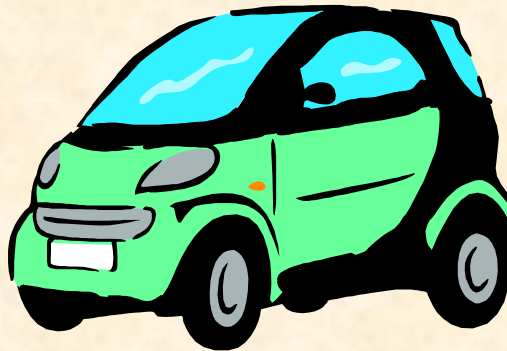
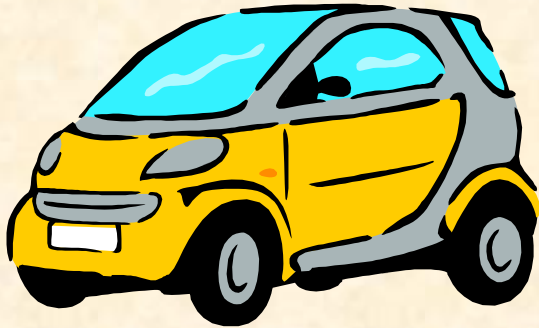
Un objeto posee Comportamiento

Lo que el objeto puede hacer

- El comportamiento de un objeto determina cómo éste actúa y reacciona frente a las peticiones de otros objetos
- Es modelado por un conjunto de mensajes a los que el objeto puede responder (operaciones que puede realizar)
- Se implementa mediante métodos

Un objeto posee Identidad

- Cada objeto tiene una identidad única, incluso si su estado es idéntico al de otro objeto

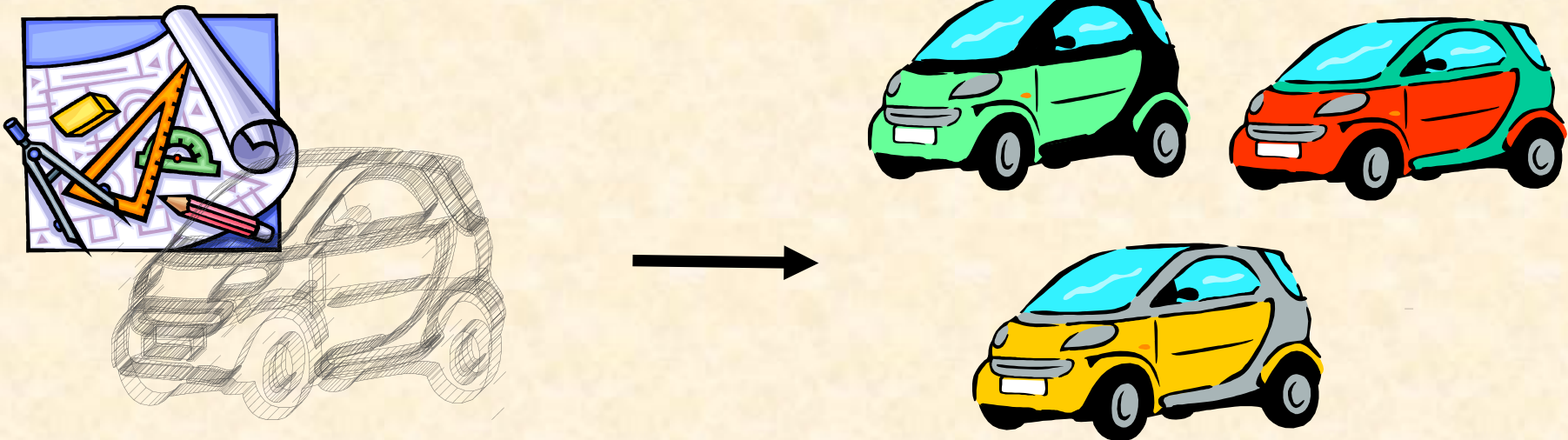


¿Qué es una Clase?

- Una clase es una descripción de un grupo de objetos con:
 - Propiedades en común (atributos)
 - Comportamiento similar (operaciones)
 - La misma forma de relacionarse con otros objetos (relaciones)
 - Una semántica en común (significan lo mismo)
- Una clase es una abstracción que:
 - Enfatiza las características relevantes
 - Suprime otras características (simplificación)
- **Un objeto es una instancia de una clase**

Objetos y Clases

- Una clase es una definición abstracta de un objeto
 - Define la estructura y el comportamiento compartidos por los objetos
 - Sirve como modelo para la creación de objetos
- Los objetos pueden ser agrupados en clases



- La orientación a objetos es una manera natural de pensar acerca del mundo y de escribir programas de computadora.
- Los objetos tienen atributos (como tamaño, forma, color, peso y aspecto) y exhiben un comportamiento.
- Los humanos aprendemos acerca de los objetos mediante el estudio de sus atributos y la observación de su comportamiento.
- Objetos distintos pueden tener muchos atributos iguales y exhibir un comportamiento similar.
- La programación orientada a objetos (POO) modela objetos del mundo real con contrapartes en software.
- Toma ventaja de las relaciones entre clases, en donde los objetos de cierta clase tienen las mismas características y comportamientos. Aprovecha las relaciones de herencia e incluso las relaciones de herencia múltiple, en donde las clases recién creadas se derivan mediante la herencia de las características de clases existentes, aunque contienen características únicas propias.

La programación orientada a objetos proporciona una manera intuitiva de ver el proceso de programación, a saber, mediante el modelado de objetos reales, sus atributos y sus comportamientos.

La POO también modela la comunicación entre objetos mediante mensajes.

La POO encapsula los datos (atributos) y las funciones (comportamiento) dentro de los objetos.

Los objetos tienen la propiedad de ocultar información. Aunque los objetos pueden saber como comunicarse entre sí a través de interfaces bien definidas, los objetos por lo general no están autorizados para saber los detalles de la implementación de otros objetos (para eliminar dependencias innecesarias).

El ocultamiento de información es crucial para una buena ingeniería de software.

En C y en otros lenguajes de programación por procedimientos, la programación tiende a ser orientada a acciones. En realidad, los datos son importantes en C, pero la visión es que los datos existen primordialmente para permitir las acciones que realizan las funciones.

Los programadores en C++ se concentran en la creación de sus propios tipos definidos denominados clases. Cada clase contiene datos, así como el conjunto de funciones que manipulan los datos. Los componentes de datos de una clase se denominan datos miembro. Los componentes de función de una clase se denominan funciones miembro o métodos.