Estructuras Compuestas

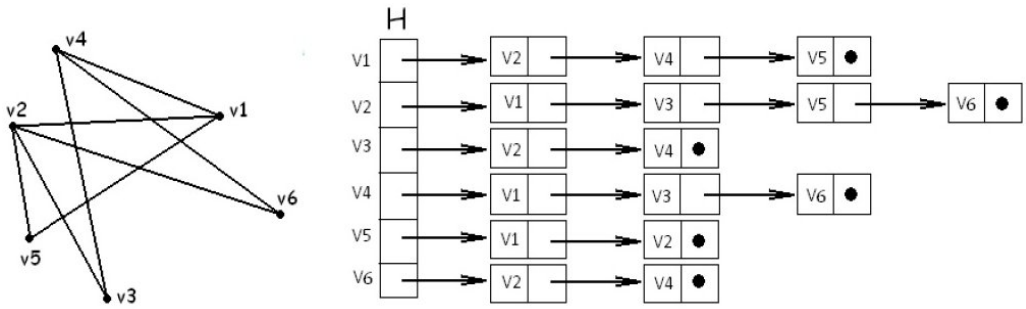
Son estructuras que permiten representar datos que resultan de la composición de estructuras más simples. Son de gran utilidad cuando se deben resolver problemas complejos que involucran la utilización de una gran variedad de tipos.

Las composiciones más frecuentes son:

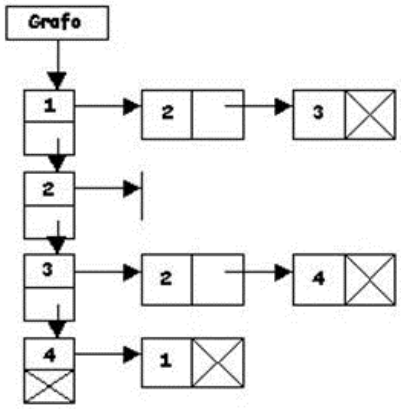
1. Arreglo de Listas.
2. Arreglo de Arboles.
3. Listas de Listas.
4. Listas de Arboles.
5. Arboles de Listas.

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 dónde 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
        }
    }
}


/*** MAIN *********************************************************************/
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\n¡Fin 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.