Árbol Genealógico

Curso: Estructura de datos

Integrantes:

- Huayllani Otarola Erick Marvil
- Balbin Gutierrez Leonardo
- Torres Escalante Jhon Jaime

Introducción

En esta sección se presenta el desarrollo de un sistema para gestionar árboles genealógicos utilizando estructuras de datos jerárquicas (árboles binarios y N-arios). Diseñado para arqueólogos que estudian una antigua civilización, el sistema permite almacenar, consultar y analizar relaciones familiares de manera eficiente, garantizando operaciones rápidas de inserción, búsqueda y eliminación.

Objetivos principales:

- Modelar relaciones familiares con un ABB (Árbol Binario de Búsqueda) y adaptarlo a un árbol N-ario para reflejar estructuras complejas.
- Implementar operaciones básicas (alta, baja, consultas) con complejidad óptima (O(log n) en promedio).
- Ofrecer una interfaz intuitiva para usuarios no técnicos, con validación robusta de datos.

Ejemplo de Código:

A continuación, se presenta el ejemplo práctico del programa que implementa la gestión del árbol genealógico. Este código ilustra:

1. La estructura base (nodos Persona con punteros a padre e hijos).
2. Operaciones clave (creación, búsqueda y vinculación de miembros).
3. Recorridos (preorden, inorden y postorden) para visualizar relaciones.

Define una estructura llamada "Persona" en C++ ,que está diseñada para construir una estructura de datos jerárquica o árbol.

```cpp
#include <iostream>
#include <string>
#include <vector>
using namespace std;

struct Persona {
    string nombre;
    int edad;
    Persona* padre;
    Persona* izquierda;
    Persona* derecha;
};
```

1. Usamos un string para almacenar el nombre de la **persona.**
2. **En padre**:Un puntero a otra Persona (el padre en una jerarquía familiar o en un árbol).
3. La **”Izquierda”** un puntero a otra persona(de un Hij@ izquierdo en la estructura del árbol).
4. La **“Derecha”**un puntero a otra persona(de un hij@ derecho en el árbol binario)

Implementa una función para crear y gestionar instancias de la estructura Persona en un árbol binario.

```
14  vector<Persona*> personas;
15
16  Persona* crearPersona(string nombre, int edad) {
17      Persona* nueva = new Persona;
18      nueva->nombre = nombre;
19      nueva->edad = edad;
20      nueva->padre = NULL;
21      nueva->izquierda = NULL;
22      nueva->derecha = NULL;
23      personas.push_back(nueva);
24      return nueva;
25  }
```

1.Registro Central:

- Vector global que almacena todos los nodos creados
- Permite seguimiento de todas las personas en el árbol

2.Inicialización Segura:

- Todos los punteros se inicializan a NULL
- Evita referencias no válidas

3.Gestión de Memoria:

- Uso de “new” para asignación dinámica
- Devuelve puntero para construcción del árbol

Implementamos una función para buscar Personas por nombre en el registro.

Declaramos la función

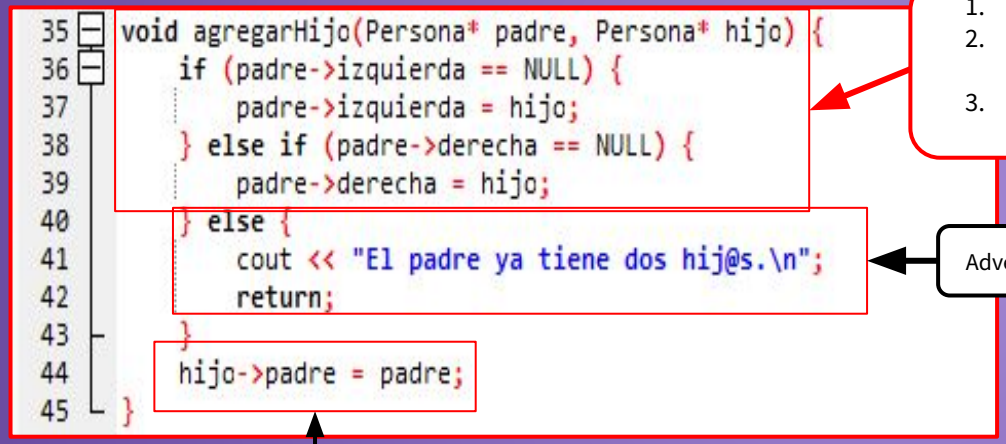
```
27  Persona* buscarPersona(string nombre) {
28      for (int i = 0; i < personas.size(); i++) {
29          if (personas[i]->nombre == nombre)
30              return personas[i];
31      }
32      return NULL;
33  }
```

comparación de nombres y retorno si encuentra y ala vez si no encuentra-

1.Mecanismo de Búsqueda:

- Recorre el vector personas de forma secuencial.
- Compara el nombre buscado con cada elemento.
- Retorna al primer match encontrado.

Va permitir establecer relaciones padre-hijo en una estructura de árbol binario.



1. Realizará la declaración.
2. Verifica y asigna hijo izquierdo.
3. Verifica y asigna hijo derecho.

Advertencia de datos existentes y salida temprana

Establece relación inversa

1.Gestión de Relaciones:

- Establece relación bidireccional(Permite navegar el árbol en ambas direcciones).
- Actualiza puntero padre del hijo automáticamente.

Muestra todos los ancestros de una persona (recursivamente hacia arriba) y Muestra todos los descendientes de una persona (recursivamente hacia abajo)

```
47  void mostrarAncestros(Persona* persona) {
48      if (persona->padre != NULL) {
49          cout << persona->padre->nombre << " (" << persona->padre->edad << ")\n";
50          mostrarAncestros(persona->padre);
51      }
52  }
53
54  void mostrarDescendientes(Persona* persona) {
55      if (persona != NULL) {
56          if (persona->izquierda)
57              cout << persona->izquierda->nombre << " (" << persona->izquierda->edad << ")\n";
58          if (persona->derecha)
59              cout << persona->derecha->nombre << " (" << persona->derecha->edad << ")\n";
60          mostrarDescendientes(persona->izquierda);
61          mostrarDescendientes(persona->derecha);
62      }
63  }
64
```

1.**Recursión simple:** Solo avanza hacia el padre
2.**Formato:** Muestra "Nombre (Edad)"
3.**Orden**: Del padre directo al ancestro más lejano
4.**Limitación**: No verifica si persona es "NULL" inicialmente.

Recursividad en ambos árboles

1.**Recursión doble:** Va explora ambos hijos
2.**Orden:** Primero hijos directos, luego sus descendientes
3.**Seguridad**: Verifica “NULL” antes de acceder a punteros
4.**Formato consistente:** Igual que ancestros ("Nombre (Edad)")

La función que elimina recursivamente un subárbol completo desde un nodo dado, incluyendo(Todos sus descendientes, Él propio nodo, Su referencia en el vector global personas)

```
65  void eliminarSubarbol(Persona* persona) {
66      if (persona == NULL) return;
67
68      eliminarSubarbol(persona->izquierda);
69      eliminarSubarbol(persona->derecha);
70
71      for (int i = 0; i < personas.size(); i++) {
72          if (personas[i] == persona) {
73              personas.erase(personas.begin() + i);
74              break;
75          }
76      }
77      delete persona;
78  }
```

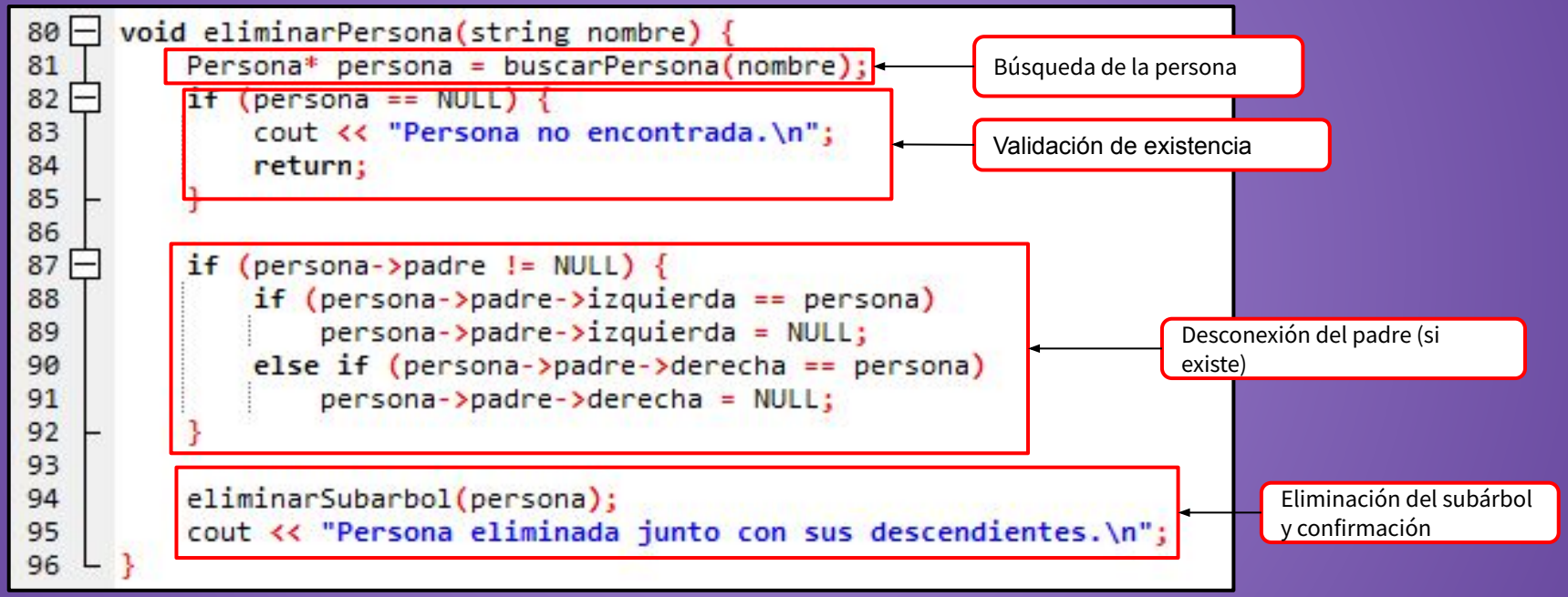
Eliminación recursiva de hijos

Eliminación del registro en el vector global

1.**Eliminación en profundidad (post-order):**

- Primero elimina hijos izquierdo y derecho (recursivamente).
- Luego elimina el nodo actual.

La función elimina una persona y todo su subárbol de descendientes, manteniendo la integridad de la estructura del árbol.



Una función que verifica y solicita al usuario seleccionar una opción mediante un número del (1 al 6), asegurándose de gestionar de manera efectiva cualquier error o entrada inválida.

```
98  int obtenerOpcionValida() {
99      int opcion;
100     while (true) {
101         cout << "Opcion: ";
102         if (cin >> opcion) {
103             if (opcion >= 1 && opcion <= 6) {
104                 cin.ignore(); // Limpiar el buffer(memori alamacenada)
105                 return opcion;
106             } else {
107                 cout << "Por favor elija una opcion valida del menu (1-6)...\n";
108             }
109         } else {
110             cout << "Entrada invalida. Por favor ingrese un numero del menu...\n";
111             cin.clear(); // Limpiar el estado de error
112             while (cin.get() != '\n'); // Limpia el buffer(memori alamacenada)
113         }
114     }
115 }
```

Interfaz amigable:

1.Detecta entradas no numéricas
2.Recupera el flujo de entrada después de errores
3.Limpieza completa del buffer cuando hay error
4.Mensajes claros para el usuario
5.Bucle continuo hasta obtener entrada válida

La función solicita y verifica que la edad ingresada por el usuario sea un número no negativo, incluyendo un manejo adecuado de errores para entradas incorrectas o inválidas.

```
117 int obtenerEdadValida() {
118     int edad;
119     while (true) {
120         cout << "Edad: ";
121         if (cin >> edad) {
122             if (edad >= 0) {
123                 cin.ignore(); // Limpiar el buffer(memori alamacenada)
124                 return edad;
125             } else {
126                 cout << "La edad no puede ser negativa. Por favor ingrese un valor valido.\n";
127             }
128         } else {
129             cout << "Entrada invalida. Por favor ingrese un numero valido.\n";
130             cin.clear();
131             while (cin.get() != '\n');
132         }
133     }
134 }
```

Validación estricta:
1.Solo acepta valores enteros no negativos (edad ≥ 0)
-Manejo robusto de errores:
2.Detecta entradas no numéricas (ejemplo: "abc")
-Rechaza valores negativos
-Recupera el flujo de entrada tras errores
3.Limpieza de buffer:
-"cin.ignore()" tras lectura exitosa
-Limpieza completa con "cin.clear()" + "bucle" cuando hay error
4.Interfaz clara:
-Mensajes de error específicos
-Bucle continuo hasta entrada válida

```cpp
int main() {
    int opcion;
    string nombre, nombrePadre;
    int edad;
    Persona* raiz = NULL;

    do {
        cout << "\n--- Menu de Arbol Genealogico ---\n";
        cout << "1. Agregar persona(s)\n";
        cout << "2. Establecer relacion padre-hij@\n";
        cout << "3. Mostrar ancestros\n";
        cout << "4. Mostrar descendientes\n";
        cout << "5. Eliminar persona\n";
        cout << "6. Salir\n";

        opcion = obtenerOpcionValida();

        if (opcion == 1) {
            do {
                cout << "\nIngrese el nombre (o 0 para terminar de ingresar datos): ";
                getline(cin, nombre);

                if (nombre == "0") break;

                edad = obtenerEdadValida();

                Persona* nueva = crearPersona(nombre, edad);
                if (raiz == NULL) raiz = nueva;
                cout << "Persona registrada.\n";

            } while (true);

        } else if (opcion == 2) {
            cout << "Nombre del padre: ";
            getline(cin, nombrePadre);
            Persona* padre = buscarPersona(nombrePadre);
            if (!padre) {
                cout << "Padre no encontrado.\n";
                continue;
            }
            cout << "Nombre del hij@: ";
            getline(cin, nombre);
            Persona* hijo = buscarPersona(nombre);
            if (!hijo) {
                cout << "Hij@ no encontrado.\n";
                continue;
            }
            agregarHijo(padre, hijo);
        } else if (opcion == 3) {
            cout << "Nombre: ";
            getline(cin, nombre);
            Persona* persona = buscarPersona(nombre);
            if (!persona) cout << "No encontrada.\n";
            else mostrarAncestros(persona);
        } else if (opcion == 4) {
            cout << "Nombre: ";
            getline(cin, nombre);
            Persona* persona = buscarPersona(nombre);
            if (!persona) cout << "No encontrada.\n";
            else mostrarDescendientes(persona);
        } else if (opcion == 5) {
            cout << "Nombre de la persona a eliminar: ";
            getline(cin, nombre);
            eliminarPersona(nombre);
        }

    } while (opcion != 6);

    cout << "Cerrando el sistema.Hasta la proxima[Presionar enter]...\n";
    return 0;
}
```

Menú interactivo para gestionar un árbol genealógico.

FIN