Actividad - Arboles

Pontificia Universidad Javeriana

Estructura de datos

Juan Manuel López Vargas

Sebastián Almanza Galvis

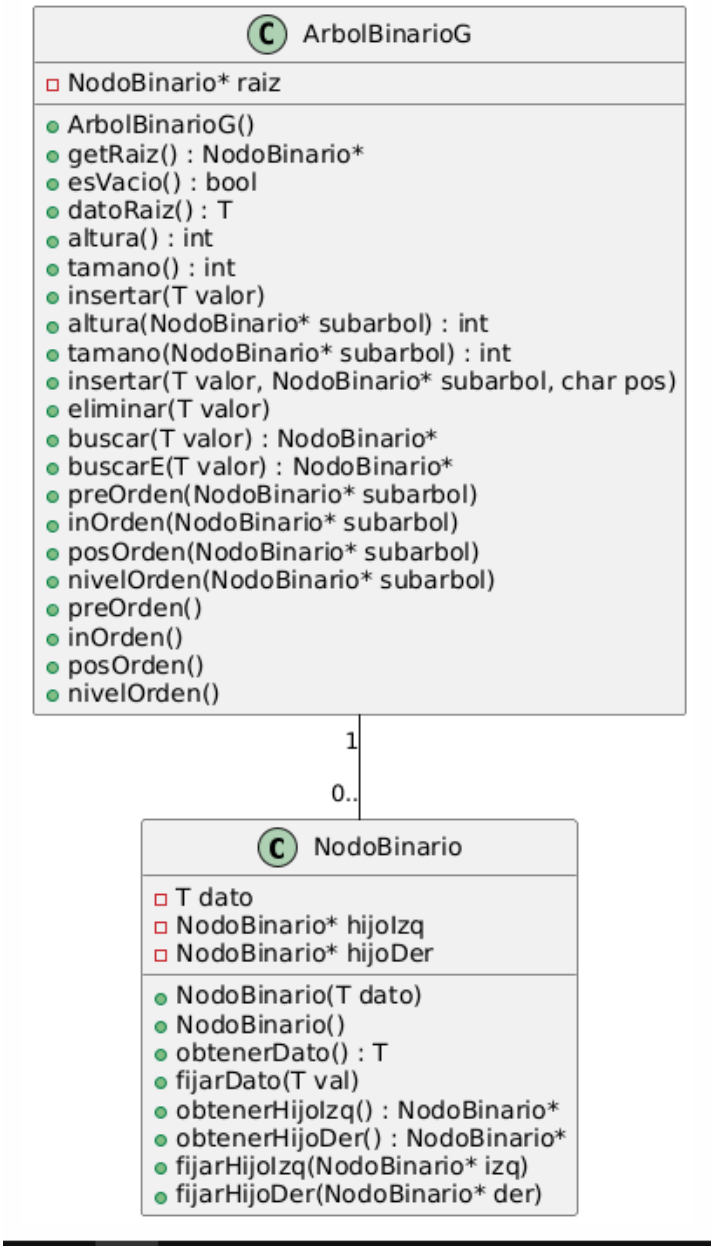
David León Velásquez

Juan Sebastián Méndez

Profesor: John Corredor

4 de octubre 2024

Árbol Binario:



El árbol binario de búsqueda es una estructura de datos en la que cada nodo tiene un valor y dos hijos, siendo el hijo izquierdo siempre menor que el nodo y el hijo derecho siempre mayor. El programa proporciona una interfaz de menú que permite al usuario realizar varias operaciones en el árbol binario. A continuación, se explica el funcionamiento de los principales componentes y opciones del programa:

Archivos del Proyecto

• NodoBinario.h: Define la clase NodoBinario, que representa un nodo en el árbol binario. Cada nodo tiene un valor (dato) y dos punteros a sus hijos izquierdo y derecho (hijoIzq y hijoDer). La clase incluye métodos para obtener y establecer el valor y los hijos, calcular la altura y el tamaño del subárbol, insertar nuevos nodos, buscar nodos y realizar recorridos del árbol en diferentes órdenes.

• NodoBinario.hxx: Implementa los métodos definidos en NodoBinario.h. Aquí se define cómo se realizan las operaciones de inserción, búsqueda, y los recorridos (preorden, inorden, postorden y por nivel).

• ArbolBinario.h: Define la clase ArbolBinario, que representa el árbol binario en sí. La clase tiene un puntero a la raíz del árbol (raiz) y métodos para verificar si el árbol está vacío, obtener el dato de la raíz, calcular la altura y el tamaño del árbol, insertar y eliminar nodos, y realizar recorridos en diferentes órdenes.

• ArbolBinario.hxx: Implementa los métodos definidos en ArbolBinario.h. Aquí se define cómo se realizan las operaciones de inserción, eliminación y los recorridos del árbol.

• main.cpp: Contiene el punto de entrada del programa y proporciona un menú interactivo para el usuario. Permite al usuario insertar nodos en el árbol y realizar diferentes tipos de recorridos.

Funcionamiento del Programa

• Inicialización del Árbol: Al iniciar el programa, se crea un objeto de la clase ArbolBinario llamado arbolito. Este árbol se inicia vacío cada vez que el programa se ejecuta el programa.

• Menú Interactivo: El programa presenta un menú con las siguientes opciones:

• Insertar un nodo: Permite al usuario ingresar un valor para insertar en el árbol. El valor se inserta siguiendo las reglas del árbol binario de búsqueda.

• Recorrido en Preorden: Muestra los valores de los nodos del árbol en el orden de preorden (nodo, hijo izquierdo, hijo derecho).

• Recorrido en Inorden: Muestra los valores de los nodos del árbol en el orden de inorden (hijo izquierdo, nodo, hijo derecho).

• Recorrido en Postorden: Muestra los valores de los nodos del árbol en el orden de postorden (hijo izquierdo, hijo derecho, nodo).

• Recorrido en NivelOrden: Muestra los valores de los nodos del árbol en el orden de nivel (nivel por nivel, de arriba hacia abajo).

• Operaciones en el Árbol:

• Insertar: Se agrega un nuevo nodo al árbol. Si el árbol está vacío, el nuevo nodo se convierte en la raíz. De lo contrario, el nodo se inserta en la posición correcta según las reglas del árbol binario de búsqueda.

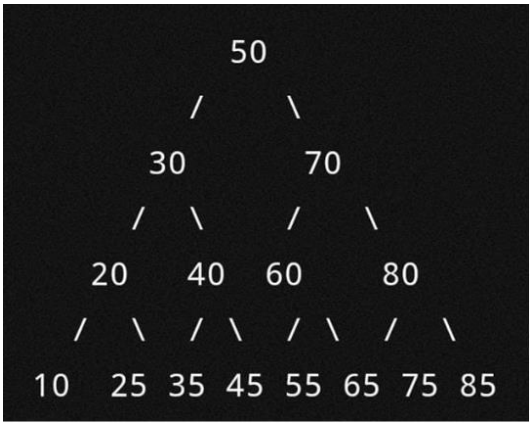
• Recorridos: Dependiendo de la opción seleccionada, el programa realiza un recorrido del árbol y muestra los valores de los nodos en el orden especificado.

• Salir del Programa: La opción 0 permite al usuario salir del programa.

Ejemplo de Ejecución:

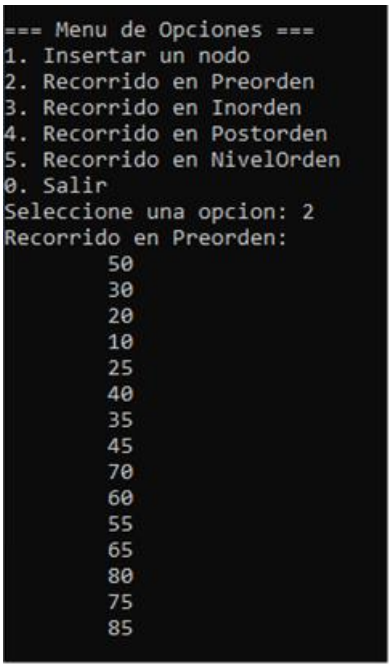
Tomemos de ejemplo la siguiente entrada al programa

: 50, 30, 70, 20, 40, 60, 80, 10, 25, 35, 45, 55, 65, 75, 85



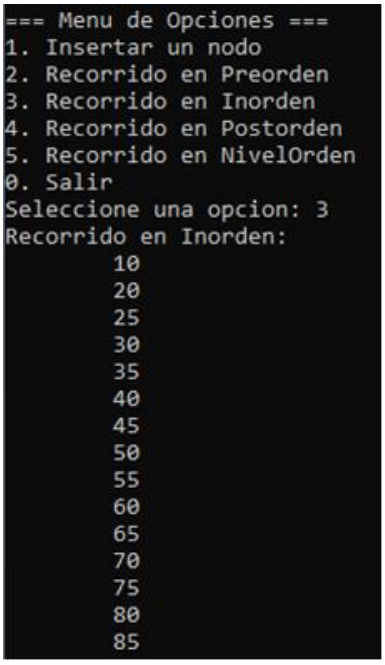
- Resultado esperado en recorrido Preorder: 50, 30, 20, 10, 25, 40, 35, 45, 70, 60, 55, 65, 80, 75, 85

Resultado:



- Resultado esperado en recorrido Inorder: 10, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75, 80, 85

Resultado:



- Resultado esperado en recorrido Postorder: 10, 25, 20, 35, 45, 40, 30, 55, 65, 60, 75, 85, 80, 70, 50

Resultado:

```
=== Menu de Opciones ===
1. Insertar un nodo
2. Recorrido en Preorden
3. Recorrido en Inorden
4. Recorrido en Postorden
5. Recorrido en NivelOrden
0. Salir
Seleccione una opcion: 4
Recorrido en Postorden:
        10
        25
        20
        35
        45
        40
        30
        55
        65
        60
        75
        85
        80
        70
        50
```

- Resultado esperado en recorrido Nivelorder: 50, 30, 70, 20, 40, 60, 80, 10, 25, 35, 45, 55, 65, 75, 85

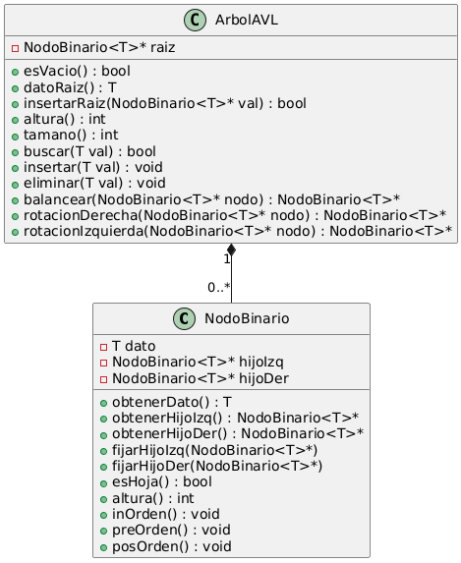
Resultado:

```
=== Menu de Opciones ===
1. Insertar un nodo
2. Recorrido en Preorden
3. Recorrido en Inorden
4. Recorrido en Postorden
5. Recorrido en NivelOrden
0. Salir
Seleccione una opcion: 5
Recorrido en NivelOrden:
        50
        30
        70
        20
        40
        60
        80
        10
        25
        35
        45
        55
        65
        75
        85
```

Concluciones:

Los resultados de ejecucion del codigo fueron los esperados, comprobando que los archivos dados por el profesor fueron correctamente implementados y mejorados. Ademas se implemento una funcion main con interfaz para el usuario para que el programa sea facil de entender y para que cualquiera pueda jugar y apropiarse del concepto de arbol binario.

Árbol AVL:



Un Árbol AVL es un árbol binario de búsqueda auto-balanceado en el que la diferencia entre las alturas de los subárboles izquierdo y derecho de cualquier nodo no puede ser mayor a uno. Si en algún momento la diferencia de alturas supera uno, el árbol se reequilibra automáticamente usando rotaciones (derecha, izquierda, derecha-izquierda, izquierda-derecha).

Componentes principales de la implementación del árbol AVL:

Clase NodoBinario:

Representa un nodo en el árbol AVL.

Contiene los datos (dato), punteros al hijo izquierdo (hijoIzq) y al hijo derecho (hijoDer).

Métodos para obtener y fijar los nodos hijos (obtenerHijoIzq, fijarHijoIzq, etc.), verificar si un nodo es una hoja (esHoja), y calcular la altura del nodo (altura).

Métodos de recorrido como inOrden, preOrden, y posOrden.

Clase ArbolAVL:

Representa el árbol AVL completo.

Operaciones principales:

Inserción (insertarRec, insertar): Se asegura de que después de la inserción, el árbol permanezca balanceado.

Eliminación (eliminarRec, eliminar): Elimina un nodo manteniendo las propiedades AVL.

Balanceo (balanceo, balancear): Garantiza que el árbol se mantenga balanceado calculando el factor de balanceo y aplicando las rotaciones adecuadas.

Rotaciones (rotacionIzquierda, rotacionDerecha): Son las operaciones fundamentales para mantener el balance del árbol tras inserciones y eliminaciones.

Métodos de recorrido del árbol (preOrden, inOrden, posOrden, nivelOrden).

Factor de Balanceo: La función balanceo calcula el factor de balanceo de un nodo, que es la diferencia entre la altura de los subárboles izquierdo y derecho. Si esta diferencia excede uno, el árbol se reequilibra.

Rotaciones:

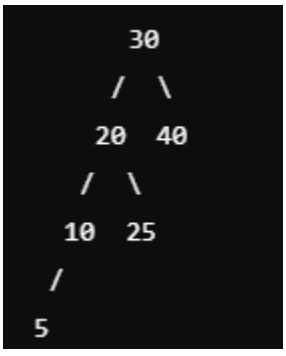
Rotación Derecha: Cuando el subárbol izquierdo es más alto que el subárbol derecho por más de uno.

Rotación Izquierda: Cuando el subárbol derecho es más alto que el subárbol izquierdo por más de uno.

Rotación Izquierda-Derecha: Cuando el hijo izquierdo es más pesado hacia la derecha.

Rotación Derecha-Izquierda: Cuando el hijo derecho es más pesado hacia la izquierda.

Se le ingresa el siguiente árbol:



Verificación del balanceo:

La altura del subárbol izquierdo de 30 es 3 (camino: 30 → 20 → 10 → 5), y la altura del subárbol derecho es 1 (nodo 40).

La diferencia de alturas en el nodo raíz (30) es 3 - 1 = 2, lo cual significa que el árbol está desbalanceado en este punto.

Balanceo del árbol:

El nodo 30 tiene un factor de balanceo de 2 (altura izquierda - altura derecha = 3 - 1), lo que indica que el subárbol izquierdo es más alto que el derecho.

El nodo 20 tiene un factor de balanceo de 1 (la diferencia entre las alturas de sus subárboles izquierdo y derecho).

Rotación necesaria: Como el árbol está desbalanceado hacia la izquierda (el subárbol izquierdo de 30 es más alto), necesitamos realizar una rotación derecha en el nodo 30.

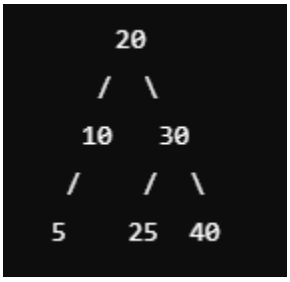
Rotación derecha en 30:

En esta rotación, 20 se convierte en la nueva raíz del subárbol.

30 se convierte en el hijo derecho de 20.

El subárbol derecho de 20 (nodo 25) permanece como el hijo izquierdo de 30.

El balanceo quedaría de la siguiente manera:



Conclusión:

Después de insertar el valor 5, el árbol AVL se desbalanceó, y la rotación derecha en el nodo 30 fue suficiente para reequilibrarlo. Así, el árbol sigue siendo un árbol de búsqueda binaria balanceado, manteniendo su propiedad AVL.

Quad-tree:

1. **Definición**

Un Quadtree es una estructura de datos de árbol que se utiliza para dividir un espacio en partes más pequeñas. Se utiliza comúnmente en aplicaciones de gráficos, sistemas de información geográfica y procesamiento de imágenes. Cada nodo del quadtree puede tener hasta cuatro hijos, cada uno representando un cuadrante del espaci

2. **TADS**:

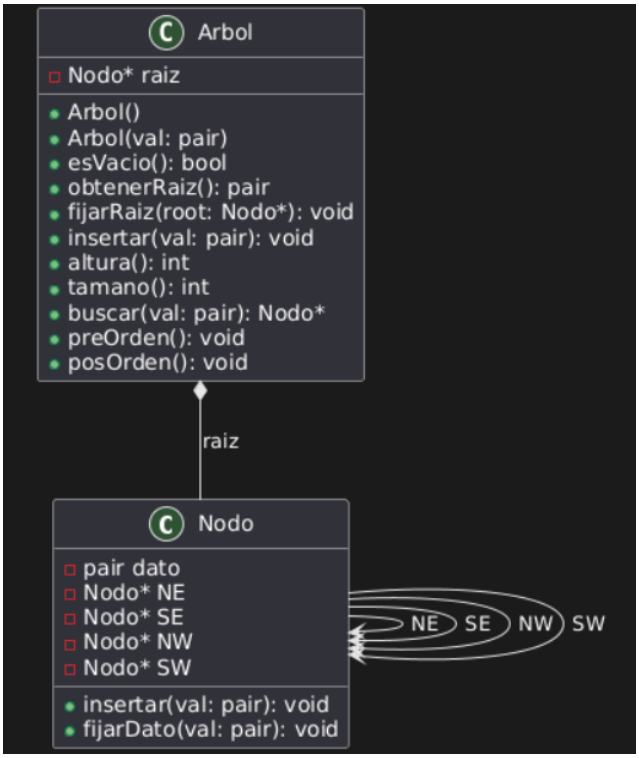
TAD Nodo<T>

- **Descripción**: Representa un nodo en el quadtree.
- **Atributos**:
    - pair<T,T> dato: Almacena el dato del nodo, que representa un punto en el espacio.
    - Nodo* NE: Apunta al nodo noreste.
    - Nodo* SE: Apunta al nodo sureste.
    - Nodo* NW: Apunta al nodo noroeste.
    - Nodo* SW: Apunta al nodo suroeste.
- **Métodos**:
    - void insertar(pair<T,T> val): Inserta un nuevo valor en el quadtree, creando nodos hijos según corresponda.
    - void fijarDato(pair<T,T> val): Establece el valor del dato del nodo.

TAD Arbol<T>

- **Descripción**: Representa el quadtree en su conjunto.
- **Atributos**:
    - Nodo<T>* raiz: Apunta a la raíz del quadtree.
- **Métodos**:
    - Arbol(): Constructor que inicializa un árbol vacío.
    - Arbol(pair<T,T> val): Constructor que inicializa el árbol con un valor.
    - bool esVacio(): Devuelve verdadero si el árbol está vacío.
    - pair<T,T> obtenerRaiz(): Devuelve el valor de la raíz del árbol.
    - void fijarRaiz(Nodo<T>* root): Establece la raíz del árbol.
    - void insertar(pair<T,T> val): Inserta un nuevo valor en el quadtree.
    - int altura(): Devuelve la altura del quadtree.
    - int tamano(): Devuelve el tamaño del quadtree.
    - Nodo<T>* buscar(pair<T,T> val): Busca un nodo en el quadtree.
    - void preOrden(): Realiza un recorrido en preorden.
    - void posOrden(): Realiza un recorrido en postorden.

2.1 Diagrama de Tads:



3. Prueba del Algoritmo

Para verificar el funcionamiento del quadtree, se realizaron las siguientes pruebas:

1. Prueba de Inserción:

- Se insertaron los puntos: (10, 20), (15, 25) y (5, 30).

```
arbol.insertar({10, 20});
arbol.insertar({15, 25});
arbol.insertar({5, 30});
```

2. **Prueba de Tamaño**:

Se verifico que el tamaño del quadtree se actualice correctamente:

```
Tamaño del árbol: 4
```

4. **Prueba de Búsqueda:**

- Se buscó el valor (15, 25) y se encontró correctamente.

```
Valor encontrado: (15, 25)
```

5. **Pruebas de Recorridos**:

- Se realizaron recorridos en preorden y postorden.

```
Recorrido preorden:
(10,20)
(5,30)
(15,25)
Recorrido posorden:
(5,30)
(15,25)
(10,20)
```

Árbol Binario Ordenado

Estructura del Proyecto

El proyecto consta de los siguientes archivos:

- **NodoBinario.h**: Declara la clase NodoBinario, que representa un nodo en el árbol binario.
- **NodoBinario.hxx**: Define las implementaciones de los métodos declarados en NodoBinario.h.
- **ArbolBinario.h**: Declara la clase ArbolBinario, que contiene la estructura y las operaciones del árbol binario.
- **ArbolBinario.hxx**: Define las implementaciones de los métodos declarados en ArbolBinario.h.
- **Main.cpp**: Archivo principal donde se realizan pruebas de las funcionalidades del árbol binario.

Descripción de Clases y Métodos

NodoBinario

- **Archivos**: NodoBinario.h y NodoBinario.hxx
- **Descripción**: Clase plantilla que representa un nodo en el árbol binario. Contiene un dato genérico y punteros a los nodos hijos izquierdo y derecho.

Métodos Principales:

- NodoBinario(): Constructor. Inicializa los punteros de los hijos a NULL.
- ~NodoBinario(): Destructor.
- T& getDato(): Devuelve una referencia al dato del nodo.
- void setDato(T& val): Establece el dato del nodo.
- NodoBinario<T>* getHijoIzq(): Devuelve un puntero al hijo izquierdo.
- NodoBinario<T>* getHijoDer(): Devuelve un puntero al hijo derecho.
- void setHijoIzq(NodoBinario<T>* izq): Asigna el hijo izquierdo.
- void setHijoDer(NodoBinario<T>* der): Asigna el hijo derecho.

ArbolBinario

- **Archivos**: ArbolBinario.h y ArbolBinario.hxx
- **Descripción**: Clase plantilla que representa el árbol binario completo, con funciones para manipular y acceder a los nodos.

Métodos Principales:

- ArbolBinario(): Constructor. Inicializa la raíz a NULL.
- ~ArbolBinario(): Destructor.
- bool esVacio(): Verifica si el árbol está vacío.
- T& datoRaiz(): Devuelve el dato en la raíz del árbol.
- int altura(NodoBinario<T>* inicio): Calcula la altura del árbol a partir de un nodo dado.
- int tamano(NodoBinario<T>* inicio): Calcula el tamaño del árbol (número de nodos).
- bool insertar(T& val, NodoBinario<T>* nodo): Inserta un valor en el árbol en la posición correcta.
- bool eliminar(T& val): Elimina un valor del árbol. (Implementación opcional)
- bool buscar(T& val): Busca un valor en el árbol.
- void preOrden(NodoBinario<T>* inicio): Recorre el árbol en PreOrden.
- void inOrden(NodoBinario<T>* inicio): Recorre el árbol en InOrden.

- void posOrden(NodoBinario<T>* inicio): Recorre el árbol en PosOrden.

Main.cpp

- **Descripción**: Realiza pruebas de las funciones de ArbolBinario. Se incluyen las siguientes pruebas:
  - pruebaInsercion: Inserta varios valores y verifica la estructura del árbol.
  - pruebaRecorridos: Realiza los recorridos PreOrden, InOrden y PosOrden.
  - pruebaBusqueda: Busca un valor específico en el árbol.
  - pruebaEliminacion: Elimina un valor y muestra el árbol actualizado.

Ejemplo de Uso en Main.cpp

El archivo Main.cpp contiene la lógica de prueba, que utiliza el árbol binario para demostrar las funcionalidades descritas. Puedes modificar este archivo para añadir más pruebas o ajustar el comportamiento de las funciones del árbol.

Estructura del Proyecto

El proyecto consta de los siguientes archivos:

NodoBinario.h: Declara la clase NodoBinario, que representa un nodo en el árbol binario.

NodoBinario.hxx: Define las implementaciones de los métodos declarados en NodoBinario.h.

ArbolBinario.h: Declara la clase ArbolBinario, que contiene la estructura y las operaciones del árbol binario.

ArbolBinario.hxx: Define las implementaciones de los métodos declarados en ArbolBinario.h.

Main.cpp: Archivo principal donde se realizan pruebas de las funcionalidades del árbol binario.

3. Descripción de Clases y Métodos

NodoBinario

Archivos: NodoBinario.h y NodoBinario.hxx

Descripción: Clase plantilla que representa un nodo en el árbol binario. Contiene un dato genérico y punteros a los nodos hijos izquierdo y derecho.

Métodos Principales:

NodoBinario(): Constructor. Inicializa los punteros de los hijos a NULL.

~NodoBinario(): Destructor.

T& getDato(): Devuelve una referencia al dato del nodo.

void setDato(T& val): Establece el dato del nodo.

NodoBinario<T>* getHijoIzq(): Devuelve un puntero al hijo izquierdo.

NodoBinario<T>* getHijoDer(): Devuelve un puntero al hijo derecho.

void setHijoIzq(NodoBinario<T>* izq): Asigna el hijo izquierdo.

void setHijoDer(NodoBinario<T>* der): Asigna el hijo derecho.

ArbolBinario

Archivos: ArbolBinario.h y ArbolBinario.hxx

Descripción: Clase plantilla que representa el árbol binario completo, con funciones para manipular y acceder a los nodos.

Métodos Principales:

ArbolBinario(): Constructor. Inicializa la raíz a NULL.

~ArbolBinario(): Destructor.

bool esVacio(): Verifica si el árbol está vacío.

T& datoRaiz(): Devuelve el dato en la raíz del árbol.

int altura(NodoBinario<T>* inicio): Calcula la altura del árbol a partir de un nodo dado.

int tamano(NodoBinario<T>* inicio): Calcula el tamaño del árbol (número de nodos).

bool insertar(T& val, NodoBinario<T>* nodo): Inserta un valor en el árbol en la posición correcta.

bool eliminar(T& val): Elimina un valor del árbol. (Implementación opcional)

bool buscar(T& val): Busca un valor en el árbol.

void preOrden(NodoBinario<T>* inicio): Recorre el árbol en PreOrden.

void inOrden(NodoBinario<T>* inicio): Recorre el árbol en InOrden.

void posOrden(NodoBinario<T>* inicio): Recorre el árbol en PosOrden.

Main.cpp

Descripción: Realiza pruebas de las funciones de ArbolBinario. Se incluyen las siguientes pruebas:

pruebaInsercion: Inserta varios valores y verifica la estructura del árbol.

pruebaRecorridos: Realiza los recorridos PreOrden, InOrden y PosOrden.

pruebaBusqueda: Busca un valor específico en el árbol.

pruebaEliminacion: Elimina un valor y muestra el árbol actualizado.

4. Ejemplo de Uso en Main.cpp

El archivo Main.cpp contiene la lógica de prueba, que utiliza el árbol binario para demostrar las funcionalidades descritas. Puedes modificar este archivo para añadir más pruebas o ajustar el comportamiento de las funciones del árbol.

5. Consideraciones Finales

Gestión de memoria: Asegúrate de manejar adecuadamente la memoria, especialmente al insertar y eliminar nodos, para evitar fugas de memoria.

Modularidad: Los archivos están estructurados para facilitar la comprensión y permitir cambios independientes en la implementación de nodos y del árbol.

- Este proyecto está diseñado para entender las operaciones básicas de un árbol binario.
- **Gestión de memoria**: Asegúrate de manejar adecuadamente la memoria, especialmente al insertar y eliminar nodos, para evitar fugas de memoria.
- **Modularidad**: Los archivos están estructurados para facilitar la comprensión y permitir cambios independientes en la implementación de nodos y del árbol.

KD-Tree:

1. TAD Punto

Este TAD define un punto en un plano bidimensional, utilizando dos coordenadas: x e y. Además, incluye varias operaciones para comparar puntos y calcular la distancia entre ellos.

Atributos esenciales:

x: Representa la coordenada en el eje X.

y: Representa la coordenada en el eje Y.

Operaciones principales:

Punto& operator=(const Punto &p): Permite asignar un punto a otro.

bool operator==(const Punto &p) const: Verifica si dos puntos son iguales.

int distanciaEuclidiana(Punto val2): Calcula la distancia euclidiana entre este punto y otro.

friend std::ostream& operator<<(std::ostream &o, const Punto &p): Sobrecarga del operador de salida para facilitar la impresión de un punto.

Propósito: Este TAD encapsula las propiedades y funcionalidades básicas de un punto, esenciales para su manipulación y operaciones en estructuras como el KD-Tree, como la inserción y búsqueda.

2. TAD NodoKD

Este TAD modela un nodo dentro de un árbol KD. Cada nodo contiene un punto (del TAD Punto) y punteros a sus nodos hijo izquierdo y derecho.

Atributos esenciales:

Punto dato: El punto almacenado en el nodo.

NodoKD *hijoIzq: Puntero al nodo hijo izquierdo.

NodoKD *hijoDer: Puntero al nodo hijo derecho.

Operaciones principales:

NodoKD(): Constructor por defecto.

NodoKD(Punto val): Inicializa el nodo con un punto específico.

~NodoKD(): Destructor que gestiona la memoria del nodo.

bool esHoja(): Verifica si el nodo es una hoja, es decir, si no tiene hijos.

Punto obtenerDato(): Obtiene el punto almacenado en el nodo.

void fijarDato(Punto val): Asigna un nuevo valor de punto al nodo.

NodoKD* obtenerHijoIzq(): Retorna el puntero al hijo izquierdo.

NodoKD* obtenerHijoDer(): Retorna el puntero al hijo derecho.

void fijarHijoIzq(NodoKD *izq): Establece el puntero al nuevo hijo izquierdo.

void fijarHijoDer(NodoKD *der): Establece el puntero al nuevo hijo derecho.

Propósito: El TAD NodoKD es una unidad estructural fundamental en el árbol KD. Almacena un punto y las referencias a los hijos, lo que permite organizar los datos de manera jerárquica.

3. TAD ArbolKD

Este TAD representa el árbol KD en su totalidad. Controla la organización de los nodos y las operaciones clave como la inserción, búsqueda y recorridos.

Atributos esenciales:

NodoKD *raiz: Puntero al nodo raíz del árbol.

Operaciones principales:

ArbolKD(): Constructor que inicializa el árbol vacío.

ArbolKD(Punto val): Constructor que inicializa el árbol con un nodo raíz.

~ArbolKD(): Destructor que libera los recursos del árbol.

Punto datoRaiz(): Retorna el punto almacenado en la raíz.

NodoKD* obtenerRaiz(): Devuelve el nodo raíz.

void fijarRaiz(NodoKD *n_raiz): Establece un nuevo nodo raíz.

bool esVacio(): Comprueba si el árbol está vacío.

bool insertar(Punto val): Inserta un nuevo punto en el árbol.

NodoKD* insertarRec(NodoKD *nodo, Punto val, bool &insertado, char dimension): Inserta recursivamente un nodo en el árbol.

NodoKD* vecinoCercano(NodoKD *raiz, Punto val): Encuentra el punto más cercano al punto proporcionado.

void vecinoCercanoRec(NodoKD *nodo, Punto val, char dimension, NodoKD *&mejorNodo, int &mejorDist): Método recursivo para buscar el vecino más cercano en el árbol.

Recorridos:

void preOrden(): Realiza un recorrido en preorden.

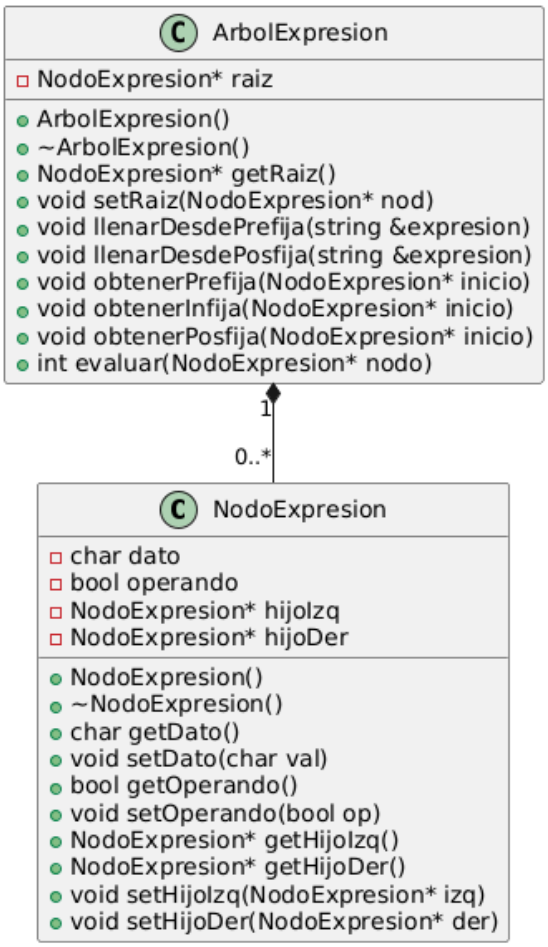
void inOrden(): Realiza un recorrido inorden.

void posOrden(): Realiza un recorrido posorden.

void nivelOrden(): Realiza un recorrido por niveles.

Propósito: El TAD ArbolKD organiza y gestiona la estructura jerárquica del árbol KD, facilitando la inserción de puntos, búsquedas eficientes, y la ejecución de diferentes tipos de recorridos en el árbol.

Árbol de expresión:



Clase ArbolExpresion:

Es una clase que gestiona un árbol de expresión.

Atributos:

NodoExpresion* raiz: Un puntero que apunta al nodo raíz del árbol.

Métodos:

ArbolExpresion(): Constructor por defecto que inicializa el árbol vacío.

~ArbolExpresion(): Destructor.

NodoExpresion* getRaiz(): Retorna el nodo raíz del árbol.

void setRaiz(NodoExpresion* nod): Establece el nodo raíz.

void llenarDesdePrefija(string &expresion): Llena el árbol usando una expresión en notación prefija.

void llenarDesdePosfija(string &expresion): Llena el árbol usando una expresión en notación posfija.

Recorridos:

void obtenerPrefija(NodoExpresion* inicio): Imprime la expresión en notación prefija.

void obtenerInfija(NodoExpresion* inicio): Imprime la expresión en notación infija.

void obtenerPosfija(NodoExpresion* inicio): Imprime la expresión en notación posfija.

int evaluar(NodoExpresion* nodo): Evalúa la expresión almacenada en el árbol, comenzando desde el nodo especificado.

Clase NodoExpresion:

Representa un nodo del árbol de expresión.

Atributos:

char dato: Contiene el operador o el operando.

bool operando: Indica si el nodo es un operando.

NodoExpresion* hijoIzq: Puntero al hijo izquierdo.

NodoExpresion* hijoDer: Puntero al hijo derecho.

Métodos:

NodoExpresion(): Constructor por defecto.

~NodoExpresion(): Destructor.

char getDato(): Retorna el valor almacenado en el nodo.

void setDato(char val): Establece un nuevo valor en el nodo.

bool getOperando(): Retorna si el nodo es un operando.

void setOperando(bool op): Establece si el nodo es un operando.

NodoExpresion* getHijoIzq(): Retorna el hijo izquierdo del nodo.

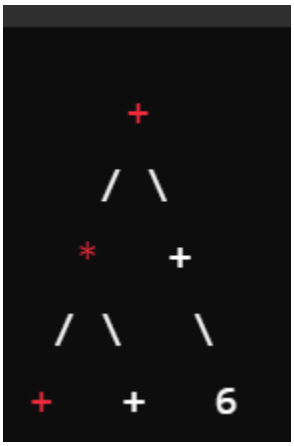
NodoExpresion* getHijoDer(): Retorna el hijo derecho del nodo.

void setHijoIzq(NodoExpresion* izq): Establece el hijo izquierdo del nodo.

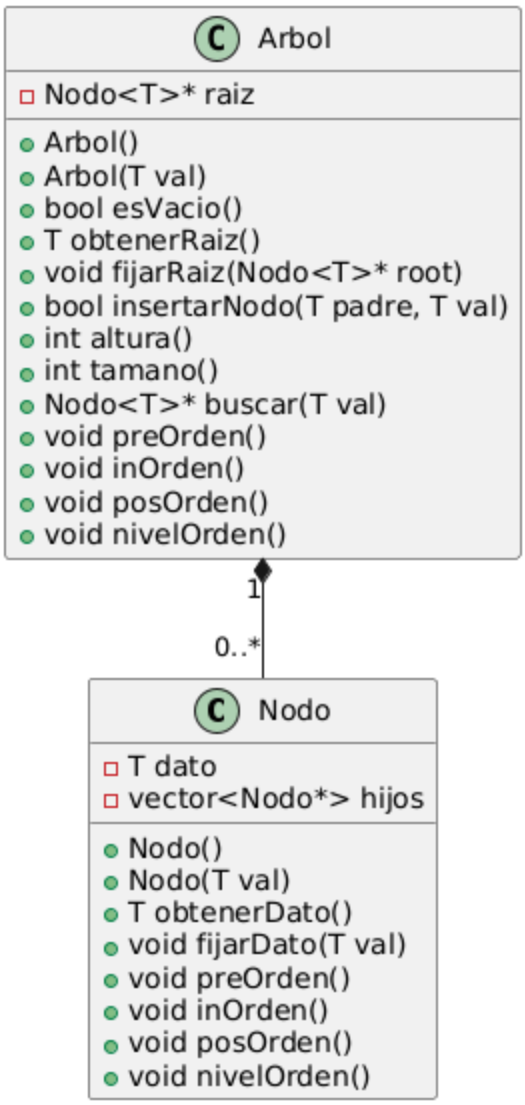
void setHijoDer(NodoExpresion* der): Establece el hijo derecho del nodo.

Prueba:

Leer la expresión posfija: "45+23+*6+87+/12+3*6+23+/*"



Árbol:



Clase Árbol:

Es una clase genérica que representa un árbol con un nodo raíz.

Atributos:

Nodo<T> *raiz: Un puntero que apunta al nodo raíz del árbol.

Métodos:

Arbol(): Constructor por defecto que inicializa el árbol vacío.

Arbol(T val): Constructor que inicializa el árbol con un nodo raíz que contiene el valor val.

bool esVacio(): Verifica si el árbol está vacío, retornando true si no hay raíz.

T obtenerRaiz(): Retorna el valor almacenado en el nodo raíz.

void fijarRaiz(Nodo<T>* root): Establece un nuevo nodo raíz.

bool insertarNodo(T padre, T val): Inserta un nuevo nodo como hijo de un nodo padre existente.

int altura(): Calcula y retorna la altura del árbol.

int tamano(): Retorna el número de nodos en el árbol.

Nodo<T>* buscar(T val): Busca y retorna el nodo que contiene el valor val.

Recorridos:

void preOrden(): Realiza un recorrido en preorden.

void inOrden(): Realiza un recorrido inorden.

void posOrden(): Realiza un recorrido en postorden.

void nivelOrden(): Realiza un recorrido por niveles.

Clase Nodo:

Representa un nodo en el árbol.

Atributos:

T dato: El valor almacenado en el nodo.

std::vector<Nodo<T>*> hijos: Un vector que contiene los hijos del nodo.

Métodos:

Nodo(): Constructor por defecto.

Nodo(T val): Inicializa un nodo con el valor val.

T obtenerDato(): Retorna el valor almacenado en el nodo.

void fijarDato(T val): Establece un nuevo valor en el nodo.

Recorridos:

void preOrden(): Recorre el subárbol en preorden.

void inOrden(): Recorre el subárbol en inorden.

void posOrden(): Recorre el subárbol en postorden.

void nivelOrden(): Recorre el subárbol por niveles.

Prueba:

Insertar los nodos 6, 7, y 8 como hijos de 5

Insertar nodos 9 y 10 como hijos de 6

Insertar nodo 11 como hijo de 7