****

**Universidad de Costa Rica**

**Facultad de Ingeniería**

**Escuela de Computación e Informática**

Estructuras de Datos y Análisis de Algoritmos

CI-1221

Grupo 001-002

**III ETAPA – I TAREA PROGRAMADA**

**Profesora:**

Sandra Kikut

**Elaborado por:**

Divney Quirós Daniel B32303

Gómez Montero Andrea B32896

Sagasti Charpentier Amanda B36269

Lunes 27 de octubre del 2014

**Tabla de contenidos mínimos**

1. Introducción 4

2. Objetivos 4

3. Enunciado 4

4. Desarrollo 6

4.1 Modelos 6

4.1.1 Modelo pila 6

4.1.2 Modelo cola 7

4.1.3 Modelo árbol tal que no importa el orden entre los hijos 8

4.2 Estructura de datos 10

4.2.1 Lista simplemente enlazada para una pila 10

4.2.2 Lista simplemente enlazada con puntero al último lleno para una cola 13

4.2.3 Lista de hijos por lista simplemente enlazada (lista principal) y lista simplemente enlazada (sublistas) para un árbol tal que no importa el orden entre los hijos 14

4.2.4 Lista de hijos por arreglo (lista principal) y lista simplemente enlazada (sublistas) para un árbol tal que no importa el orden entre los hijos 15

4.2.5 Hijo Más Izquierdo-Hermano Derecho por punteros para un árbol tal que no importa el orden entre los hijos 16

4.2.6 Hijo Más Izquierdo-Hermano Derecho por punteros con puntero al padre y al hermano izquierdo para un árbol tal que no importa el orden entre los hijos 17

4.2.7 Hijo Más Izquierdo-Hermano Derecho por punteros tal que el último hijo de un nodo apunta al padre para un árbol tal que no importa el orden entre los hijos 18

4.3 Algoritmos 19

4.3.1 HermanoIzquierdo(TipoÁrbol a, entero HD) 19

4.3.2 AlturaNodo(tipoNodo n) 20

4.3.3 etiquetasRepetidas(tipoÁrbol a) 21

4.3.4 nivelesArbol(tipoÁrbol A) 23

4.3.5 hijosNodo(tipoÁrbol A, tipoNodo n) 23

4.3.6 listarHijosNodo(Árbol \*a, tipoNodo n) 23

4.3.7 borrarSubÁrbol(tipoÁrbol A) 24

4.3.8 copiarArbol(tipoÁrbol A) 25

4.3.9 revisaArbolesIguales(tipoÁrbol A, tipoÁrbol B) 26

4.3.10 PreOrden(tipoÁrbol A) 27

4.3.11 PostOrden(tipoÁrbol A) 27

4.3.12 InOrden(tipoÁrbol A) 28

4.3.13 listadoNiveles(tipoÁrbol A) 28

4.3.14 ListadoRecursivoPila() 29

4.3.15 encuentraNodo(tipoEtiqueta e, tipoÁrbol A) 29

5. Manual del usuario  30

5.2. Requerimientos de software  30

5.3. Arquitectura del programa  30

5.4. Compilación  30

5.5. Especificación de las funciones del programa 30

6. Datos de Prueba  30

6.1. Formato de los datos de prueba  30

6.2. Salida esperada  31

6.3. Salida obtenida (Análisis en caso de fallo) 31

8. Listado de archivos 31

9. Referencias 31

# 1. Introducción

Los modelos Pila, Cola y Árbol son algunos de los modelos más básicos utilizados en la computación. Es muy importante tomarlos en cuenta debido a la gran cantidad de implementaciones en las que se ha usado, tanto así que es indispensable conocerlos para comprender programas complejos ya existentes y crear los propios. Y es por eso que en la Tarea Programada I es de mucha importancia definir y especificar a estos modelos y sus operadores básicos con el fin de poder implementarlos en un futuro al diseñar y crear estructuras de datos que los contengan o al simplemente implementar alguna de estas estructura de datos individualmente.

# 2. Objetivos

Definir, especificar, implementar y usar los modelos Pila, Cola y Árbol n-ario tal que NO importa el orden entre los hijos de un nodo.

Realizar un análisis teórico y un análisis real del tiempo de ejecución de las diferentes estructuras de datos y algoritmos utilizados.

# 3. Enunciado

Se definen formalmente los modelos Pila, Cola y Árbol n-ario tal que no importa el orden entre los hijos de un nodo. Para cada modelo, se especifica de manera lógica, formal y completa los operadores básicos. Cada operador incluye: nombre, parámetros con sus tipos y las cláusulas: Efecto, Requiere y Modifica.

Se implementan los siguientes modelos:

|  |  |
| --- | --- |
| Modelo | Estructura de datos |
| 4.1.1 Pila | Lista simplemente enlazada |
| 4.1.2 Cola | Lista simplemente enlazada con puntero al último |
| 4.1.3 Árbol n-ario tal que no importa el orden entre los hijos | Lista de hijos por lista simplemente enlazada (lista principal) y lista simplemente enlazada (sublistas) |
|  | Lista de hijos por arreglo(lista principal) y lista simplemente enlazada (sublistas) |
|  | Hijo más izquierdo-Hermano derecho por punteros |
|  | Hijo más izquierdo-Hermano derecho por punteros, con puntero al padre y al hermano izquierdo |
|  | Hijo más izquierdo-Hermano derecho por punteros, tal que el último hijo de un nodo apunta al padre. |

Para probar estos modelos, se hace un programa de prueba que permita verificar que todas las implementaciones de las estructuras de datos del Cuadro 1 están correctas. El programa muestra un menú que permite al usuario utilizar diferentes operadores básicos de cada modelo.

# 4. Desarrollo

## 4.1 Modelos

### 4.1.1 Modelo pila

#### 4.1.1.1 Definición modelo pila

Una pila es un tipo especial de lista en la cual las inserciones y los borrados se hacen en un solo extremo llamado el “tope.” También se llama “LIFO”, que quiere decir “last-in first-out”, ya que el último elemento agregado es el primero en salir de la lista. Un ejemplo intuitivo de una pila es una pila de platos en un estante, una torre de chips de poker, o cualquier otra situación en la cual es más conveniente quitar el último elemento de primero. (Aho, 1983, p. 16)

#### 4.1.1.2 Definición y especificación de los operadores básicos del modelo pila

Crear(tipoPila P)

Efecto: Inicializa en memoria un contenedor tipo pila vacío

Requiere: P no esté inicializado

Modifica: P

Destruir(tipoPila P)

Efecto: Elimina lógicamente a P de la memoria

Requiere: P inicializada

Modifica: P

Vaciar(tipoPila P)

Efecto: Vacía a P de contenido

Requiere: P inicializada

Modifica: P

Vacía(tipoPila P)

Devuelve: Un booleano

Efecto: Devuelve verdadero si P está vacía y falso si no

Requiere: P inicializada

Modifica: N/A

Agregar(tipoElemento e, tipoPila P)

Efecto: Agrega el elemento e a la pila P

Requiere: P inicializada

Modifica: P

Sacar(TipoPila P)

Devuelve: TipoElemento e

Efecto: Retorna e, el último elemento de la pila P, y lo borra de P

Requiere: P inicializada y no vacía

Modifica: P

Tope(TipoPila P)

Devuelve: TipoElemento e

Efecto: Retorna e, el último elemento de la pila P, sin borrarlo

Requiere: P inicializada y no vacía

Modifica: N/A

### 4.1.2 Modelo cola

#### 4.1.2.1 Definición modelo cola

Una cola es un tipo especial de lista donde los elementos se insertan en un extremo y se eliminan en el otro. El extremo donde se eliminan los elementos se llama “frente” de la cola. Otro nombre para la cola es “FIFO”, que quiere decir “first-in first-out”, ya que el primer elemento insertado es el primero en ser eliminado. Un ejemplo de una cola es una fila de un banco, o cualquier otra situación de la vida real en la cual es más conveniente entrar de primero para salir de primero. Las operaciones de una cola son análogas a las de una pila con una única diferencia: las inserciones se hacen al final de la lista en lugar de al principio. (Aho, 1983, p.19)

#### 4.1.2.2 Definición y especificación de los operadores básicos del modelo cola

Crear(tipoCola C)

Efecto: inicializa en memoria un contenedor de tipo cola vacío

Requiere: C no inicializada

Modifica: C

Destruir(tipoCola C)

Efecto: Elimina lógicamente a C de la memoria

Requiere: C inicializada

Modifica: C

Vaciar(tipoCola C)

Efecto: Vacía a C de contenido

Requiere: C inicializada

Modifica: C

Vacía(tipoCola C)

Devuelve: Un booleano

Efecto: Devuelve verdadero si C está vacía y falso si no

Requiere: C inicializada

Modifica: N/A

Agregar(tipoElemento e, tipoCola C)

Efecto: Agrega el elemento e a la cola C

Requiere: C inicializada

Modifica: C

Sacar(tipoCola C)

Devuelve: TipoElemento e (opcional)

Efecto: Elimina el primer elemento de C, y opcionalmente lo devuelve

Requiere: C inicializada y no vacía

Modifica: C

Frente(tipoCola C)

Devuelve: TipoElemento e

Efecto: Retorna e, el primer elemento de la cola C

Requiere: C inicializada y no vacía

Modifica: N/A

### 4.1.3 Modelo árbol tal que no importa el orden entre los hijos

#### 4.1.3.1 Definición árbol tal que no importa el orden entre los hijos

Un árbol es una colección de elementos llamados nodos que tienen una relación de jerarquía entre sí. Uno de esos nodos se distingue como la raíz del árbol. Un nodo, como un elemento de una lista, puede ser de cualquier tipo. Usualmente un nodo es un caracter, una hilera o un número. (Aho, 1983, p. 91)

Formalmente un árbol se puede definir como:

1. Un nodo es en sí solo un árbol. Ese nodo también es la raíz del árbol.
2. Sea n un nodo y T1, T2,…, Tk árboles con raíces n1, n2, . . ., nk respectivamente. Podemos construir un nuevo árbol al hacer que n sea el padre de los nodos n1, n2,…, nk. En este nuevo árbol n es la raíz y T1, T2,…, Tk son subárboles de esa raíz. Los nodos n1, n2,…, nk son los hijos del nodo n.

(Aho, 1983, p. 91)

Los nodos de un árbol usualmente se ordenan de izquierda a derecha, pero en este caso el orden entre los hijos no importa. Eso quiere decir que el árbol (dibujo) y (dibujo) son el mismo. (Aho, 1983, p. 93)

#### 4.1.3.2 Definición y especificación de los operadores básicos del modelo árbol tal que no importa el orden entre los hijos

Crear (tipoÁrbol A)

Efecto: Inicializa en memoria un contenedor de tipo árbol vacío

Requiere: A no inicializado

Modifica: A

Destruir (tipoÁrbol A)

Efecto: Elimina lógicamente a A de la memoria

Requiere: A inicializado

Modifica: A.

Vaciar(tipoÁrbol A)

Efecto: Vacia a A de contenido

Requiere: A inicializado

Modifica: A.

Vacío(tipoÁrbol A)

Devuelve: Un booleano

Efecto: Devuelve verdadero si A está vacío y falso si no

Requiere: A inicializado

Modifica: N/A

PoneRaíz(tipoEtquieta e, tipoÁrbol A)

Efecto: Crea un TipoNodo n que contiene la etiqueta e, y lo establece como la raíz.

Requiere: A inicializado y vacío, y e válida en A.

Modifica: A.

AgregaHijo(tipoNodo n, tipoEtiqueta e, tipoÁrbol A)

Efecto: Agrega un nodo con la etiqueta e como hijo del nodo n en el árbol A

Requiere: A inicializado, n válido en A

Modifica: A

BorraHoja(tipoNodo n, tipoÁrbol A)

Efecto: Elimina n del árbol A

Requiere: A inicializado, n válido en A y que n sea una hoja

Modifica: A

ModificaEtiqueta(tipoNodo n, tipoEtiqueta e, tipoÁrbol A)

Efecto: Actualiza la etiqueta de n a e

Requiere: A inicializado, n válido en A.

Modifica: A

Raíz(tipoÁrbol A)

Devuelve: tipoNodo

Efecto: Devuelve el nodo raíz de A.

Requiere: A inicializado y no vacío

Modifica: N/A

Padre(tipoNodo n, tipoÁrbol A)

Devuelve: tipoNodo

Efecto: Devuelve el nodo padre de n.

Requiere: A inicializado y n válido en A

Modifica: N/A

HijoMásIzquierdo(tipoNodo n, tipoÁrbol A)

Devuelve: tipoNodo

Efecto: Devuelve el nodo hijo más izquierdo de n

Requiere: A inicializado y n válido en A

Modifica: N/A

HermanoDerecho(tipoNodo n, tipoÁrbol A)

Devuelve: tipoNodo

Efecto: Devuelve el nodo hermano derecho de n

Requiere: A inicializado y n válido en A

Modifica: N/A.

Etiqueta(tipoNodo n, tipoÁrbol A)

Devuelve: tipoEtiqueta

Efecto: Devuelve la etiqueta de n

Requiere: A inicializado y n válido en A

Modifica: N/A

EsHoja(tipoNodo n, tipoÁrbol A)

Devuelve: tipoBooleano

Efecto: Devuelve verdadero si n no tiene hijos y falso si no

Requiere: A inicializado y n válido en A

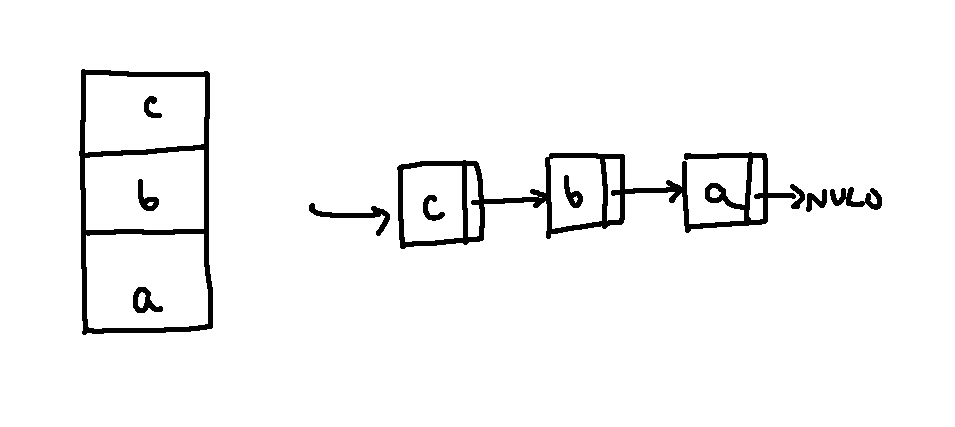
Modifica: N/A

## 4.2 Estructura de datos

### 4.2.1 Lista simplemente enlazada para una pila

#### 4.2.1.1 Diagrama y descripción de lista simplemente enlazada para una pila

Figura 1: Diagrama del modelo e implementación de una lista simplemente enlazada para una pila.



La estructura de datos lista simplemente enlazada se puede utilizar para representar el modelo pila. El primer elemento en la lista es el tope, que es el último en entrar y el primero en salir.

#### 4.2.1.2 Definición en C++ de lista simplemente enlazada para una pila

template <class T>

class pila{

private:

struct nodo{

T contenido;

nodo \*siguiente;

};

nodo \*cabeza = new nodo;

public:

pila();

~pila(); //declarar las funciones

bool vacia();

void vaciar();

void agregar(T);

T sacar();

T tope();

};

template <class T>

pila<T>::pila(){

cabeza = NULL;

}

template <class T> //esto va antes de cada funcion

pila<T>::~pila(){

nodo \*borrar; // pila<T>::vaciar significa el metodo vaciar dentro de la clase pila con template T

while(cabeza != NULL){

borrar = cabeza;

cabeza = cabeza->siguiente;

delete borrar;

}

delete cabeza;

}

template <class T>

void pila<T>::vaciar(){

nodo \*borrar;

while(cabeza != NULL){

borrar = cabeza;

cabeza = cabeza->siguiente;

delete borrar;

}

}

template <class T>

bool pila<T>::vacia(){

if (cabeza == NULL){

return true;

}

else{

return false;

}

}

template <class T>

void pila<T>::agregar(T elemento){

nodo \*n = new nodo;

n->contenido = elemento;

n->siguiente = cabeza;

cabeza = n;

}

template <class T>

T pila<T>::sacar(){

nodo \*borrar = cabeza;

T ret = cabeza->contenido;

cabeza = cabeza->siguiente;

delete borrar;

borrar = NULL;

return ret;

}

template <class T>

T pila<T>::tope(){

T ret = cabeza->contenido;

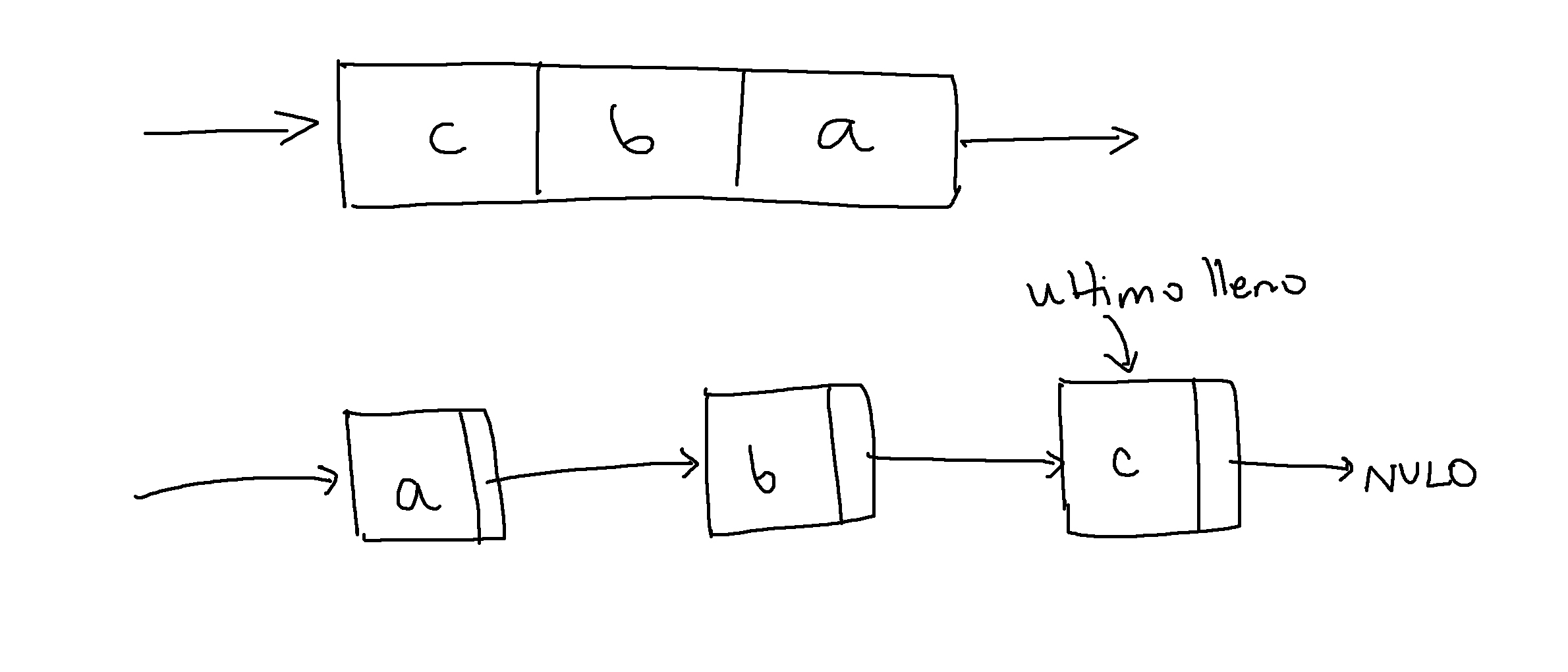
return ret;

}

### 4.2.2 Lista simplemente enlazada con puntero al último lleno para una cola

#### 4.2.2.1 Diagrama y descripción de lista simplemente enlazada con puntero al último lleno para una cola

Figura 2: Diagrama del modelo e implementación de una lista simplemente enlazada con puntero al ultimo lleno para una cola



La estructura de datos lista simplemente enlazada con puntero al último lleno se puede utilizar para representar el modelo cola. El primer elemento de la lista es el primero en entrar, al igual que el primero en salir. El puntero al último lleno apunta al último elemento añadido, que sería el último en salir.

#### 4.2.2.2 Definición en C++ de lista simplemente enlazada con puntero al último lleno para una cola

class Cola{

public:

Cola();//Constructor de Cola

virtual ~Cola();//Destructor de la cola.

void Vaciar();

bool Vacia();

void Agregar(int elem);

int Sacar();

int Frente();

void mostrarEnPantalla();

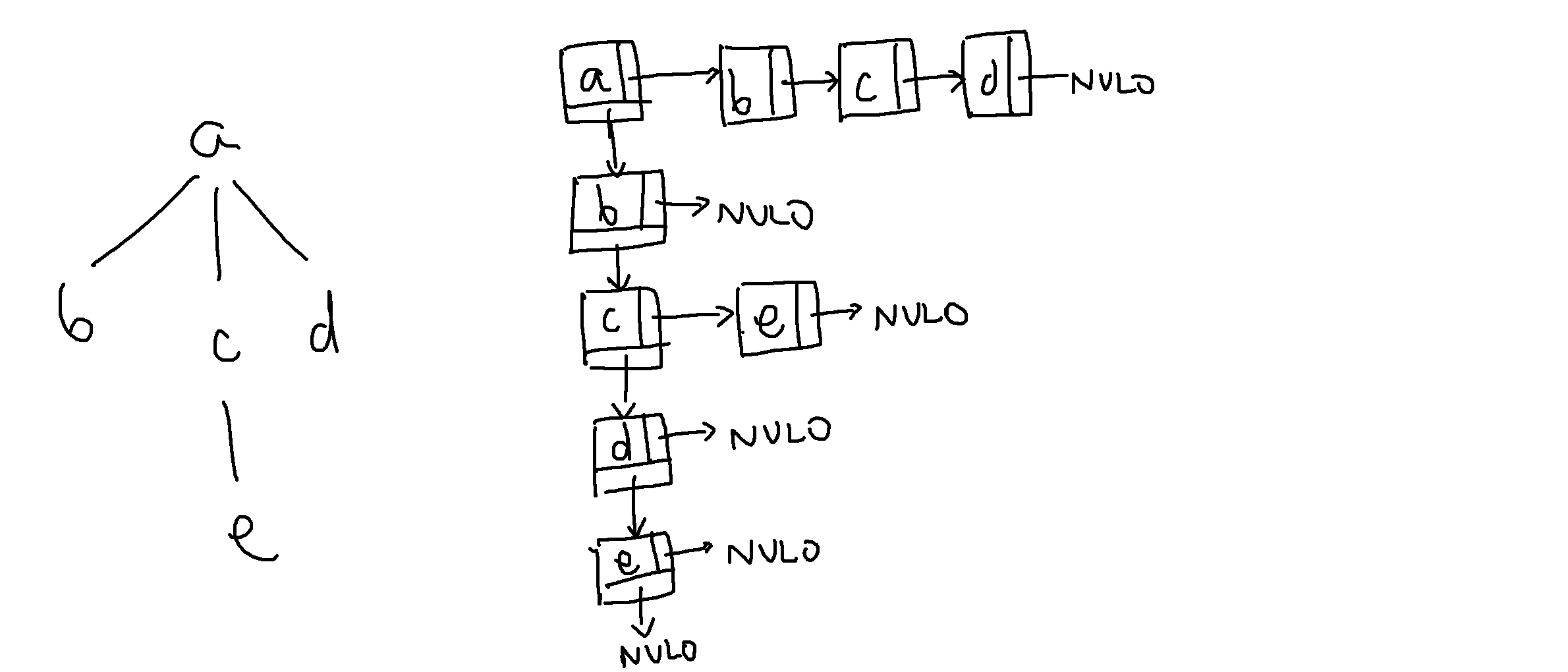
CajaCola \*ultim;

CajaCola \*prim;};

### 4.2.3 Lista de hijos por lista simplemente enlazada (lista principal) y lista simplemente enlazada (sublistas) para un árbol tal que no importa el orden entre los hijos

#### 4.2.3.1 Diagrama y descripción de lista de hijos por lista simplemente enlazada (lista principal) y lista simplemente enlazada (sublistas) para un árbol tal que no importa el orden entre los hijos

Figura 3: Diagrama del modelo e implementación de una lista de hijos por lista simplemente enlazada (lista principal) y lista simplemente enlazada (sublistas) para un árbol tal que no importa el orden entre los hijos



La estructura lista de hijos por lista simplemente enlazada para la lista principal y las sublistas se puede utilizar para implementar un árbol tal que no importa el orden entre los hijos. En la lista principal se encuentran todos los nodos del árbol y cada nodo apunta a otra lista simplemente enlazada que contiene a los hijos de ese nodo. Esta es una estructura de datos no análoga al modelo, ya que es iterativa y el modelo de árbol tal que no importa el orden entre los hijos es recursivo.

#### 4.2.3.2 Definición en C++ de lista de hijos por lista simplemente enlazada (lista principal) y lista simplemente enlazada (sublistas) para un árbol tal que no importa el orden entre los hijos

class Arbol1 {

public:

Arbol1();///probado

virtual ~Arbol1();///probado

void Vaciar();///probado

bool Vacio();///probado

void PoneRaiz(int elem);///probado

void AgregaHijo(CajLSE \*padre, int e);///probado

void BorraHoja(CajLSE \*aBorrar);///probado

void ModificaEtiqueta(CajLSE \*aCambiar, int elem);///probado

CajLSE \* Raiz();///probado

CajLSE \*Padre(CajLSE \*hijo);///probado

CajLSE \*Pertenece(int i);///probado

CajLSE \*HijoMasIzquierdo(CajLSE \*padre);///probado

CajLSE \*HermanoDerecho(CajLSE \*hermano);

int Etiqueta(CajLSE \*yo);///probado

bool EsHoja(CajLSE \*c);///probado

void mostrarLprincipal();///probado

CajLSE \*raiz;

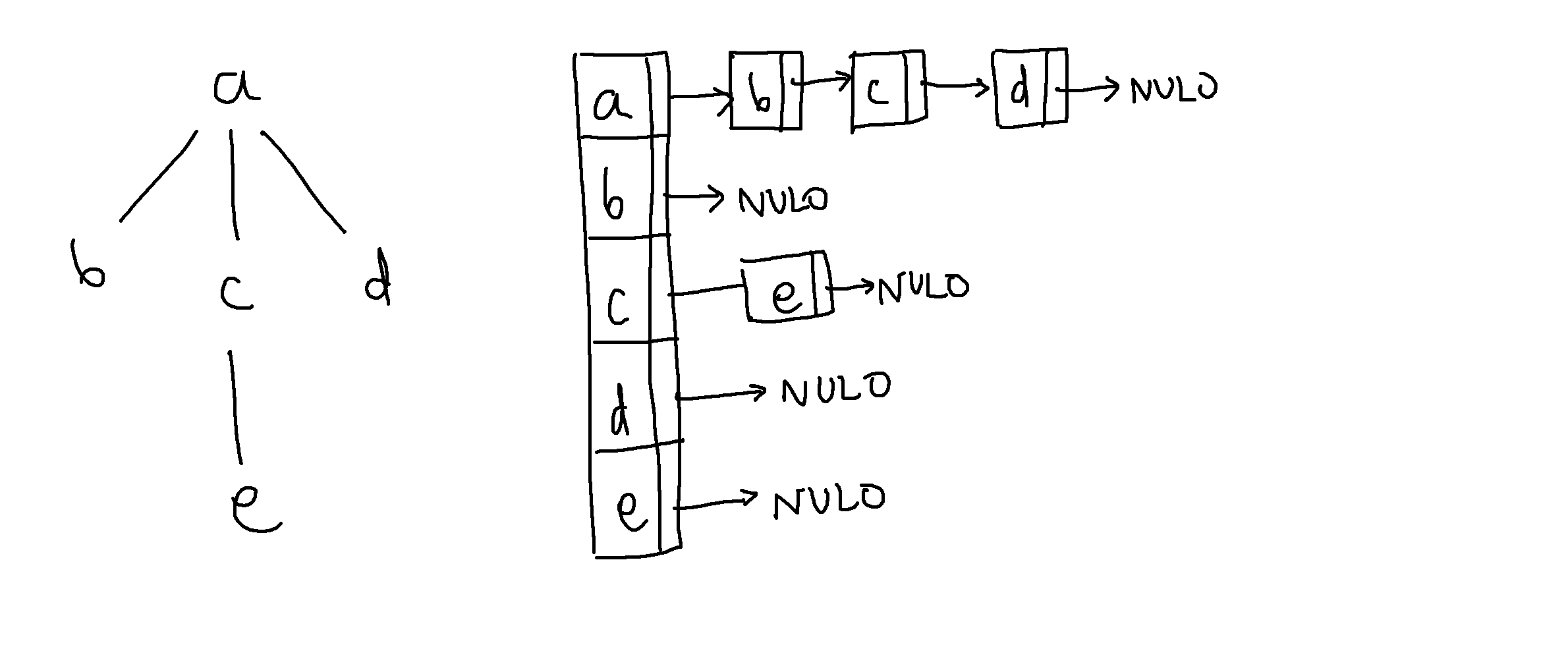
int numElem;

};

### 4.2.4 Lista de hijos por arreglo (lista principal) y lista simplemente enlazada (sublistas) para un árbol tal que no importa el orden entre los hijos

#### 4.2.4.1 Diagrama y descripción de lista de hijos arreglo (lista principal) y lista simplemente enlazada (sublistas) para un árbol tal que no importa el orden entre los hijos

Figura 4: Diagrama del modelo e implementación de una lista de hijos por arreglo (lista principal) y lista simplemente enlazada (sublistas) para un árbol tal que no importa el orden entre los hijos



La estructura lista de hijos por arreglo para la lista principal y listas simplemente enlazadas para las sublistas se puede utilizar para implementar un árbol tal que no importa el orden entre los hijos. En un arreglo se encuentran todos los nodos del árbol y cada nodo apunta a una lista simplemente enlazada que contiene a los hijos de ese nodo. Esta es una estructura de datos no análoga al modelo, ya que es iterativa y el modelo de árbol tal que no importa el orden entre los hijos es recursivo. Una limitación del modelo es que el arreglo es de tamaño fijo.

#### 4.2.4.2 Definición en C++ de lista de hijos con arreglo (lista principal) y lista simplemente enlazada (sublistas) para un árbol tal que no importa el orden entre los hijos

class Arbol2{

public:

Arbol2();///probado

virtual ~Arbol2();///probado

void Vaciar();///probado

bool Vacio();///probado

void PoneRaiz(int elem);///probado

void AgregaHijo(CajaLisArray \*padre, int e);///probado

void BorraHoja(CajaLisArray \*aBorrar);///probado

void ModificaEtiqueta(CajaLisArray \*aCambiar, int elem);///probado

CajaLisArray \*Raiz();///probado

CajaLisArray \*Padre(int hijo);///probado

CajaLisArray \*Padre(CajaLisArray \*hijo);///probado

CajaLisArray \*HijoMasIzquierdo(CajaLisArray \*padre);///probado

CajaLisArray \*HermanoDerecho(CajaLisArray \*hermano);///probado

int Etiqueta(CajaLisArray \*yo);///probado

bool EsHoja(CajaLisArray \*c);///probado

void mostrarLprincipal();///probado

CajaLisArray \*Pertenece(int i);///probado

CajaLisArray \*miArbol[100];

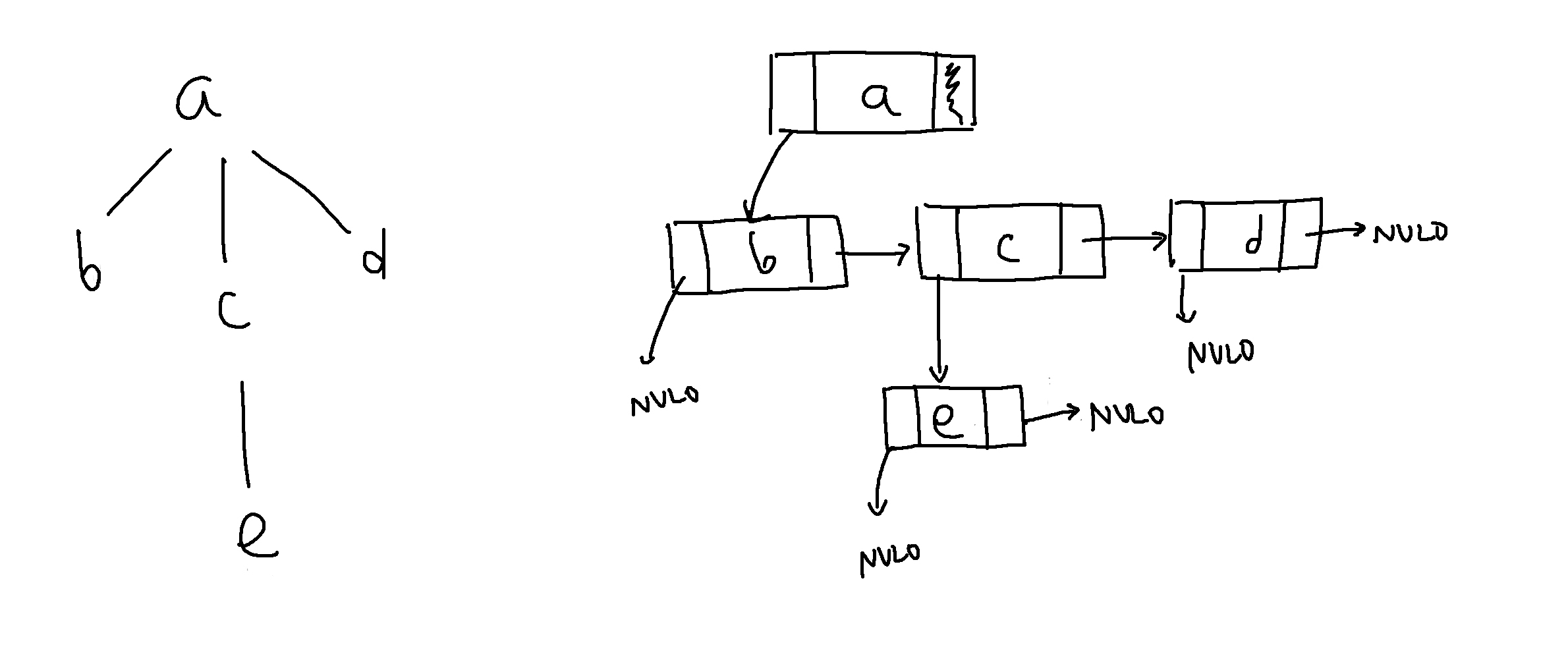
int ultimo;

};

### 4.2.5 Hijo Más Izquierdo-Hermano Derecho por punteros para un árbol tal que no importa el orden entre los hijos

#### 4.2.5.1 Diagrama y descripción de hijo Más Izquierdo-Hermano Derecho por punteros para un árbol tal que no importa el orden entre los hijos

Figura 5: Diagrama del modelo e implementación de una árbol Hijo Más Izquierdo-Hermano Derecho por punteros para un árbol tal que no importa el orden entre los hijos



La estructura Hijo Más Izquierdo-Hermano Derecho por punteros se puede utilizar para implementar un árbol tal que no importa el orden entre los hijos. Cada nodo tiene tres campos: un puntero al hijo más izquierdo, un puntero al hermano derecho, y la etiqueta correspondiente a ese nodo. Esta es una estructura de datos análoga al modelo original, ya que es recursiva.

#### 4.2.5.2 Definición en C++ de hijo Más Izquierdo-Hermano Derecho por punteros para un árbol tal que no importa el orden entre los hijos

class arbolHIHD{

nodo \*root = new nodo;

public:

arbolHIHD();

~arbolHIHD();

void poneRaiz(int);

void vaciar();

bool vacia();

nodo\* hijoMasIzquierdo(nodo\*);

nodo\* hermanoDerecho(nodo\*);

void agregarHijo(nodo\*, int);

void borrarNodo(nodo\*);

int etiqueta(nodo\*);

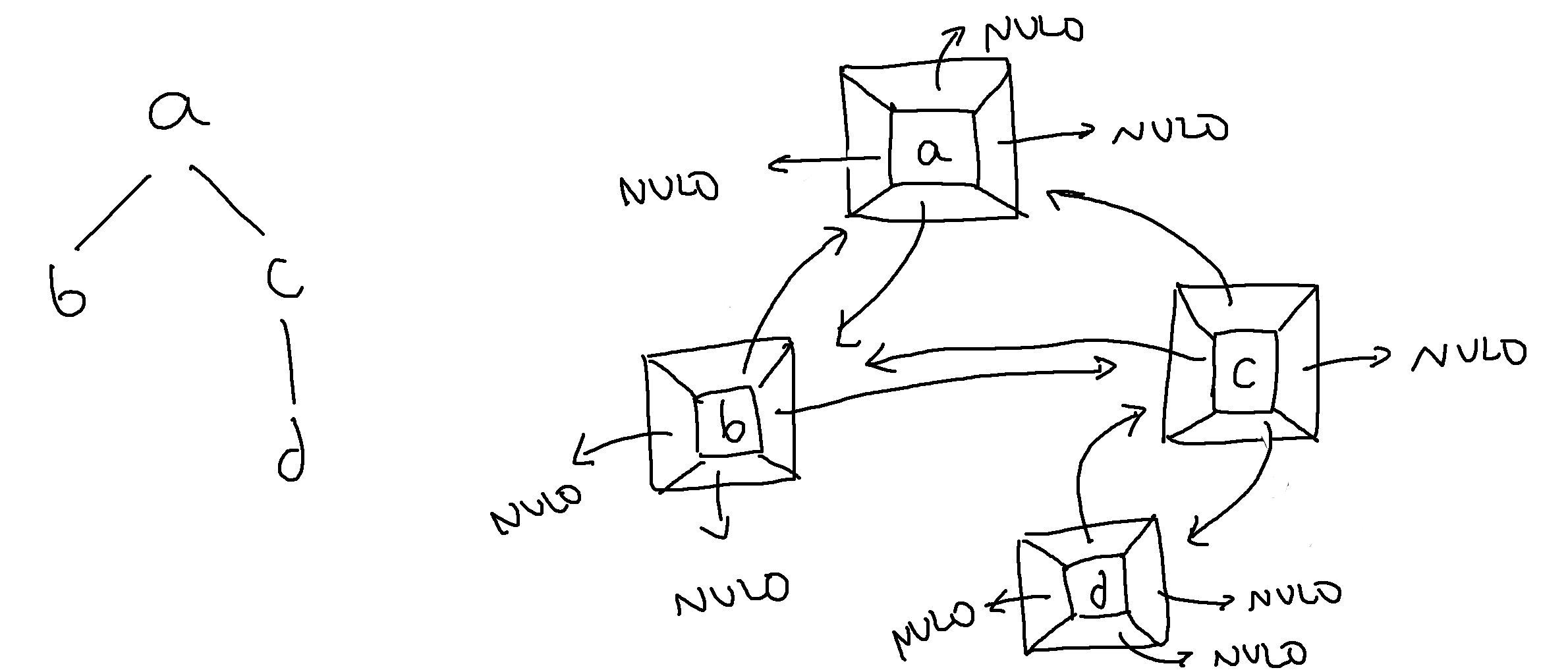
nodo\* raiz();

};

### 4.2.6 Hijo Más Izquierdo-Hermano Derecho por punteros con puntero al padre y al hermano izquierdo para un árbol tal que no importa el orden entre los hijos

#### 4.2.6.1 Diagrama y descripción de hijo Más Izquierdo-Hermano Derecho por punteros con puntero al padre y al hermano izquierdo para un árbol tal que no importa el orden entre los hijos

Figura 6: Diagrama del modelo e implementación de una árbol Hijo Más Izquierdo-Hermano Derecho por punteros con puntero al padre y al hermano izquierdo para un árbol tal que no importa el orden entre los hijos



La estructura Hijo Más Izquierdo-Hermano Derecho por punteros se puede utilizar para implementar un árbol tal que no importa el orden entre los hijos. En este caso específico, cada nodo tiene cinco campos: un puntero al hijo más izquierdo, un puntero al hermano derecho, un puntero al hermano izquierdo, un puntero al padre y la etiqueta correspondiente a ese nodo. Esta es una estructura de datos análoga al modelo original, ya que es recursiva.

#### 4.2.6.2 Definición en C++ de hijo Más Izquierdo-Hermano Derecho por punteros con puntero al padre y al hermano izquierdo para un árbol tal que no importa el orden entre los hijos

class arbolHIHD{

nodo \*root = new nodo;

public:

arbolHIHD();

~arbolHIHD();

void poneRaiz(int);

void vaciar();

bool vacia();

nodo\* hijoMasIzquierdo(nodo\*);

nodo\* hermanoDerecho(nodo\*);

void agregarHijo(nodo\*, int);

void borrarNodo(nodo\*);

int etiqueta(nodo\*);

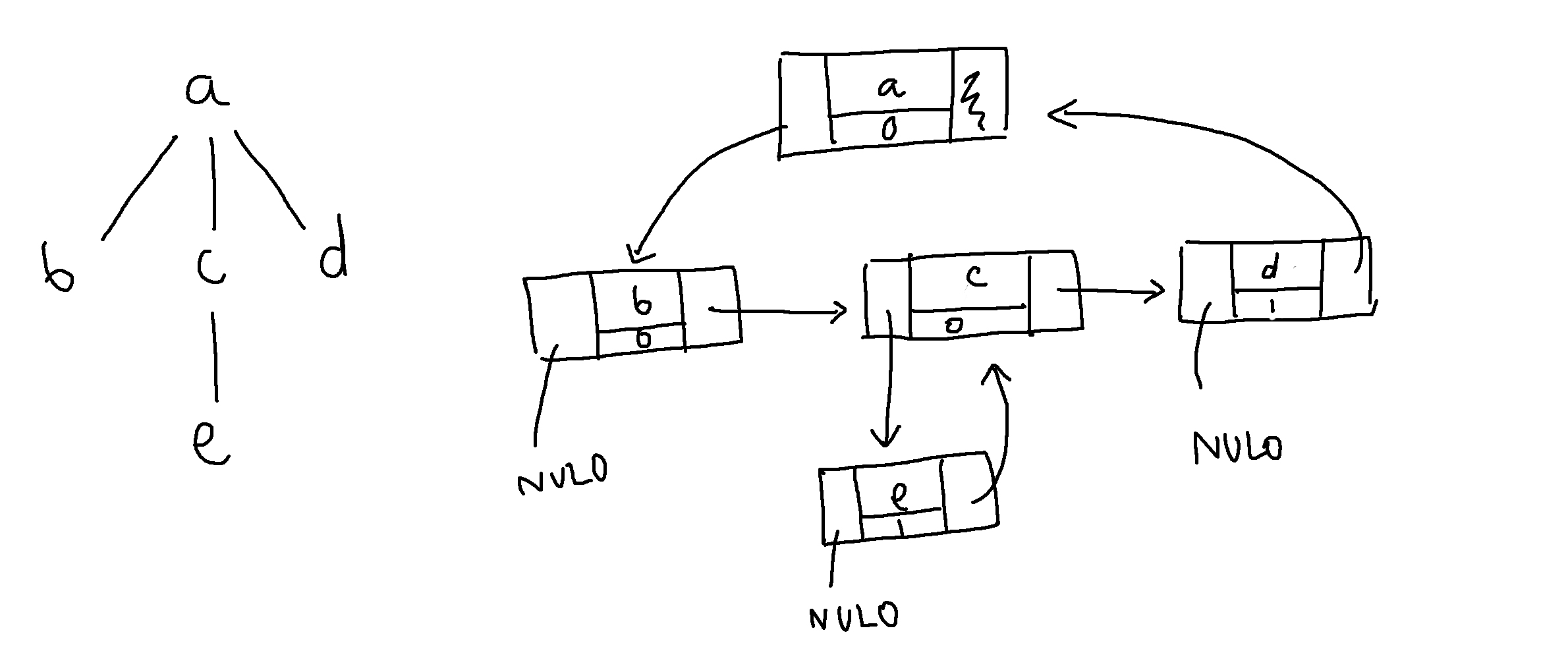
nodo\* raiz();

};

### 4.2.7 Hijo Más Izquierdo-Hermano Derecho por punteros tal que el último hijo de un nodo apunta al padre para un árbol tal que no importa el orden entre los hijos

#### 4.2.7.1 Diagrama y descripción de hijo Más Izquierdo-Hermano Derecho por punteros tal que el último hijo de un nodo apunta al padre para un árbol tal que no importa el orden entre los hijos

Figura 7: Diagrama del modelo e implementación de una árbol Hijo Más Izquierdo-Hermano Derecho por punteros tal que el último hijo de un nodo apunta al padre para un árbol tal que no importa el orden entre los hijos



La estructura Hijo Más Izquierdo-Hermano Derecho por punteros se puede utilizar para implementar un árbol tal que no importa el orden entre los hijos. Cada nodo tiene tres campos: un puntero al hijo más izquierdo, un puntero al hermano derecho, y la etiqueta correspondiente a ese nodo. En este caso particular, el último hijo de un mismo nivel apunta al padre. Esta es una estructura de datos análoga al modelo original, ya que es recursiva.

#### 4.2.7.2 Definición en C++ de hijo Más Izquierdo-Hermano Derecho por punteros tal que el último hijo de un nodo apunta al padre para un árbol tal que no importa el orden entre los hijos

class arbolPPHI{

nodo \* root = new nodo;

public:

arbolPPHI();

~arbolPPHI();

void vaciar();

bool vacia();

nodo\* hijoMasIzquierdo(nodo\*);

nodo\* hermanoDerecho(nodo\*);

void agregarHijo(nodo\*, int);

void borrarNodo(nodo\*);

int etiqueta(nodo\*);

nodo\* raiz();

void poneRaiz(int);

nodo\* encuentraNodo(int);

};

## 4.3 Algoritmos

### 4.3.1 HermanoIzquierdo(TipoÁrbol a, entero HD)

#### 4.3.1.1 Definición y especificación del algoritmo

Devuelve: Un nodo

Efecto: Averigua cual es el hermano izquierdo de un nodo y lo devuelve(o devuelve un nulo de no tenerlo)

Requiere: Arbol inicializado y no vacio, que la etiqueta enviada como parametro este en algun nodo del arbol y no sea repetida.

Modifica: N/A

#### 4.3.1.2 Descripción, detalles, pseudolenguaje

entero HermanoIzquierdo(TipoArbol a, entero HD){

tipoNodo n=a->Pertenece(HD);

n=HermanoIzquierdoM(a,n);

entero j=(a->Etiqueta(n));

return j;

}

tipoNodo Algoritmos::HermanoIzquierdoM(TipoArbol a, tipoNodo n){

tipoNodo HI;

if (n==a->Raiz())

{

HI= 0;

}

else

{

tipoNodo q=a->Padre(n);

if (a->HijoMasIzquierdo(q)==n)

{

HI=0;

}

else

{

tipoNodo iter=a->HijoMasIzquierdo(q);

tipoNodo iterHD=a->HermanoDerecho(iter);

bool encontrado=false;

while ((iterHD!=0)&&(!encontrado))

{

if(iterHD==n)

{

HI=iter;

encontrado=true;

}

else

{

iter=iterHD;

iterHD=a->HermanoDerecho(iter);

}

}

}

}

return HI;

}

### 4.3.2 AlturaNodo(tipoNodo n)

#### 4.3.2.1 Definición y especificación del algoritmo

Devuelve: Un entero

Efecto: Averigua a que altura esta un nodo.

Requiere: Arbol inicializado y no vacio, n valido en a.

Modifica: N/A

#### 4.3.2.2 Descripción, detalles, pseudolenguaje

entero AlturaNodo(TipoArbol a, entero etiqueta){

tipoNodo n=a->Pertenece(etiqueta);

return AlturaNodo(a,n);

}

entero Algoritmos::AlturaNodo(TipoArbol a, tipoNodo n)

{

return AlturaNodoRe(a,n);

}

entero AlturaNodoRe(TipoArbol a, tipoNodo n)

{

entero nivelMayor=1;

cout<<"Etiq:"<<a->Etiqueta(n)<<endl;

if(a->Etiqueta(n)!=(17))

{

if(!n->EsHoja())

{

tipoNodo iter=a->HijoMasIzquierdo(n);

entero nivelIter=1;

while (iter!=0)

{

nivelIter=AlturaNodoRe(a,iter);

if (nivelIter>nivelMayor)

{

nivelMayor=nivelIter;

}

iter=a->HermanoDerecho(iter);

}

nivelMayor=nivelMayor+1;

}

}

else

{

return 0;

}

return nivelMayor;

}

### 4.3.3 etiquetasRepetidas(tipoÁrbol a)

#### 4.3.3.1 Definición y especificación del algoritmo

Devuelve: Un booleano

Efecto: Averigua si un arbol tiene etiquetas repetidas y regresa un booleano (verdadero si las hay o falso si no).

Requiere: Arbol inicializado y no vacio, n valido en a.

Modifica: N/A

#### 4.3.3.2 Descripción, detalles, pseudolenguaje

bool Algoritmos::etiquetasRepetidas(TipoÁrbol a){

list<entero> coll;

list<entero>::iterator pos;

Cola \*cOrig=new Cola();

tipoNodo nActOrig;

tipoNodo hijOrig=0;

entero etiqueta;

bool repetidos=false;

if (!(a->Vacio()))

{

nActOrig=a->Raiz();

cOrig->Agregar(nActOrig);

while ((!(cOrig->Vacia()))&&(!repetidos))

{

nActOrig=cOrig->Sacar();

etiqueta=a->Etiqueta(nActOrig);

pos = find (coll.begin(), coll.end(), etiqueta);

if(pos!=coll.end())

{

repetidos=true;

}

else

{

coll.push\_back(etiqueta);

}

hijOrig=a->HijoMasIzquierdo(nActOrig);

while ((hijOrig!=0)&&(!repetidos))

{

etiqueta=a->Etiqueta(hijOrig);

cOrig->Agregar(hijOrig);

hijOrig=a->HermanoDerecho(hijOrig);

}

}

}

return repetidos;

}

### 4.3.4 nivelesArbol(tipoÁrbol A)

#### 4.3.4.1 Definición y especificación del algoritmo

Devuelve: Un entero

Efecto: Devuelve cuantos niveles en árbol

Requiere: A inicializado

Modifica: N/A

#### 4.3.4.2 Descripción, detalles, pseudolenguaje

entero Algoritmos::nivelesArbol(TipoArbol a)

{

return AlturaNodo(a,a->Raiz());

}

### 4.3.5 hijosNodo(tipoÁrbol A, tipoNodo n)

#### 4.3.5.1 Definición y especificación del algoritmo

Devuelve: Un entero

Efecto: Devuelve el número de hijos que tiene un nodo

Requiere: A inicializado

Modifica: N/A

#### 4.3.5.2 Descripción, detalles, pseudolenguaje

entero Algoritmos::hijosNodo(TipoÁrbol a, tipoNodo n)

{

entero hijos=0;

if(!n->EsHoja())

{

tipoNodo iter=a->HijoMasIzquierdo(n);

while (iter!=0)

{

hijos=hijos+1;

iter=a->HermanoDerecho(iter);

}

}

return hijos;

}

### 4.3.6 listarHijosNodo(Árbol \*a, tipoNodo n)

#### 4.3.6.1 Definición y especificación del algoritmo

Efecto: Imprime las etiquetas correspondientes a los hijos de un nodo

Requiere: A inicializado

Modifica: N/A

#### 4.3.6.2 Descripción, detalles, pseudolenguaje

void listarHijosNodo(Arbol \*a, tipoNodo n)

{

tipoNodo hijo=a->HijoMasIzquierdo(n);

if(hijo==0)

{

cout<<"Este nodo no tiene hijos"<<endl;

}

else

{

cout<<"Hijos del nodo con etiqueta "<<a->Etiqueta(n)<<":"<<endl;

while(hijo!=0)

{

cout<<a->Etiqueta(hijo)<<endl;

hijo=a->HermanoDerecho(hijo);

}

cout<<"Fin de lista de hijos"<<endl;

}

}

### 4.3.7 borrarSubÁrbol(tipoÁrbol A)

#### 4.3.7.1 Definición y especificación del algoritmo

Efecto: Borra el sub-árbol generado a partir de un nodo

Requiere: A inicializado

Modifica: A

#### 4.3.7.2 Descripción, detalles, pseudolenguaje

void Algoritmos::borrarSubArbol(TipoArbol a, tipoNodo n){

cout << "entro iter"<< endl;

borrarSubArbolRe(a,n);

cout << "salio iter"<< endl;

}

void borrarSubArbolRe(tipoArbol a, tipoNodo n){

if((a->HijoMasIzquierdo(n)!=0))

{

tipoNodo iter=a->HijoMasIzquierdo(n);

tipoNodo iter2;

while (iter!=0)

{

iter2=a->HermanoDerecho(iter);

borrarSubArbolRe(a,iter);

iter=iter2;

}

}

a->BorraHoja(n);

}

### 4.3.8 copiarArbol(tipoÁrbol A)

#### 4.3.8.1 Definición y especificación del algoritmo

Efecto: Copia el árbol A en el árbol B

Requiere: A inicializado

Modifica: B

#### 4.3.8.2 Descripción, detalles, pseudolenguaje

void Algoritmos::copiarÁrbol (TipoÁrbol aOriginal, TipoÁrbol aCopia) //requiere aCopia vacío

{

Cola \*cOrig=new Cola();

Cola \*cCop=new Cola ();

tipoNodo nActOrig;

tipoNodo nActCop;

tipoNodo hijOrig=0;

tipoNodo hijCop=0;

entero etiqueta;

if (!(aOriginal->Vacio()))

{

aCopia->PoneRaiz(aOriginal->Etiqueta(aOriginal->Raiz()));

nActOrig=aOriginal->Raiz();

nActCop=aCopia->Raiz();

cOrig->Agregar(nActOrig);

cCop->Agregar(nActCop);

while (!cOrig->Vacia())

{

nActOrig=cOrig->Sacar();

nActCop=cCop->Sacar();

hijOrig=aOriginal->HijoMasIzquierdo(nActOrig);

while (hijOrig!=0)

{

etiqueta=aOriginal->Etiqueta(hijOrig);

aCopia->AgregaHijo(nActCop,etiqueta);

hijCop=direccionNodo(aCopia,nActCop,etiqueta);

cOrig->Agregar(hijOrig);

cCop->Agregar(hijCop);

hijOrig=aOriginal->HermanoDerecho(hijOrig);

}

}

}

}

tipoNodo direccionNodo(TipoArbol a, tipoNodo padre, entero etiqueta)

{

tipoNodo n=a->HijoMasIzquierdo(padre);

bool noEncontrado=true;

while(n!=0&&noEncontrado)

{

if(a->Etiqueta(n)==etiqueta)

{

noEncontrado=false;

}

else

{

n=a->HermanoDerecho(n);

}

}

return n;

}

### 4.3.9 revisaArbolesIguales(tipoÁrbol A, tipoÁrbol B)

#### 4.3.9.1 Definición y especificación del algoritmo

Devuelve: Un booleano

Efecto: Retorna verdadero si los arboles A y B son iguales, falso si no

Requiere: A y B inicializados

Modifica: N/A

#### 4.3.9.2 Descripción, detalles, pseudolenguaje

bool revisaArbolesIguales(arbol A, arbol B){

si (A.raiz() == nodoNulo Y B.raiz() == nodoNulo){

retornar verdadero;

}

si otro (revisaArbolesIgualesRecursivo(A, A.raiz(), B, B.raiz()) == verdadero){

retornar verdadero;

}

si no {

retornar falso;

}

bool revisaArbolesIgualesRecursivo(arbol A, nodo AN, arbol B, nodo BN){

nodo actualA = AN;

nodo hijoA = actualA.hijoMasIzquierdo();

nodo actualB = BN;

nodo hijoB;

si (actualA.etiqueta() != actualB.etiqueta()){

retorna falso;

}

mientras (hijoA != NULO){

si (actualB.hijoMasIzquierdo == NULO){

retornar falso;

}

hijoB = actualB.hijoMasIzquierdo();

mientras (hijoB != NULO){

si (hijoA.etiqueta() == hijoB.etiqueta()){

si (revisaArbolesIgualesRecursivo(A, hijoA, B, hijoB) == verdadero){

break;

}

si no {

retornar falso;

}

}

si no {

si (hijoB.hermanoDerecho() == nodoNULO){

retornar falso;

}

si no {

hijoB = hijoB.hermanoDerecho();

}

}

}

hijoA = hijoA.hermanoDerecho();

}

}

### 4.3.10 PreOrden(tipoÁrbol A)

#### 4.3.10.1 Definición y especificación del algoritmo

Efecto: Imprime en pantalla los nodos del árbol en pre-orden

Requiere: A inicializado

Modifica: N/A

#### 4.3.10.2 Descripción, detalles, pseudolenguaje

PreOrden(tipoÁrbol A){

nodo = A.HijoMasIzquierdo()

Imprimir raíz

Mientras nodo != nulo {

PreOrdenRec(nodo, lista)

nodo = A.HermanoDerecho()

}

### 4.3.11 PostOrden(tipoÁrbol A)

#### 4.3.11.1 Definición y especificación del algoritmo

Efecto: Imprime en pantalla los nodos del árbol en post-orden

Requiere: A inicializado

Modifica: N/A

#### 4.3.11.2 Descripción, detalles, pseudolenguaje

PostOrdenRec(tipoÁrbol A){

nodo = A.HijoMasIzquierdo()

Mientras nodo != nulo {

PostOrdenRec(nodo, lista)

nodo = A.HermanoDerecho()

}

Imprimir raíz

}

### 4.3.12 InOrden(tipoÁrbol A)

#### 4.3.12.1 Definición y especificación del algoritmo

Devuelve: Una lista de nodos generada por recorrido in-orden

Efecto: Recorre el árbol A in-orden de manera recursiva (utilizando la pila de la recursividad que provee el compilador) para listar los nodos que lo conforman

Requiere: A inicializado

Modifica: N/A

#### 4.3.12.2 Descripción, detalles, pseudolenguaje

InOrden(tipoÁrbol A){

nodo = A.HijoMasIzquierdo()

Mientras nodo != nulo {

Imprimir raíz

PostOrdenRec(nodo, lista)

nodo = A.HermanoDerecho()

}

}

### 4.3.13 listadoNiveles(tipoÁrbol A)

#### 4.3.13.1 Definición y especificación del algoritmo

Efecto: Imprime los nodos de A en orden de niveles

Requiere: A inicializado

Modifica: N/A

#### 4.3.13.2 Descripción, detalles, pseudolenguaje

listadoNiveles(arbol A){

nodo actual = A.raiz()

nodo hijo;

cola C;

C.agregar(actual)

mientras (C.vacia() == Falso){

actual = C.sacar()

imprimir actual;

hijo = actual.hijoMasIzquierd()

mientras (hijo != nodoNulo){

C.agregar(hijo)

hijo = hijo.hermanoDerecho();

}

}

}

### 4.3.14 ListadoRecursivoPila()

#### 4.3.14.1 Definición y especificación del algoritmo

Efecto: Imprime los nodos de A en preorden

Requiere: A inicializado

Modifica: N/A

#### 4.3.14.2 Descripción, detalles, pseudolenguaje

ListadoRecursivoPila(Arbol A){

pila P;

nodo actual = A.raiz();

P.agregar(actual);

Imprimir actual;

mientras (P.vacia() == Falso){

si(actual.hijoMasIzquierdo != nodoNulo){

actual = actual.hijoMasIzquierdo();

P.agregar(actual);

imprimir actual;

continue;

}

actual = P.sacar();

si (actual.hermanoDerecho != nodoNulo){

actual = actual.hermanoDerecho;

P.agregar(actual);

imprimir actual;

}

}

}

### 4.3.15 encuentraNodo(tipoEtiqueta e, tipoÁrbol A)

#### 4.3.15.1 Definición y especificación del algoritmo

Devuelve: Un nodo

Efecto: Busca el nodo con la etiqueta E en el árbol A y lo retorna

Requiere: A inicializado, e válida en A

Modifica: N/A

#### 4.3.15.2 Descripción, detalles, pseudolenguaje

encuentraNodo(etiqueta E, Arbol A){

cola C;

nodo actual = A.raiz();

C.agregar(actual);

mientras (C.vacia() == Falso){

actual = C.sacar();

si (actual.etiqueta() == E){

retornar actual;

}

si (actual.hijoMasIzquierdo() != nodoNulo){

actual = actual.hijoMasIzquierdo();

C.agregar(actual);

mientras (actual.hermanoDerecho() != nodoNulo){

actual = actual.hermanoDerecho();

C.agregar(actual);

}

}

}

}

# 5. Manual del usuario

5.1. Requerimientos de hardware

Se requiere una computadora que después de cargar y correr el sistema operativo tenga 512 MB de RAM libres.

## 5.2. Requerimientos de software

Debe contar con [Code::Blocks](http://www.codeblocks.org/downloads) para compilar y correr el programa.

## 5.3. Arquitectura del programa

El programa cuenta con una interfaz que se comunica con las distintas estructuras de datos, que son implementaciones de los modelos descritos en la sección 4.1. Las estructuras de datos están implementadas en siete archivos .h que se incluyen en el programa principal de prueba.

## 5.4. Compilación

Para compilar se utiliza el entorno de Code::Blocks, que utiiza el compilador GNU GCC (MinGW). El archivo .cbp es el que se utiliza para generar el ejecutable que tiene todos los datos de prueba. Para compilar seleccione la opción “Build” y para correr seleccione “Run”.

## 5.5. Especificación de las funciones del programa

# 6. Datos de Prueba

## 6.1. Formato de los datos de prueba

Los datos de prueba se encuentran en el método principal del programa. Estos consisten en diferentes valores escogidos y conocidos para verificar que las estructuras de datos implementadas estuvieran funcionando de la manera adecuada.

En el main hicimos pruebas de la siguiente manera:

En primer lugar tomamos los métodos de cada estructura de datos, los inicializamos, agregamos datos y verificamos que todos los operadores funcionaran. Esto lo hicimos por medio de preguntas. Como fuimos nosotros los que les agregamos valores sabíamos de ante mano que valores debían salir por lo que en su mayoría hicimos preguntas que se imprimen en pantalla como por ejemplo:

“cout << "Cola esta vacia?True"<<((c->Vacia())?"Bien”: “Mal")<<endl;”

Esta línea de código lo que hace es imprimir "Cola esta vacia? True" (true es el valor que debería devolver) y si devuelve el booleano esperado imprime “bien” y de no hacerlo “mal”.

## 6.2. Salida esperada

## 6.3. Salida obtenida (Análisis en caso de fallo)

# 8. Listado de archivos

El archivo etapaII\_B32303\_B32896\_B36269.zip contiene:

1. Una carpeta llamada etapaII\_B32303\_B32896\_B36269 la cual contiene:
   1. Un archivo B32303\_B32896\_B36269.doc que contiene la documentación externa y el manual de usuario.
   2. Una carpeta “bin” que contiene la aplicación en el archivo ejecutable
   3. Una carpeta “include” que contiene los archivos de encabezado
      1. Arbol1.h, Arbol2.h, ArbolHIHD.h , ArbolPPHI.h, ArbolUHPB.h, CajaCola.h, CajaLisArray.h, CajLSE.h, CajSubLSE1.h, CajSubLSE2.h, Cola.h, Pila.h
   4. Un archivo “main.cpp” que contiene el método principal del programa
   5. Una carpeta “obj” que contiene el código objeto del proyecto
   6. Una carpeta “src” que contiene el código fuente en C++
      1. Arbol1.cpp, Arbol2.cpp, ArbolHIHD.cpp , ArbolPPHI.cpp, ArbolUHPB.cpp, CajaCola.cpp, CajaLisArray.cpp, CajLSE.cpp, CajSubLSE1.cpp, CajSubLSE2.cpp, Cola.cpp, Pila.cpp
   7. Un archivo TP1.cbp que contiene el proyecto en Code::Blocks

# 9. Referencias

Aho, A. (1983). Data Structures and Algorithms. Addison-Wesley. pp. 16, 19, 91, 93