Universidad del Valle
Facultad de Ingeniería
Escuela de Ingeniería de Sistemas y Computación
Fundamentos de Programación
Víctor Bucheli, Ph. D.
Giovanny Hidalgo, Ms.C.
Andrés Castillo, Ph.D.

***Este documento se basa en los capítulos 14 y 15 del libro how to design programs. Es realizado sólo con fines académicos para el curso de Fundamentos de programación de la Escuela de Ingeniería de Sistemas y Computación*.**

Más definiciones auto-referenciales de datos

Muchas clases de datos pueden requerir definiciones más complejas que las definiciones auto-referenciales de listas. Ahora estudiaremos cómo formular tipos de datos propios. Iniciamos por las descripciones informales de la información y después se puede seguir la receta de diseño de datos auto-referenciales ligeramente modificada.

Estructuras en estructuras

En el curso de Fundamentos de Programación (FP) se ha llevado a cabo dos juegos de programadores y novatos, en los cuales se puede observar una red o lo que es lo mismo una estructura dentro de otra estructura. Esto es como una red, similar a la presentada en la siguiente figura, donde cada estudiante es un nodo y se relaciona con otro nodo, la red se construye al momento de que cada Programador enseña a otro estudiante y se crea un vínculo entre programadores y novatos. Cuando se acumulan muchas de estas relaciones se construye una red, y esto en JS puede ser representado como una estructura de estructuras donde cada nodo es una estructura y las relaciones que tiene son nodos también.

De una forma similar se puede ver un árbol genealógico, el cual es una estructura dentro de otra estructura, así son similares las redes, los árboles genealógicos o árboles familiares. Los árboles genealógicos son utilizados para estudiar características heredadas, por ejemplo, buscar en un árbol genealógico un cierto color de ojos o la prevalencia de una enfermedad hereditaria. De igual forma en la red de programadores y novatos se puede buscar los programadores que ayudan más a los compañeros o cuales son los programadores más centrales en la red que apoyan en mayor medida la actividad.

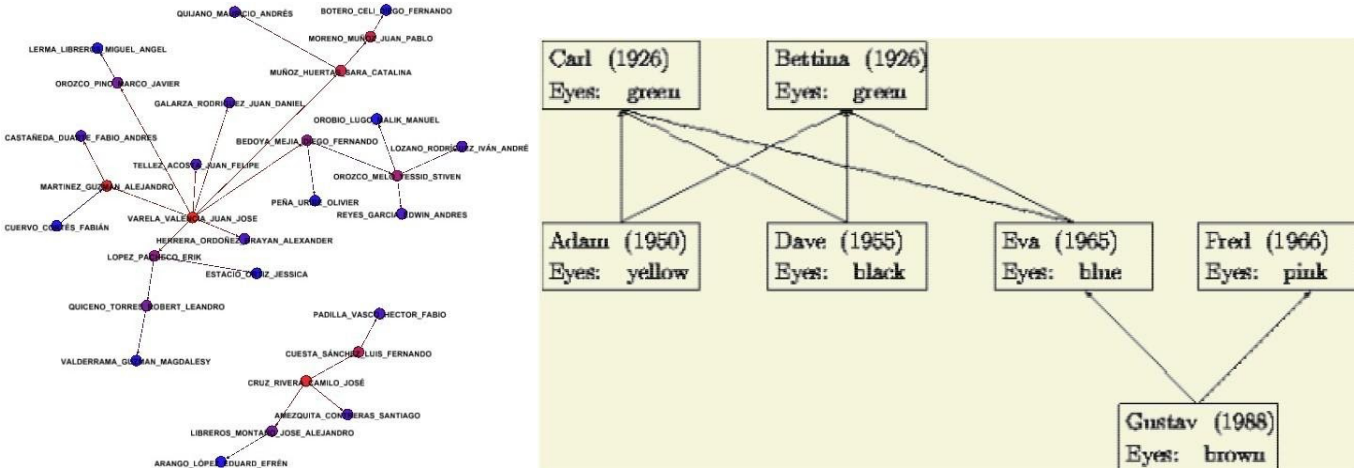


Figura 1: Red de programadores y novatos y Árbol genealógico

El cómputo de una estructura de estructuras se puede realizar en JavaScript, de ahí que sea natural representar redes de aprendizaje o árboles familiares como estructuras de estructuras. Usted puede desarrollar funciones para realizar algún procesamiento con dichas estructuras, por ejemplo, desarrollar una función que permita saber qué estudiantes de FP sacan una nota superior a 4,5 o busca cuantos de los padres en un árbol genealógico tienen ojos azules.

Una manera de crear una red de aprendizaje es crear un nodo cuando un estudiante de FP resuelve el reto e informa que otro estudiante fue quien le enseñó (**Figura 1**). Similarmente, una forma de construir un árbol familiar es añadir un nodo al árbol cada vez que nace un niño (**Figura 1**). Las conexiones entre ellos son inmediatas, por ejemplo, la conexión entre Juan Jose Varela y Sara Catalina Huertas, o las conexiones entre Adam, Carl y Bettina.

Para saber si los conceptos se están entendiendo desarrollemos las siguientes actividades:

1. **Identifique dos relaciones entre Programador líder y Programador novato en la red del juego.**
2. **Identifique dos relaciones Padre e hijo en el árbol genealógico.**
3. **Identifique dos nodos que no tienen relación en la red del juego.**

Para el caso del árbol genealógico se pueden registrar más información, como, por ejemplo: la fecha de nacimiento, el peso al nacer, el color de los ojos, o el color del pelo, entre otras características.

Ejercicio: Adam es el hijo de Bettina y Carl, tiene ojos amarillos, y nació en 1950. De forma similar Gustav es el hijo de Eva y Fred, tiene ojos cafés, y nació en 1988. Para representar un niño en el árbol familiar se necesita combinar diversas piezas de información: el padre, la madre, el nombre, la fecha de nacimiento, y el color de los ojos.

Los cinco campos de estructuras ***hijo*** registran la información necesaria, la cual sugiere la siguiente definición de datos:

La estructura ***hijo***:

```
const hijo: {
      padre:hijo o null,
      madre:hijo o null,
      nombre: "",
      fecha: "",
      ojos: ""
   };
```

Donde ***padre*** y ***madre*** son estructuras hijo; ***nombre, fecha*** y ***ojos*** son texto. Esta definición permite crear sólo un nodo en la red. Para representar árboles familiares, suponga que se requiere añadir un ***hijo*** a un árbol familiar existente, que además se tienen ya representaciones de los padres. Por ejemplo, para Adam se podría crear la siguiente estructura, suponiendo que *Carl* y *Bettina* representan a los padres de Adam:

```
const Adam: {
      padre:Carl,
      madre: Bettina,
      nombre: "Adam",
      fecha: "1950",
      ojos: "yellow"
   };
```

El problema es que no siempre se conocen los padres de una persona. En la familia representada en el árbol genealógico, no se conocen los padres de Bettina. Por lo que a pesar de que no se conozcan el padre o la madre de una persona, se debe emplear algún valor. En este caso utilizamos **null**. Por ejemplo, para construir una estructura hijo para Bettina, se hace lo siguiente:

```
const Bettina = {
  padre: null,
  madre: null,
  nombre: "Bettina",
  fecha: "1926",
  ojos: "green"
};
```

El análisis sugiere que un nodo ***hijo*** tiene la siguiente definición de datos:

1. ***padre*** y ***madre*** son cualquiera de
    a. **null**
    b. nodos *hijo*;
2. ***nombre, fecha*** y ***ojos*** son símbolos;

Actividad: Construya la estructura para el juego programadores y novatos y adicione la nota y un campo booleano que dice si se divirtió o no en el juego. Hágalo específicamente para Juan Perea que recibió colaboración de Pedro Palomino y que las notas son 4 y 4.5 respectivamente los dos se divirtieron.

Un ***nodo-árbol-familiar*, *naf*,** es cualquiera

1. **null**, o
2. *hijo (padre madre nombre fecha ojos*)

   Donde ***padre*** y ***madre*** son ***naf*s**, ***nombre, fecha*** y ***ojos*** son texto.

Con base en esta definición el árbol genealógico se puede definir de la siguiente manera:

La información para Carl es fácil de traducir a una estructura.

```
const Carl = {
  padre: null,
  madre: null,
  nombre: "Carl",
  fecha: "1926",
  ojos: "green"
};
```

Bettina y Fred se representan con nodos similares. A su vez, el nodo para Adam se crea con:

```
const Adam = {
  padre: Carl,
  madre: Bettina,
  nombre: "Adam",
  fecha: "1950",
  ojos: "yellow"
}
```

Como muestran los ejemplos, es una correspondencia simple **nodo-por-nodo** del árbol genealógico y llamadas recursivas a la estructura. Por ejemplo, si se construye la estructura hijo para Dave como la de Adam, se obtiene:

Padre

```
const Carl = {
  padre: null,
  madre: null,
  nombre: "Carl",
  fecha: "1926",
  ojos: "green"
};
```

Madre

```
const Bettina = {
  padre: null,
  madre: null,
  nombre: "Bettina",
  fecha: "1926",
  ojos: "green"
};
```

```
const Dave = {
  padre: Carl,
  madre: Bettina,
  nombre: "Adam",
  fecha: "1955",
  ojos: "black"
}
```

De ahí que sea una buena idea introducir una definición de variable por nodo y emplearla después. Para facilitar las cosas, se emplea Carl para representar la estructura hijo que describe Carl, etc. El árbol completo en se define de la siguiente forma:

```
//Generación mayor
const Carl = { padre: null, madre: null, nombre: "Carl", fecha: "1926", ojos: "green"};
const Bettina = { padre: null, madre: null, nombre: "Bettina", fecha: "1926", ojos: "green"};

//Generación intermedia
const Adam = { padre: Carl, madre: Bettina, nombre: "Adam", fecha: "1950", ojos: "yellow"};

const Dave = { padre: Carl, madre: Bettina, nombre: "Adam", fecha: "1955", ojos: "black"};

const Eva = { padre: Carl, madre: Bettina, nombre: "Eva", fecha: "1965", ojos: "blue"};

const Fred = { padre: null, madre: null, nombre: "Fred", fecha: "1966", ojos: "pink"};

//Generación más joven
const Gustav = { padre: Fred, madre: Eva, nombre: "Gustav", fecha: "1988", ojos: "brown"};
```

ACTIVIDAD: Escriba la estructura *naf* correspondiente para los 6 estudiantes de la red del juego programadores y novatos que se encuentra a la derecha de la figura 1.

Funciones de estructuras en estructuras

Para las estructuras dentro de estructuras, se construyen funciones para este tipo de datos. Puesto que la definición de datos ***nafs*** contiene dos cláusulas, una que sea un nodo hijo u otra que sea un nodo solamente, caso en el cual padre o madre son nulos. La función debe contener una expresión ***if***. La primera opera con padre == ***null*****,** la segunda con estructuras ***hijo***:

```
function UnaFuncion(tree) {
   if (tree.padre == null) return true;
   else return false;
}
```

En la función anterior para la primera instrucción, la entrada sólo necesita evaluar y no hay ninguna tarea adicional. Para la segunda instrucción ***else***, la entrada contiene cinco piezas de información: dos nodos de árbol familiar adicionales padre o madre, el nombre de la persona, la fecha de nacimiento, y el color de ojos. Esto es aplicar UnaFuncion a los campos del *padre* y la *madre* debido a su **auto-referencia**.

Ahora se va a desarrollar el paso a paso de una función para saber si hay ancestros de ojos azules en el árbol. Siguiendo la receta, se desarrollan algunos ejemplos. Considérese el nodo del árbol familiar para Carl. No tiene ojos azules, y debido a que no tiene ningún ancestro conocido en el árbol familiar, el árbol familiar representado por dicho nodo no contiene una persona con ojos azules. De ahí que:

>(isBlue Carl) devuelve ***false***.

En cambio, el árbol familiar representado por *Gustav* contiene un nodo para Eva quien tiene ojos azules.

>(isBlue *Gustav*) devuelve ***true***

El formato de la función es como el de UnaFuncion, se emplea el formato para orientar el diseño de la función. Primero se supone que se presenta (*un-árbol vacío*), en cuyo caso, el árbol familiar es ***vacío***, y nadie tiene ojos azules. De ahí que la respuesta deba ser ***false***.

La segunda cláusula del formato contiene varias expresiones que requieren interpretación:

1. (isBlue (*hijo-padre un-árbol*))**:** la cual determina si alguien en el ***naf*** del padre tiene ojos azules.
2. (isBlue (*hijo-madre un-árbol*)), la cual determina si alguien en el ***naf*** de la madre tiene ojos azules.
3. (*hijo-nombre un-árbol*), extrae el nombre del ***hijo***.
4. (*hijo-fecha un-árbol*), extrae la fecha de nacimiento del ***hijo***.
5. (*hijo-ojos un-árbol*), obtiene el color de los ojos del ***hijo y se evalúa si son azules***.

Es evidente que si la estructura ***hijo*** contiene *"**blue**"* en el campo ***ojos***, la función responde ***true***, o si hay alguna persona en el árbol del padre o de la madre. El resto de los casos son ***false***.

Ese análisis sugiere formular una expresión condicional y que su primera condición es:

if(tree.ojos == "blue")

Las dos recursiones son dos condiciones adicionales. Si alguna produce ***true***, la función produce ***true*****;** *en caso contrario* genera ***false***.

En resumen, la respuesta en la segunda cláusula es la expresión:

```
function isBlue(tree) {
    if (tree.padre == null)
        return false;
    else {
        if (tree.ojos == "blue" || isBlue(tree.madre) || isBlue(tree.padre)) return true;
        else return false;
    }
}
```

ACTIVIDAD: Desarrollar *contar-personas*, consume un *naf* y genera el número de personas en el correspondiente árbol familiar.

```
const isLeaf = tree => {
    if (!tree.padre && !tree.madre) {
        return true
    } else {
        return false
    }
}

function coutleaf(tree) {
    if (tree.padre == null) {
        return 0
    } else if (isLeaf(tree)) {
        return 1
    } else {
        return 1 + coutleaf(tree.padre) + coutleaf(tree.madre)
    }
}
```

ACTIVIDAD: Desarrollar *edad-promedio*, consume un *naf* y el año actual, y genera la edad promedio de todos los individuos de un árbol familiar.

```
function contar(tree) {
    if (tree.padre == null) return 1;
    else {
        return 1 + contar(tree.madre) + contar(tree.padre);
    }
}

function promedio(tree, year) {
    if (tree.padre == null) return (year - tree.fecha);
    else {
        return year - tree.fecha + promedio(tree.madre, year) + promedio(tree.padre, year);
    }
}
const promedioEdad = (tree, year) => promedio(tree, year) / contar(tree);
```

Árboles de búsqueda binaria

Los árboles de búsqueda binaria se emplean en muchas aplicaciones orientadas a almacenar y recuperar información. En este contexto, un árbol de búsqueda binaria es parecido a un árbol genealógico que en lugar de estructuras hijo contiene ***nodos***:

```
const nodo = {
  nss: "",
  izquierdo: "",
  derecho: ""
};
```

Donde la estructura define un nodo el cual tiene un ***ns*** que es un número y otros dos árboles; tal como si fueran campos parentales de árboles familiares, en los cuales cada nodo tiene un árbol que crece por la izquierda y por la derecha. La correspondiente definición de datos es parecida a la de los árboles familiares:

Un ***árbol binario***, ***AB***, es cualquiera de

1. **false**, o
2. (*nodo ns izq der*)
   donde ***ns*** es un número, e ***izq*** y ***der*** son ***AB*s**.

```
const nodo = {
  ns: “”,
  izquierdo: "",
  derecho: ""
};
```

En los árboles binarios existe la opción de ***false*** para indicar null. Se pudo utilizar null nuevamente, pero ***false*** es generalmente utilizado.

Ejemplos de árboles binarios:

ACTIVIDAD: Explique la siguiente estructura de árbol binario.

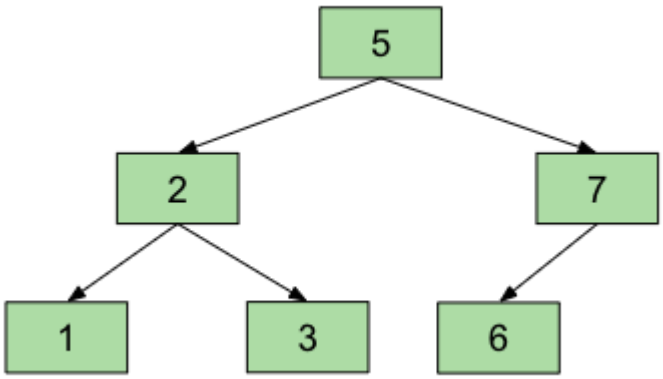
```
const nodo15 = {
  ns: "15",
  izquierdo: false,
  derecho: false
};
```

ACTIVIDAD: Explique la siguiente estructura de árbol binario.

```
const nodo24 = {
  ns: "24",
  izquierdo: false,
  derecho: false
};
```

La siguiente figura muestra una representación de árbol binario, que se construyen hacia abajo, o sea, la raíz tiene el valor medio y las ramas hacia abajo contienen valores menores por izquierda y mayores por derecha. Cada valor corresponde a un nodo, etiquetados con su campo ***ns***, en la representación se omite ***false***.

Árbol A



Estructura del Árbol A

```
{
    ns: 5,
    left: {
        ns: 2,
        left: {
            ns: 1
        },
        right: {
            ns: 3
        }
    },
    right: {
        ns: 7,
        left: {
            ns: 6
        }
    }
}
```

La secuencia del árbol ***A*** está ordenada de forma ascendente. Un árbol binario ***AB*** que tiene una secuencia ordenada de información es un ***ÁRBOL DE BÚSQUEDA BINARIA ABB***.
Un *árbol de búsqueda binaria*, ***ABB***, es un ***AB***

1. **false** es siempre un ***ABB***, y
2. (*nodo ns izq der*) es un ***ABB*** si
    1. ***izq*** y ***der*** son *ABB*s,
    2. los ***nss*** en ***izq*** son menores que ***ns***, y
    3. los ***nss*** en ***der*** son mayores que ***ns***.

Las condiciones 2 y 3 son diferentes de cualquier definición de datos vista hasta ahora. Ya que ubican un obstáculo adicional en la construcción de ***ABBs***. Ahora se requiere inspeccionar todos los números en dichos árboles y garantizar que son menores (o mayores) que ***ns***.

ACTIVIDAD: Desarrollar la función en orden, la cual consume un AB y genera una lista de los nss en el árbol, la lista contiene los números de izquierda a derecha. Para el caso particular árbol A devuelve los números: **1, 2, 3, 5, 6, 7**

```
function AuxinOrder(node) {
    if (node) {
        AuxinOrder(node.left);
        console.log(node.ns);
        AuxinOrder(node.right);
    }
}
```

ACTIVIDAD: Emplear la función ***append***, la cual concatena listas, y a partir de la función inorder crear una lista de números.

ACTIVIDAD: desarrollar el postorder (left, rigth, root) del árbol y devolver una lista de números.