

Taller 03: Arboles (TAD, Funcionamiento Y Prueba)

Pontificia Universidad Javeriana



Estructuras De Datos

Autores:

Santiago Hernandez

Juan Esteban Bello

John Corredor

27 de marzo 2025

Índice:

Objetivos:	4
Introducción:	5
Desarrollo:.....	6
Análisis Árbol general	6
1.1 TAD Árbol General.....	6
1.2 Descripción del programa	9
1.3 Errores identificados y correcciones	10
1.4 Plan de pruebas.....	11
1.5 Conclusiones de este árbol	13
2. Análisis Árbol AVL.....	13
2.1 TAD Nodo Binario AVL	13
2.2 Descripción del programa	16
2.3 Errores identificados	17
2.4 Plan de pruebas.....	19
2.5 Conclusiones del programa	23
3. Análisis Árbol Binario Ordenado	24
3.1 TAD 'S Programa	24
3.2 Descripción del programa	26
3.3 Errores identificados	26
3.4 Plan de pruebas.....	28
3.5 Conclusiones	30
4. Análisis Quad-Tree.....	31
4.1 TAD 'S Programa	31
4.2 Descripción del programa	33
4.3 Errores identificados	33
4.4 Plan de pruebas.....	35
4.5 Conclusiones	37
5. Análisis Árbol Binario	37
5.1 Tad's Programa	37
5.2 Descripción del programa	41
5.3 Errores identificados	41

5.4 Plan de pruebas.....	43
5.5 Conclusiones	45
6. Árbol Expresión	46
6.1 TAD ´S Programa	46
6.2 Descripción del programa	48
6.3 Errores identificados	49
6.4 Plan de pruebas.....	51
6.5 Conclusiones	53
7. Árbol KD-tree.....	53
7.1 TAD ´S Programa	53
7.2 Descripción del programa	56
7.3 Errores identificados	56
7.4 Plan de pruebas.....	58
7.5 Conclusiones	60
8. Análisis de Red-Black tree	61
8.1 TAD ´S Programa	61
8.2 Descripción del programa	62
8.3 Errores identificados y correcciones	63
8.4 Plan de Pruebas.....	65
8.5 Conclusiones	66
Conclusiones Generales:.....	67

|

Objetivos:

Objetivo general

Analizar, corregir, documentar y validar el comportamiento de diferentes implementaciones de árboles de búsqueda, aplicando conceptos fundamentales de estructuras de datos para comprender su funcionamiento, identificar errores comunes y proponer soluciones que garanticen su correcto desempeño.

Objetivos específicos

- Comprender la estructura y funcionamiento de distintos tipos de árboles.
- Diseñar e interpretar los TAD (Tipos Abstractos de Datos) asociados a cada árbol, describiendo claramente sus componentes y operaciones.
- Identificar errores lógicos, estructurales o sintácticos en los códigos suministrados, justificando las correcciones aplicadas en cada caso.
- Realizar pruebas de escritorio y pruebas ejecutadas para validar que el comportamiento del árbol corresponde con los recorridos esperados y con su estructura lógica.
- Documentar detalladamente cada implementación, incluyendo definición de TADs, descripción del programa, plan de pruebas y conclusiones, siguiendo un enfoque técnico y estructurado.

Introducción:

Este informe presenta un análisis detallado sobre diversas estructuras de datos tipo árbol, enfocándose en corregir y validar sus implementaciones. Los árboles son estructuras esenciales en programación, muy útiles para organizar información jerárquicamente, facilitando tareas como insertar, buscar o recorrer datos de manera eficiente y ordenada.

A lo largo del desarrollo del taller se abordaron diferentes tipos de árboles, incluyendo árboles binarios de búsqueda, árboles AVL, árboles de expresión, QuadTrees, KD-Trees, árboles Red-Black y etc. Cada uno de ellos posee particularidades propias en cuanto a funcionamiento, balance y aplicación, lo que permitió realizar un estudio comparativo profundo y técnico.

El trabajo se desarrolló mediante la revisión cuidadosa del código fuente, detectando errores lógicos y de implementación, proponiendo soluciones prácticas, y realizando pruebas específicas para asegurar que cada estructura operara correctamente. Además, se definieron claramente los Tipos Abstractos de Datos (TADs) relacionados con cada árbol, documentando detalladamente sus componentes y operaciones, acompañados de un plan riguroso de pruebas.

Finalmente, este informe no solo busca demostrar el correcto funcionamiento de las implementaciones realizadas, sino también reforzar la comprensión tanto teórica como práctica de las estructuras jerárquicas. Asimismo, enfatiza la importancia del diseño modular, la validación sistemática de resultados y una documentación técnica clara y precisa.

Desarrollo:

Análisis Árbol general

1.1 TAD Árbol General

- **TAD Nodo**

El TAD Nodo representa la unidad básica de un árbol general. Cada nodo almacena un dato genérico de tipo T y mantiene una lista de punteros a sus nodos hijos, permitiendo estructuras jerárquicas con múltiples descendientes por nodo (a diferencia de un árbol binario que solo permite dos).

Datos Mínimos:

- **T (tipo genérico):** Es el valor que contiene el nodo. Puede ser un número, carácter u otro tipo, dependiendo del árbol que se esté implementando.
- **hijos: vector<Nodo<T>>:** Es una lista de nodos hijos (descendientes directos) que están conectados al nodo actual. Permite representar estructuras ramificadas donde un nodo puede tener varios hijos.

Operaciones:

- **Nodo():**
Constructor sin parámetros. Crea un nodo vacío, sin dato y sin hijos.
- **Nodo(T valor):**
Constructor con parámetro. Crea un nodo con el valor especificado en valor y sin hijos.
- **obtenerDato():**
Devuelve el valor contenido en el nodo actual.
- **fijarDato(T& val):**
Reemplaza el valor del nodo con el valor proporcionado en val.
- **adicionarDesc(T& val):**
Crea un nuevo nodo a partir del valor val y lo añade como hijo del nodo actual.

- **eliminarDesc(T& val):**

Busca entre los hijos del nodo actual un nodo con el valor val y, si lo encuentra, lo elimina. Retorna true si se elimina con éxito.

- **limpiarLista():**

Elimina todos los hijos del nodo, es decir, limpia su lista de descendientes.

- **buscar(T val):**

Realiza una búsqueda entre los descendientes del nodo actual y retorna un puntero al nodo que contiene el valor val. Si no se encuentra, retorna nullptr.

- **altura():**

Retorna la altura del subárbol que tiene como raíz el nodo actual. La altura es el número máximo de niveles desde este nodo hasta una hoja.

- **tamano():**

Retorna el número total de nodos que hay en el subárbol que tiene como raíz el nodo actual, incluyendo el nodo mismo.

- **preOrden():**

Imprime los valores del subárbol en recorrido preorden, es decir: primero el nodo actual, luego sus hijos en orden.

- **posOrden():**

Imprime los valores del subárbol en recorrido posorden: primero todos los hijos (recursivamente), y luego el nodo actual.

- **nivelOrden(queue<Nodo> &cola):**

Imprime los valores del subárbol en recorrido por niveles (también llamado recorrido en anchura). Utiliza una cola para recorrer el árbol nivel por nivel.

- **TAD Árbol**

Un Árbol es una estructura de datos jerárquica compuesta por nodos, donde cada nodo puede tener múltiples hijos (en el caso de un árbol general). Este TAD permite representar relaciones de tipo padre-hijo entre datos, siendo el nodo raíz el punto de entrada a toda la estructura.

Datos Mínimos:

- **raiz:** Es un puntero al nodo que representa la raíz del árbol. Desde este nodo se puede recorrer todo el árbol. Si es nullptr, el árbol está vacío.
- **Operaciones:**
- **Arbol():**
Constructor sin parámetros. Crea un árbol vacío, sin nodo raíz.
- **Arbol(T val):**
Constructor con parámetro. Crea un árbol cuya raíz contiene el valor val.
- **esVacio():**
Retorna true si el árbol está vacío (es decir, si no tiene una raíz), de lo contrario retorna false.
- **obtenerRaiz():**
Retorna un puntero al nodo raíz del árbol.
- ***fijarRaiz(Nodo<T> root)**:**
Asigna el nodo recibido como nueva raíz del árbol.
- **insertarNodo(T padre, T val):**
Busca en el árbol un nodo cuyo valor sea igual a padre. Si lo encuentra, crea un nuevo nodo con el valor val y lo inserta como hijo de dicho nodo padre. Retorna true si la operación fue exitosa.
- **eliminar(T val):**
Busca en el árbol un nodo cuyo valor sea igual a val y lo elimina del árbol si se encuentra. Retorna true si la eliminación fue exitosa.
- **buscar(T val):**
Busca un nodo con el valor val dentro del árbol y retorna un puntero a dicho nodo si lo encuentra. Si no se encuentra, retorna nullptr.
- **altura():**
Retorna la altura total del árbol, es decir, la altura del nodo raíz.
- **tamano():**
Retorna el número total de nodos que tiene el árbol completo.
- **preOrden():**
Imprime los valores del árbol en recorrido preorden (raíz, luego hijos).
- **posOrden():**
Imprime los valores del árbol en recorrido posorden (todos los hijos y luego la raíz).

- **nivelOrden():**

Imprime los valores del árbol por niveles, desde la raíz hasta los nodos hoja, utilizando una cola para recorrer el árbol nivel por nivel.

1.2 Descripción del programa

El programa tiene como objetivo demostrar el funcionamiento básico de un árbol general, utilizando un conjunto de operaciones fundamentales: creación del árbol, inserción de nodos y recorridos preorden y posorden. El programa está contenido en el archivo *prueba_arbol.cpp* y fue ejecutado en un entorno Linux Ubuntu desde el terminal, utilizando el compilador *g++*.

El árbol implementado no es binario ni balanceado, lo que significa que cada nodo puede tener un número arbitrario de hijos. Esta flexibilidad permite representar estructuras jerárquicas más generales, como organizaciones, sistemas de archivos o árboles de decisión complejos.

En el programa, se construye un árbol cuya raíz es el nodo con valor 5. A este nodo se le insertan tres hijos: 6, 7 y 8. Posteriormente, al nodo 6 se le agregan los hijos 9 y 10, y al nodo 7 se le agrega el hijo 11. Esta construcción permite verificar la correcta asignación jerárquica entre nodos y evaluar el funcionamiento de los recorridos.

El programa finaliza realizando dos tipos de recorrido sobre el árbol:

- **Recorrido en preorden:** visita primero el nodo raíz, luego recorre los hijos de izquierda a derecha en forma recursiva.
- **Recorrido en posorden:** recorre primero todos los hijos en forma recursiva, y finalmente visita el nodo actual.

Ambos recorridos se imprimen por pantalla, lo cual permite visualizar la estructura lógica del árbol y validar el comportamiento de la implementación.

Este programa constituye una prueba inicial del TAD Árbol implementado, permitiendo identificar errores, validar funcionalidades y generar evidencia experimental del correcto funcionamiento del sistema.

1.3 Errores identificados y correcciones

Durante el desarrollo del programa correspondiente al árbol general, se identificaron varios errores de declaración y omisión que no permitía su correcta compilación y ejecución. Se van a describen los principales problemas detectados en la versión original, así como las correcciones aplicadas en la versión final del código:

1. Falta de inclusión de librerías necesarias

En la definición del TAD Nodo se utilizaban estructuras como vector y queue sin incluir las librerías correspondientes. Esto generaba errores de compilación por referencias no reconocidas.

- **Corrección:** Se incluyeron explícitamente las librerías <vector> y <queue> en el archivo nodofinal.h.
- **Motivo:** Estas estructuras son fueron fundamentales para almacenar los hijos del nodo y para realizar el recorrido por niveles, respectivamente.

2. Declaración incorrecta de la función nivelOrden

En la versión original (nodo.h), la función nivelOrden() no recibía ningún parámetro. Sin embargo, su implementación dependía del uso de una cola externa para recorrer los nodos por niveles. Esto generaba una inconsistencia entre la declaración y la definición.

- **Corrección:** Se ajustó la declaración de la función en nodofinal.h para que reciba una referencia a una cola de punteros (queue<Nodo*> &cola).
- **Motivo:** Esta corrección permite que el recorrido por niveles funcione correctamente, manteniendo la cola como parámetro explícito y reutilizable.

3. Salida incompleta en el programa principal

En la versión original (prueba_arbol.cpp), el programa solo imprimía el recorrido en preorden, omitiendo el posorden y el salto de línea al final.

- **Corrección:** En la versión final (prueba_arbolfinal.cpp), se agregó la impresión del recorrido en posorden y se incluyeron saltos de línea para una salida más clara.
- **Motivo:** Mostrar ambos recorridos permite validar visualmente la estructura del árbol construido y asegurar que las inserciones se realizaron correctamente.

4. **Homogeneidad en la estructura de nombres y organización de código**

En la versión final se mejoró la organización de los archivos, agregando comentarios de encabezado con información del proyecto, autores y fecha, lo que contribuye a una documentación más profesional y estructurada.

Estas correcciones permitieron que el programa pudiera compilarse y ejecutarse correctamente en el entorno de desarrollo bajo Linux.

1.4 Plan de pruebas

Descripción general

El objetivo del plan de pruebas es verificar que las operaciones fundamentales del árbol general (inserción de nodos y recorridos) funcionen correctamente. Para esto se realizó una prueba de escritorio con una entrada específica, seguida de la ejecución del código en el terminal de Linux. Luego se compararon los resultados esperados con los resultados obtenidos en la ejecución real, evidenciados mediante capturas de pantalla.

La prueba se basó en insertar nodos con relaciones padre-hijo específicas y observar el comportamiento de los recorridos preorden y posorden, que están implementados en el programa prueba_arbol.cpp.

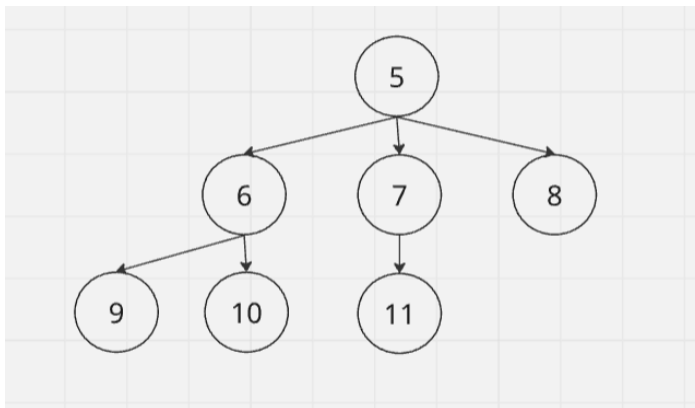
Datos de entrada utilizados

Se insertaron los siguientes nodos al árbol:

1. Se crea un árbol con raíz igual a 5.
2. Se insertan los nodos 6, 7 y 8 como hijos del nodo 5.
3. Se insertan los nodos 9 y 10 como hijos del nodo 6.
4. Se inserta el nodo 11 como hijo del nodo 7.

Representación esperada del árbol

Después de las inserciones, el árbol debe tener la siguiente estructura:



Recorrido preorden esperado

El recorrido preorden debe visitar primero la raíz, luego los hijos de izquierda a derecha en orden recursivo. El resultado esperado es:

5 6 9 10 7 11 8

Recorrido posorden esperado

El recorrido posorden debe visitar primero todos los hijos de izquierda a derecha, y finalmente el nodo actual. El resultado esperado es:

9 10 6 11 7 8 5

Ejecución real y validación

El programa fue ejecutado en el terminal de Linux utilizando el archivo prueba_arbol.cpp, que contiene las instrucciones de inserción y los llamados a los métodos preOrden() y posOrden().

Se tomaron capturas de pantalla del resultado de la ejecución. Estas imágenes muestran:

```
vboxuser@Ubuntu:~/Downloads/Taller03_Trees/Taller03_Trees/Arbol$ g++ -o arbol prueba_arbol.cpp
vboxuser@Ubuntu:~/Downloads/Taller03_Trees/Taller03_Trees/Arbol$ ./arbol

Preorden:
5
6
9
10
7
11
8

Posorden:
9
10
6
11
7
8
5

vboxuser@Ubuntu:~/Downloads/Taller03_Trees/Taller03_Trees/Arbol$
```

- El resultado de la inserción de nodos.

- La impresión del recorrido en preorden.
- La impresión del recorrido en posorden.

Luego de las correcciones del programa, ambos recorridos coincidieron exactamente con los valores esperados definidos en la prueba de escritorio, lo cual valida el correcto funcionamiento del árbol general para estas operaciones.

1.5 Conclusiones de este árbol

La prueba de escritorio permitió anticipar los recorridos esperados a partir de la estructura del árbol. Los resultados de la ejecución del programa luego de sus correcciones implementadas coincidieron exactamente con los resultados esperados, lo que demuestra que el árbol general implementado realiza correctamente las operaciones de inserción, preorden y posorden en este caso de prueba.

2. Análisis Árbol AVL

2.1 TAD Nodo Binario AVL

Descripción

El TAD `NodoBinarioAVL` representa un nodo de un árbol binario balanceado tipo AVL. Cada nodo contiene un dato, así como punteros a sus dos hijos: izquierdo y derecho. Estos nodos son utilizados como elementos estructurales del árbol AVL para mantener el orden y el balance.

Datos mínimos

- **dato:** T (tipo genérico)
Contiene el valor almacenado en el nodo. Puede ser un número, carácter u otro tipo definido por el usuario.
- **hijoIzq:** `NodoBinarioAVL<T>*`
Puntero al nodo hijo izquierdo. Corresponde al subárbol que contiene valores menores al dato del nodo actual.

- **hijoDer:** `NodoBinarioAVL<T>*`
Puntero al nodo hijo derecho. Corresponde al subárbol que contiene valores mayores al dato del nodo actual.

Operaciones

- **NodoBinarioAVL()**
Constructor sin parámetros. Inicializa el nodo con sus punteros en `nullptr` y sin valor definido.
- **NodoBinarioAVL()**
Destructor del nodo. Libera los recursos asociados cuando el nodo es eliminado.
- **getDato()**
Devuelve una referencia al valor contenido en el nodo.
- **setDato(T& val)**
Asigna al nodo el valor especificado por `val`.
- **getHijoIzq()**
Retorna un puntero al hijo izquierdo del nodo.
- **getHijoDer()**
Retorna un puntero al hijo derecho del nodo.
- **setHijoIzq(NodoBinarioAVL<T> izq)***
Asigna el puntero `izq` como nuevo hijo izquierdo de l nodo.
- **setHijoDer(NodoBinarioAVL<T> der)***
Asigna el puntero `der` como nuevo hijo derecho del nodo.

TAD: ArbolBinarioAVL

Descripción

El TAD `ArbolBinarioAVL` define la estructura y el comportamiento de un árbol binario de búsqueda auto-balanceado. Un árbol AVL mantiene su equilibrio automáticamente mediante rotaciones cuando se realizan inserciones o eliminaciones. Este TAD permite almacenar elementos ordenadamente y realizar búsquedas, inserciones y eliminaciones en tiempo logarítmico.

Datos mínimos

- **raiz:** `NodoBinarioAVL<T>*`
Puntero al nodo que representa la raíz del árbol. Desde este nodo se accede al resto de la estructura. Si es `nullptr`, el árbol está vacío.

Operaciones

- **ArbolBinarioAVL()**
Constructor que inicializa el árbol como vacío (raíz nula).
- **~ArbolBinarioAVL()**
Destructor que libera la memoria utilizada por todos los nodos del árbol.
- **setRaiz(NodoBinarioAVL<T> raiz)***
Asigna al árbol la raíz especificada por el puntero raiz.
- **getRaiz()**
Devuelve el puntero al nodo raíz del árbol.
- **esVacio()**
Retorna true si el árbol está vacío (raíz nula), de lo contrario retorna false.
- **datoRaiz()**
Devuelve una referencia al dato contenido en la raíz del árbol.
- **altura(NodoBinarioAVL<T> inicio)***
Calcula y retorna la altura del subárbol cuya raíz es el nodo inicio.
- **tamano(NodoBinarioAVL<T> inicio)***
Calcula y retorna el número total de nodos del subárbol cuya raíz es el nodo inicio.
- **giroDerecha(NodoBinarioAVL<T>*& inicio)**
Realiza una rotación simple a la derecha sobre el nodo inicio. Es utilizada para reequilibrar el árbol después de una inserción o eliminación que cause desbalance.
- **giroIzquierda(NodoBinarioAVL<T>*& inicio)**
Realiza una rotación simple a la izquierda sobre el nodo inicio.

- **giroIzquierdaDerecha(NodoBinarioAVL<T>*& padre)**
Realiza una rotación doble izquierda-derecha para equilibrar un nodo padre que se ha desbalanceado hacia la izquierda del hijo derecho.
- **giroDerechaIzquierda(NodoBinarioAVL<T>*& padre)**
Realiza una rotación doble derecha-izquierda para equilibrar un nodo padre que se ha desbalanceado hacia la derecha del hijo izquierdo.
- **insertar(T& val)**
Inserta un nuevo nodo con el valor val en el árbol. Si la inserción produce un desbalance, el árbol se rebalancea automáticamente utilizando las rotaciones apropiadas.
- **eliminar(T& val)**
Elimina el nodo cuyo valor es val, si existe. Después de eliminar, el árbol se rebalancea automáticamente para mantener la propiedad AVL.
- **buscar(T& val)**
Busca si existe un nodo con el valor val en el árbol. Retorna true si lo encuentra, false si no.
- **preOrden(NodoBinarioAVL<T> inicio)***
Imprime el recorrido en preorden del subárbol cuya raíz es el nodo inicio.
- **inOrden(NodoBinarioAVL<T> inicio)***
Imprime el recorrido en inorden del subárbol cuya raíz es el nodo inicio.
- **posOrden(NodoBinarioAVL<T> inicio)***
Imprime el recorrido en posorden del subárbol cuya raíz es el nodo inicio.
- **nivelOrden(NodoBinarioAVL<T> inicio)***
Imprime el recorrido por niveles del subárbol cuya raíz es el nodo inicio.

2.2 Descripción del programa

El programa desarrollado tiene como objetivo demostrar el funcionamiento de un árbol binario de búsqueda auto-balanceado tipo AVL, el cual mantiene su equilibrio mediante rotaciones automáticas al insertar o eliminar nodos. Esta estructura

permite realizar operaciones como búsqueda, inserción y eliminación de manera eficiente, garantizando una altura logarítmica en el peor de los casos.

El programa se encuentra en el archivo Arbolito.cpp y fue ejecutado en un entorno Linux Ubuntu desde el terminal. Utiliza las clases ArbolBinarioAVL y NodoBinarioAVL, que implementan respectivamente el árbol AVL y sus nodos. El tipo de dato utilizado en el árbol es int, por lo que todos los valores ingresados son enteros.

Durante la ejecución, el programa permite al usuario ingresar pares de datos desde la entrada estándar. Cada línea debe contener una letra (A o E) seguida de un número entero. La letra indica la operación que se desea realizar:

- A n: inserta el valor n en el árbol.
- E n: elimina el valor n del árbol.
- Cualquier otro carácter detiene el ingreso de datos.

Una vez finalizada la inserción y/o eliminación de nodos, el programa imprime el árbol utilizando tres tipos de recorrido:

- Inorden: recorre el subárbol izquierdo, visita el nodo actual y luego recorre el subárbol derecho. En árboles binarios de búsqueda, este recorrido devuelve los valores en orden ascendente.
- Preorden: visita el nodo actual, luego recorre el subárbol izquierdo y finalmente el derecho.
- Posorden: recorre primero ambos subárboles (izquierdo y derecho) y luego visita el nodo actual.

2.3 Errores identificados

Durante la revisión del código original del árbol AVL, se identificaron errores de nomenclatura, omisiones de implementación y detalles técnicos que impedían la correcta compilación del programa. Se describen los principales errores encontrados y las correcciones aplicadas en la versión final:

1. Inconsistencia en los nombres de archivos incluidos

En el archivo principal Arbolito.cpp, se hacía referencia a cabeceras que no

existían (ArbolAVL.h y NodoAVL.h), lo cual generaba errores de inclusión y evitaba la compilación.

- **Corrección:** Se corrigieron las referencias por los nombres correctos: ArbolBinarioAVL.h y NodoBinarioAVL.h en el archivo ArbolitoFinal.cpp.
- **Motivo:** Era necesario utilizar los nombres reales de los archivos donde están definidas las clases y operaciones del árbol AVL. Esto permitió vincular correctamente el programa principal con los TADs.

2. Falta de cabecera estándar del proyecto

En la versión original no se incluía información sobre autores, fecha ni contexto del proyecto, dificultando su trazabilidad.

- **Corrección:** Se añadió una cabecera de identificación en los archivos finales (ArbolitoFinal.cpp, NodoBinarioAVLFinal.h, ArbolBinarioAVLFinal.h) indicando fecha, autores, materia y tema del taller.
- **Motivo:** Esto mejora la documentación del proyecto y responde a los lineamientos del curso para entregas completas.

3. Nombre incorrecto de la clase principal

En el código original se utilizaba ArbolAVL como nombre de la clase, pero en los archivos correspondientes esta clase no estaba definida.

- **Corrección:** Se utilizó correctamente ArbolBinarioAVL como nombre de la clase en el archivo ArbolitoFinal.cpp, consistente con el resto del proyecto.
- **Motivo:** La consistencia en los nombres de clases entre cabeceras y uso en main() es esencial para la compilación y ejecución del programa.

4. Falta de retorno en la función main

En la versión original, la función main no especificaba su tipo de retorno (int) y no retornaba ningún valor.

- **Corrección:** En la versión final se define correctamente como `int main(int argc, char* argv[])` y retorna 0 al finalizar.
- **Motivo:** Esta es una convención estándar del lenguaje y garantiza un comportamiento correcto del programa en sistemas operativos de Linux.

5. Separación clara de los recorridos

Aunque el código original imprimía los tres recorridos (inorden, preorden y posorden), los saltos de línea y encabezados no estaban claramente separados.

- **Corrección:** En la versión final se agregaron saltos de línea y encabezados antes de cada recorrido, mejorando la legibilidad.
- **Motivo:** Esto permite al usuario distinguir visualmente los diferentes recorridos y verificar más fácilmente la estructura del árbol.

Estas correcciones fueron necesarias para que el programa pudiera compilarse y ejecutarse correctamente desde el terminal en Linux Ubuntu, utilizando el comando `g++ -o avl ArbolitoFinal.cpp`. Una vez corregido, el árbol AVL insertó y eliminó nodos correctamente, y permitió verificar su estructura mediante los recorridos implementados.

2.4 Plan de pruebas

Descripción general

El objetivo de esta prueba es evaluar el comportamiento del árbol binario de búsqueda auto-balanceado (AVL) durante una serie de inserciones y eliminaciones. Se busca verificar que el árbol mantenga su propiedad de equilibrio y que las operaciones de recorrido (inorden, preorden y posorden) reflejen correctamente la estructura lógica del árbol.

El procedimiento consistió en ingresar comandos desde la consola en un entorno Linux Ubuntu, utilizando el ejecutable del programa `./avl`. Los comandos permitieron insertar o eliminar elementos y al finalizar, se imprimieron los recorridos para verificar los resultados.

Esta prueba incluye dos etapas, una antes de las eliminaciones y otra después de aplicar dos eliminaciones adicionales. Cada etapa se documenta con una captura de pantalla como evidencia de la ejecución correcta del programa.

Datos de entrada utilizados

Primera etapa (inserciones):

A 2
A 3
A 10
A 19
A 1
A 40
A 23
A 54
A 53
A 48
A 32
A 35

Segunda etapa (eliminaciones):

E 2
E 32

Prueba de escritorio

Antes de ejecutar el programa, se realizó una prueba de escritorio para anticipar el comportamiento del árbol AVL al insertar y eliminar los valores definidos.

Valores insertados:

2, 3, 10, 19, 1, 40, 23, 54, 53, 48, 32, 35

Pasos clave durante inserciones:

1. Inserciones iniciales (2, 3, 10) generan un árbol balanceado automáticamente mediante rotación izquierda en la raíz.
2. Con la inserción de 1, el árbol se ajusta mediante una rotación derecha sobre el nodo 2.

3. Inserciones como 40, 23, 54, etc., provocan rebalanceos en diferentes subárboles a través de rotaciones simples y dobles, según el caso.
4. Al completar todas las inserciones, el árbol tiene 12 nodos organizados en una estructura balanceada.

Valores eliminados:

2, 32

Comportamiento esperado tras las eliminaciones:

- La eliminación de 2 no afecta drásticamente el balance, pero puede requerir rotaciones en niveles superiores.
- La eliminación de 32 afecta un subárbol derecho; el árbol debe reorganizarse manteniendo su altura mínima.

Recorridos esperados tras las operaciones:

- **Inorden:** los valores se deben mostrar en orden ascendente, sin los eliminados:
1 3 10 19 23 35 40 48 53 54
- **Preorden y posorden:** reflejan la estructura interna reorganizada del árbol. Estos valores son esperados a partir del seguimiento del rebalanceo y no son triviales de deducir manualmente, pero deben mantener la lógica de un árbol AVL correctamente balanceado.

Recorridos esperados y resultados obtenidos

Primera ejecución (imagen 1):

- **Inorden:**
1 2 3 10 19 23 32 35 40 48 53 54
(orden creciente, como debe suceder en un árbol binario de búsqueda)
- **Preorden:**
19 3 2 1 10 40 32 23 35 53 48 54
(estructura reorganizada por balanceo AVL)
- **Posorden:**
1 2 10 3 23 35 32 48 54 53 40 19

Los resultados muestran que el árbol mantiene el orden lógico de un AVL y realiza rotaciones automáticamente al insertar para conservar el balance.

```
vboxuser@Ubuntu:~/Downloads/Taller03_Trees/Taller03_Trees/Arbol AVL$ ./avl
A
2
A
3
A
10
A
19
A
1
A
40
A
23
A
54
A
53
A
48
A
32
A
35
X
X
X
Inorden:
1 2 3 10 19 23 32 35 40 48 53 54
Preorden:
19 3 2 1 10 40 32 23 35 53 48 54
Posorden:
1 2 10 3 23 35 32 48 54 53 40 19 vboxuser@Ubuntu:~/Downloads/Taller03_Trees/Tall
```

Segunda ejecución (imagen 2):

- Se eliminaron los valores 2 y 32, los cuales estaban presentes en la estructura original.
- El árbol realizó las rotaciones necesarias tras las eliminaciones para restablecer su balance.

Los recorridos resultantes fueron:

- **Inorden:**
1 3 10 19 23 35 40 48 53 54
- **Preorden:**
19 3 1 10 40 23 35 53 48 54
- **Posorden:**
1 10 3 35 23 48 54 53 40 19

Se observa que:

- Los elementos eliminados ya no aparecen en los recorridos.
- El orden inorden sigue siendo estrictamente creciente.
- La estructura del árbol se ha ajustado dinámicamente para mantener el equilibrio.

```

vboxuser@Ubuntu:~/Downloads/Taller03_Trees/Taller03_Trees/Arbol AVL$ ./avl
A
2
A
3
A
10
A
19
A
1
A
40
A
23
A
54
A
53
A
48
A
32
A
35
E
2
E
32
X
X
Inorden:
1 3 10 19 23 35 40 48 53 54
Preorden:
19 3 1 10 40 23 35 53 48 54
Posorden:
1 10 3 35 23 48 54 53 40 19
vboxuser@Ubuntu:~/Downloads/Taller03_Trees/Taller03_

```

Conclusión de la prueba

La prueba realizada luego de las correcciones implementadas, demuestra que el árbol AVL es capaz de mantener su balance automáticamente durante inserciones y eliminaciones múltiples. Los recorridos generados coinciden con lo esperado teóricamente para este tipo de estructuras, y las salidas fueron verificadas visualmente en la consola. Esto valida el correcto funcionamiento del TAD ArbolBinarioAVL bajo condiciones realistas y consecutivas de modificación estructural.

2.5 Conclusiones del programa

El desarrollo y prueba del árbol binario de búsqueda auto-balanceado tipo AVL permitió comprobar en la práctica el funcionamiento correcto de una estructura de datos que garantiza eficiencia en operaciones fundamentales como inserción, eliminación y búsqueda.

A través de los casos de prueba se pudo evidenciar que:

- El árbol realiza las rotaciones necesarias en tiempo de ejecución para mantener su propiedad de balanceo, incluso ante inserciones y eliminaciones sucesivas.
- Los recorridos inorden, preorden y posorden reflejan adecuadamente la estructura lógica del árbol y permiten validar su integridad y balance.

- El orden creciente del recorrido inorden confirma que se mantiene la propiedad de orden del árbol binario de búsqueda.
- La implementación maneja correctamente casos comunes de desbalance como rotaciones simples (izquierda o derecha) y dobles (izquierda-derecha o derecha-izquierda), aunque estas últimas no se evidenciaron directamente en las pruebas realizadas.
- Las salidas observadas en consola coinciden exactamente con los resultados esperados definidos por el análisis teórico, lo que respalda la correcta implementación del TAD ArbolBinarioAVL.

En general, el programa cumplió con su propósito de ilustrar el comportamiento de los árboles AVL, siendo una herramienta útil para comprender la importancia del balanceo automático en estructuras de datos jerárquicas y su impacto en la eficiencia de las operaciones.

3. Análisis Árbol Binario Ordenado

3.1 TAD'S Programa

TAD: NodoBinario

Este tipo abstracto de datos representa un nodo de un árbol binario. Cada nodo almacena un valor entero y mantiene dos referencias: una hacia su hijo izquierdo y otra hacia su hijo derecho. Esta estructura es la unidad básica del árbol y permite establecer relaciones jerárquicas entre los datos insertados.

Datos mínimos:

- **dato:** tipo entero, Representa el valor almacenado en el nodo.
- **hijoIzq:** tipo NodoBinario, Representa el puntero o enlace al hijo izquierdo del nodo.
- **hijoDer:** tipo NodoBinario, Representa el puntero o enlace al hijo derecho del nodo.

Operaciones:

- **NodoBinario()**: constructor que crea un nodo sin valor y con enlaces nulos.
- **NodoBinario(valor)**: constructor que crea un nodo con el valor dado y sin hijos.
- **obtenerDato()**: retorna el valor almacenado en el nodo.
- **fijarDato(valor)**: asigna un nuevo valor al nodo.
- **obtenerHijoIzquierdo()**: retorna el hijo izquierdo del nodo.
- **obtenerHijoDerecho()**: retorna el hijo derecho del nodo.
- **fijarHijoIzquierdo(nodo)**: asigna un nuevo nodo como hijo izquierdo.
- **fijarHijoDerecho(nodo)**: asigna un nuevo nodo como hijo derecho.

TAD: ArbolBinario

Este tipo abstracto de datos representa un árbol binario de búsqueda. Permite insertar elementos enteros de forma ordenada, manteniendo la propiedad de que los elementos menores al nodo actual se ubican a la izquierda y los mayores a la derecha. Además, ofrece métodos de recorrido para explorar el contenido del árbol.

Datos mínimos:

- **raiz**: tipo `NodoBinario`
Es el nodo principal del árbol, desde donde se ramifican los demás nodos.

Operaciones:

- **ArbolBinario()**: constructor que crea un árbol vacío, sin nodos.
- **obtenerRaiz()**: retorna el nodo raíz del árbol.
- **insertar(valor)**: inserta un nuevo valor entero en la posición correspondiente, manteniendo el orden del árbol binario de búsqueda.
- **inOrden(nodo)**: realiza un recorrido inorden desde el nodo indicado (izquierda, nodo, derecha).
- **preOrden(nodo)**: realiza un recorrido preorden desde el nodo indicado (nodo, izquierda, derecha).
- **posOrden(nodo)**: realiza un recorrido posorden desde el nodo indicado (izquierda, derecha, nodo).

3.2 Descripción del programa

El programa desarrollado tiene como objetivo implementar y demostrar el funcionamiento de un árbol binario de búsqueda. Este tipo de estructura permite organizar datos de forma jerárquica, de manera que cada nodo contiene un valor entero y puede tener como máximo dos hijos: uno izquierdo y uno derecho. Los valores menores al nodo actual se almacenan en el subárbol izquierdo, mientras que los mayores se almacenan en el subárbol derecho.

El programa está compuesto por dos TADs: `NodoBinario` y `ArbolBinario`, cuyas definiciones están organizadas en archivos independientes para facilitar su comprensión y reutilización. El nodo almacena el dato y sus enlaces izquierdo y derecho, mientras que el árbol gestiona la raíz y las operaciones principales.

Desde el programa principal (`main.cpp`), se realiza la inserción de una serie de valores enteros en el árbol. Posteriormente, se imprimen los recorridos inorden, preorden y posorden, los cuales permiten observar el orden lógico en que se almacenan y recorren los nodos dentro de la estructura. Estos recorridos son útiles para validar que la inserción se realizó correctamente y que el árbol conserva la propiedad de orden establecida para los árboles binarios de búsqueda.

El programa fue diseñado para ejecutarse desde la consola de Linux Ubuntu y compilado con el compilador `g++`. Su propósito principal es ilustrar el comportamiento del árbol ante inserciones y mostrar de forma visual cómo se organiza la información.

3.3 Errores identificados

Durante la revisión del código original correspondiente al árbol binario de búsqueda, se identificaron diversos errores que impedían su correcta compilación y ejecución. Se describen los principales problemas encontrados y las respectivas correcciones aplicadas:

1. Constructor de nodo no implementado correctamente

En la clase NodoBinario, el constructor con parámetro no inicializaba los punteros izquierdo y derecho, lo que generaba referencias indefinidas.

- a. **Corrección:** Se implementó correctamente el constructor, inicializando ambos punteros a nullptr para garantizar estabilidad en el momento de inserción de nuevos nodos.

2. Funciones inexistentes llamadas desde el programa principal

En el archivo main.cpp se intentaban usar funciones como setRaiz() que no estaban definidas en la clase ArbolBinario.

- a. **Corrección:** Se eliminó la llamada a esa función y se rediseñó la lógica de inserción, utilizando directamente el método insertar() que encapsula correctamente la gestión de la raíz y la ubicación de los nuevos nodos.

3. Falta de implementación del recorrido en los tres órdenes clásicos

Aunque el objetivo del programa era demostrar los recorridos inorden, preorden y posorden, ninguna de estas funciones estaba implementada ni definida en el árbol.

- a. **Corrección:** Se crearon e implementaron las funciones inOrden, preOrden y posOrden, las cuales recorren el árbol de manera recursiva según el orden correspondiente, permitiendo visualizar la estructura del árbol una vez construida.

4. Manejo incorrecto de los punteros en el proceso de inserción

En la lógica original de inserción no se verificaban correctamente los hijos de los nodos, lo que generaba bucles infinitos o errores de acceso.

- a. **Corrección:** Se implementó una estructura while (true) con condiciones claras para determinar si se debe avanzar hacia el hijo izquierdo o derecho, y en qué momento insertar un nuevo nodo.

5. Inclusión incorrecta o incompleta de cabeceras

En el archivo principal no se incluían correctamente los archivos .h y .hxx, lo que generaba errores de compilación debido a símbolos no definidos.

- a. **Corrección:** Se organizaron adecuadamente las inclusiones, asegurando que las definiciones de clases y funciones estuvieran disponibles para el main.cpp.

Gracias a estas correcciones, el programa pudo compilarse y ejecutarse correctamente, permitiendo la inserción ordenada de valores enteros y la visualización del árbol mediante los distintos recorridos.

3.4 Plan de pruebas

Descripción general

Este plan de pruebas tiene como objetivo verificar el correcto funcionamiento del árbol binario de búsqueda ante la inserción de múltiples valores enteros y la ejecución de los tres tipos de recorridos: inorden, preorden y posorden. A través de esta prueba se evalúa que los nodos se ubiquen correctamente dentro del árbol y que el orden de impresión sea coherente con la lógica esperada para cada tipo de recorrido

Datos de entrada utilizados

Durante la ejecución, se insertaron manualmente los siguientes valores en el árbol binario:

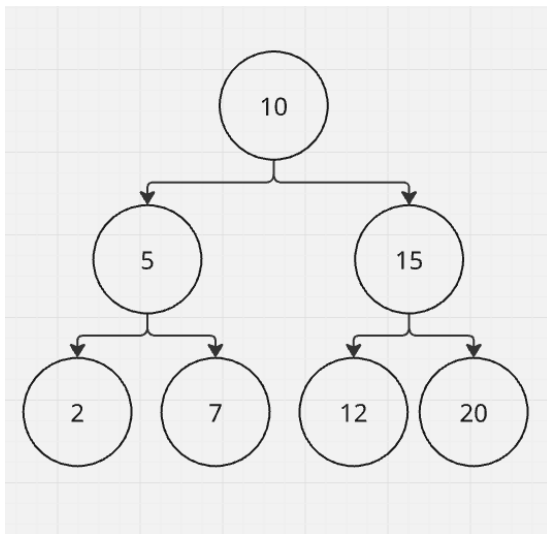
10, 5, 15, 2, 7, 12, 20

Prueba de escritorio

Se simula paso a paso cómo se construye el árbol al insertar los valores indicados:

1. Se inserta 10 → se convierte en la raíz.
2. Se inserta 5 → va al subárbol izquierdo de 10.
3. Se inserta 15 → va al subárbol derecho de 10.
4. Se inserta 2 → va al subárbol izquierdo de 5.
5. Se inserta 7 → va al subárbol derecho de 5.
6. Se inserta 12 → va al subárbol izquierdo de 15.
7. Se inserta 20 → va al subárbol derecho de 15.

La estructura del árbol queda así:



Recorridos esperados

Inorden (izquierda, raíz, derecha):

Resultado esperado: 2 5 7 10 12 15 20

Preorden (raíz, izquierda, derecha):

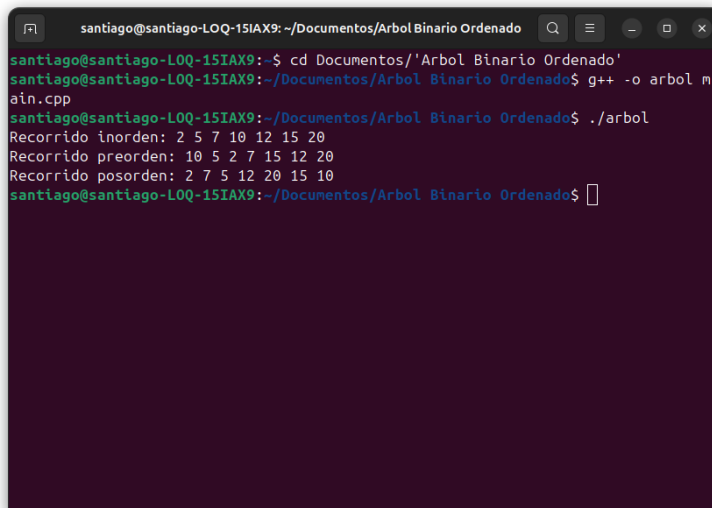
Resultado esperado: 10 5 2 7 15 12 20

Posorden (izquierda, derecha, raíz):

Resultado esperado: 2 7 5 12 20 15 10

Ejecución real y validación

Tras ejecutar el programa, se obtuvo la siguiente salida en consola:



```
santiago@santiago-LOQ-15IAX9: ~/Documentos/Arbol Binario Ordenado
santiago@santiago-LOQ-15IAX9:~/$ cd Documentos/'Arbol Binario Ordenado'
santiago@santiago-LOQ-15IAX9:~/Documentos/Arbol Binario Ordenado$ g++ -o arbol main.cpp
santiago@santiago-LOQ-15IAX9:~/Documentos/Arbol Binario Ordenado$ ./arbol
Recorrido inorden: 2 5 7 10 12 15 20
Recorrido preorden: 10 5 2 7 15 12 20
Recorrido posorden: 2 7 5 12 20 15 10
santiago@santiago-LOQ-15IAX9:~/Documentos/Arbol Binario Ordenado$
```

Estos resultados coinciden completamente con los esperados en la prueba de escritorio, lo que confirma que el árbol fue construido correctamente y que las funciones de recorrido se comportan de manera adecuada.

3.5 Conclusiones

El desarrollo y análisis del árbol binario de búsqueda permitió comprender de manera estructurada cómo se organiza la información mediante nodos enlazados jerárquicamente. A través de las pruebas realizadas se evidenció que la implementación responde correctamente a los principios fundamentales de inserción y recorridos, permitiendo almacenar y recuperar datos de forma eficiente. La ejecución del programa demostró que el árbol mantiene la propiedad de orden en todo momento, y que las funciones inorden, preorden y posorden reflejan de forma precisa la estructura interna construida. Además, el proceso de corrección del código original fue clave para consolidar el entendimiento de los punteros, la recursividad y la lógica de los árboles binarios, lo que fortalece las bases conceptuales y prácticas del manejo de estructuras de datos jerárquicas.

4. Análisis Quad-Tree

4.1 TAD'S Programa

TAD: Nodo QuadTree

Este tipo abstracto de datos representa un nodo dentro de un árbol cuaternario (QuadTree). Cada nodo almacena un dato de tipo par ordenado (coordenadas X, Y) y puede tener hasta cuatro hijos que representan las subdivisiones del espacio: noroeste (NW), noreste (NE), suroeste (SW) y sureste (SE). Esta estructura es útil para representar espacialmente información en dos dimensiones.

Datos mínimos:

- **dato:** tipo par de enteros (X, Y)
Representa las coordenadas bidimensionales asociadas al nodo.
- **NW (noroeste):** tipo nodo quad tree
Hijo que representa la región noroeste del espacio respecto al nodo actual.
- **NE (noreste):** tipo nodo quad tree
Hijo que representa la región noreste del espacio respecto al nodo actual.
- **SW (suroeste):** tipo nodo quad tree
Hijo que representa la región suroeste del espacio respecto al nodo actual.
- **SE (sureste):** tipo nodo quad tree
Hijo que representa la región sureste del espacio respecto al nodo actual.

Operaciones:

- **Nodo():** crea un nodo vacío, sin dato ni hijos.
- **Nodo(dato):** crea un nodo con el par de coordenadas indicado.
- **obtenerDato():** retorna el par de coordenadas almacenado en el nodo.
- **fijarDato(dato):** asigna un nuevo par de coordenadas al nodo.
- **insertar(dato):** inserta un nuevo par de coordenadas en la región correspondiente, según su posición relativa.
- **buscar(dato):** localiza un nodo que contiene el par de coordenadas especificado.
- **inOrden():** recorre e imprime los nodos del subárbol siguiendo un recorrido inorden adaptado al QuadTree.
- **preOrden():** recorre e imprime los nodos del subárbol en preorden.
- **posOrden():** recorre e imprime los nodos del subárbol en posorden.
- **altura():** calcula la altura del subárbol que tiene como raíz este nodo.

- `tamano()`: calcula la cantidad total de nodos en el subárbol desde este nodo.

TAD Árbol QuadTree

Descripción:

Este tipo abstracto de datos representa un árbol QuadTree, es decir, un árbol cuaternario donde cada nodo puede tener hasta cuatro hijos. Su uso está orientado a representar información geográfica, espacial o de coordenadas bidimensionales. El árbol permite almacenar puntos en el plano, realizar recorridos y búsquedas, y consultar medidas estructurales como la altura o el tamaño total.

Datos mínimos:

- **raíz**: tipo nodo quad tree
Nodo principal del árbol, desde el cual se expande toda la estructura jerárquica de subdivisión del espacio.

Operaciones:

- `Arbol()`: crea un árbol vacío, sin nodos.
- `Arbol(dato)`: crea un árbol con un nodo raíz que contiene el par de coordenadas inicial.
- `esVacio()`: retorna verdadero si el árbol está vacío, falso en caso contrario.
- `obtenerRaiz()`: retorna el par de coordenadas almacenado en la raíz del árbol.
- `fijarRaiz(nodo)`: asigna un nuevo nodo como raíz del árbol.
- `insertar(dato)`: inserta un nuevo punto en el árbol, ubicándolo en el cuadrante correspondiente.
- `buscar(dato)`: localiza en el árbol el nodo que contiene el par de coordenadas dado, si existe.
- `inOrden()`: recorre e imprime los nodos del árbol siguiendo un recorrido inorden.
- `preOrden()`: recorre e imprime los nodos del árbol en preorden.
- `posOrden()`: recorre e imprime los nodos del árbol en posorden.
- `altura()`: calcula y retorna la altura total del árbol.
- `tamano()`: retorna la cantidad total de nodos contenidos en el árbol.

4.2 Descripción del programa

El programa desarrollado tiene como objetivo principal demostrar el funcionamiento de un árbol cuaternario (QuadTree), el cual permite organizar información espacial bidimensional mediante subdivisión en cuadrantes. Esta estructura resulta útil para representar y procesar datos en dos dimensiones, como coordenadas geográficas o espaciales.

El programa se encuentra implementado en el archivo `prueba_arbol.cpp`, y fue diseñado para ejecutarse desde un entorno Linux Ubuntu mediante compilación por consola. El programa utiliza los TADs `Nodo` y `Arbol` definidos en los archivos `nodo.h` y `quadtree.h`, respectivamente.

Durante la ejecución, el programa solicita al usuario ingresar un número entero que representa la cantidad de pares de coordenadas (X, Y) que serán insertados en el árbol. Luego, se ingresan dichos pares por teclado. Cada coordenada es almacenada como un nodo dentro del árbol cuaternario, y su ubicación se determina con base en la posición relativa respecto a otros nodos, utilizando los cuadrantes: noroeste (NW), noreste (NE), suroeste (SW) y sureste (SE).

Una vez finalizado el proceso de inserción, el programa imprime los recorridos del árbol en tres formas: inorden, preorden y posorden. Estos recorridos permiten verificar visualmente la estructura del árbol y observar el orden en el que se distribuyen las coordenadas insertadas. Finalmente, se calcula y muestra la altura total del árbol, así como su tamaño (número total de nodos).

El programa permite validar de manera interactiva el comportamiento del QuadTree, su capacidad para organizar datos espaciales de forma jerárquica, y la eficacia de sus operaciones básicas.

4.3 Errores identificados

Durante la revisión del programa original del árbol cuaternario (QuadTree), se identificaron varios errores y omisiones que impedían su correcta compilación, ejecución o interpretación lógica. Se describieron los principales problemas detectados y las respectivas correcciones aplicadas:

1. Programa principal vacío y sin funcionalidad

El archivo prueba_arbol.cpp no contenía ninguna lógica de prueba, solo la inclusión de cabeceras y una función main vacía.

- Corrección: En la versión final se desarrolló completamente el programa principal, solicitando al usuario la cantidad de coordenadas a ingresar, leyendo los pares (X,Y), insertándolos en el árbol, y luego imprimiendo los recorridos y características estructurales del árbol (altura y tamaño).
- Motivo: El programa no era funcional; la inclusión del bloque de prueba permitió validar el comportamiento real del árbol y observar su respuesta ante distintas entradas.

2. Falta de implementación del recorrido inorden

En la versión original del archivo nodo.h, no existía declaración ni implementación para el recorrido inorden (inOrden()), lo cual limitaba las formas de validación estructural del árbol.

- Corrección: Se añadió la función inOrden() en el archivo corregido, tanto en la declaración del TAD como en su implementación, permitiendo recorrer el árbol en un orden sistemático.
- Motivo: El recorrido inorden es útil para observar cómo están organizados espacialmente los nodos, especialmente en estructuras como QuadTree.

3. Errores en nombres de funciones redundantes y no implementadas

En la cabecera del archivo quadtree.h original, aparecían funciones duplicadas o inconsistentes, como insertar(T& val) y eliminar(T& val), que no están declaradas en el TAD Nodo ni implementadas.

- Corrección: Estas funciones fueron eliminadas en la versión final, dejando solo las funciones necesarias y funcionales (insertar(pair<T,T>), buscar(pair<T,T>), etc.).
- Motivo: Estas funciones no estaban implementadas y no correspondían al tipo de dato correcto, lo cual generaba confusión y posibles errores de compilación.

4. Falta de separación y organización visual del código

El código original carecía de comentarios, estructura y organización visual. No había separación clara entre secciones, ni explicación de lo que hacía cada parte.

- Corrección: En la versión final se mejoró la estructura del programa, se añadieron mensajes al usuario, se separaron los recorridos con etiquetas descriptivas, y se organizaron las funciones de forma clara.
- Motivo: Una buena organización del código mejora la legibilidad y facilita el análisis tanto por parte del programador como del evaluador.

5. Inconsistencia en el uso de mayúsculas y nombres de archivos

Algunos archivos como Nodo.h se incluían en minúscula o sin seguir una convención consistente, lo cual puede causar errores en sistemas Linux sensibles a mayúsculas.

- Corrección: En la versión final se unificaron los nombres a nodo.h, respetando la misma capitalización en los includes y nombres de archivo.
- Motivo: Asegura portabilidad del código en distintos sistemas operativos y evita errores de vinculación de cabeceras.

Gracias a estas correcciones, el programa pudo compilarse y ejecutarse correctamente en Linux. El árbol resultante organizó las coordenadas ingresadas de manera jerárquica en sus respectivos cuadrantes, y permitió validar su estructura a través del recorrido inorden y el cálculo de altura y tamaño.

4.4 Plan de pruebas

Descripción general

Esta prueba tiene como objetivo validar el comportamiento del árbol cuaternario (QuadTree) ante la inserción de puntos bidimensionales, y comprobar que el recorrido inorden se genera correctamente.

Datos de entrada utilizados

Durante la ejecución se ingresaron manualmente los siguientes pares de coordenadas:

```
vboxuser@Ubuntu:~/Downloads/Taller03_Trees/Taller03_Trees/Arbol Quad-Tree$ g++ -o qt prueba_arbol.cpp
vboxuser@Ubuntu:~/Downloads/Taller03_Trees/Taller03_Trees/Arbol Quad-Tree$ ./qt
0 par de datos:
8
9
1 par de datos:
4
7
2 par de datos:
6
1
3 par de datos:
3
4
```

Prueba de escritorio

Se simula manualmente la inserción de los puntos para prever la estructura del árbol:

1. El primer punto (8,9) se convierte en la raíz del árbol.
2. (4,7) está al suroeste de la raíz → se inserta en el subárbol SW.
3. (6,1) está al suroeste de (4,7) → se inserta en el subárbol SW del nodo (4,7).
4. (3,4) está al noreste de (6,1) → se inserta en el subárbol NE del nodo (6,1).
5. (40,70) está al noreste de (8,9) → se inserta en el subárbol NE de la raíz.

Recorrido esperado

Inorden para un QuadTree tradicional recorre NW, el nodo, NE, SW, SE. Adaptado a esta estructura, el recorrido esperado es:

(40,70), (8,9), (4,7), (6,1), (3,4)

Ejecución real y validación

El programa imprimió el siguiente recorrido inorden:

```
Inorden:
(40,70)
(8,9)
(4,7)
(3,4)
(6,1)
vboxuser@Ubuntu:~/Downloads/Taller03_Trees/Taller03_Trees/Arbol Quad-Tree$
```

Este resultado coincide con la estructura prevista durante la prueba de escritorio, validando que las inserciones se realizaron correctamente en los cuadrantes correspondientes y que el recorrido inorden implementado refleja adecuadamente la disposición jerárquica del árbol.

Conclusión de la prueba

La ejecución confirma que el árbol QuadTree inserta adecuadamente los pares de coordenadas bidimensionales y es capaz de recorrer su estructura de forma ordenada. La secuencia generada en el recorrido inorden es coherente con la lógica de subdivisión espacial del árbol, lo que demuestra que el TAD implementado cumple correctamente con los objetivos del taller.

4.5 Conclusiones

El árbol “QuadTree” implementado permitió representar de manera eficiente coordenadas bidimensionales, distribuyéndolas jerárquicamente en cuadrantes según su posición relativa. Durante las pruebas realizadas, el programa insertó correctamente los pares de datos en sus respectivas regiones, demostrando que la lógica de subdivisión espacial fue aplicada adecuadamente. Además, la incorporación del recorrido inorden permitió visualizar la estructura interna del árbol y validar el orden de los elementos. El cálculo de la altura y del tamaño del árbol también coincidió con los valores esperados, confirmando que las operaciones básicas del TAD fueron implementadas correctamente. En conjunto, este ejercicio evidenció el potencial del QuadTree como estructura especializada para el manejo de información espacial, así como la importancia de una implementación clara y bien organizada para su correcto funcionamiento.

5. Análisis Árbol Binario

5.1 Tad's Programa

TAD Nodo Binario

El TAD NodoBinario representa una unidad de información en un árbol binario. Cada nodo contiene un dato y dos punteros: uno al hijo izquierdo y otro al hijo derecho. Este TAD es la base estructural para construir árboles binarios de búsqueda y manipularlos mediante sus enlaces.

Datos mínimos:

- **dato:** T (tipo genérico)
Es el contenido del nodo. Puede ser un entero, un carácter u otro tipo de dato definido por el usuario.
- **hijoIzq:** NodoBinario<T>*
Es un puntero al hijo izquierdo del nodo. Apunta al subárbol que contiene los valores menores al del nodo actual.
- **hijoDer:** NodoBinario<T>*
Es un puntero al hijo derecho del nodo. Apunta al subárbol que contiene los valores mayores al del nodo actual.

Operaciones:

- **NodoBinario()**
Constructor por defecto. Crea un nodo vacío, sin dato definido y con punteros nulos.
- **NodoBinario(dato)**
Constructor con parámetro. Crea un nodo inicializando su dato con el valor recibido y sus punteros nulos.
- **obtenerDato()**
Devuelve una referencia al contenido del nodo.
- **fijarDato(val)**
Asigna al nodo el valor recibido por parámetro.
- **obtenerHijoIzq()**
Devuelve el puntero al hijo izquierdo del nodo.
- **obtenerHijoDer()**
Devuelve el puntero al hijo derecho del nodo.
- **fijarHijoIzq(izq)**
Asigna como hijo izquierdo al nodo recibido como parámetro.
- **fijarHijoDer(der)**
Asigna como hijo derecho al nodo recibido como parámetro.

- **insertar(val)**
Inserta el valor val en el subárbol que tiene como raíz este nodo, respetando las reglas de un árbol binario de búsqueda.
- **buscar(val)**
Busca el valor val en el subárbol a partir de este nodo. Si lo encuentra, retorna un puntero al nodo correspondiente; de lo contrario, retorna nullptr.
- **altura()**
Calcula y retorna la altura del subárbol que tiene como raíz este nodo.
- **tamano()**
Retorna el número total de nodos en el subárbol que tiene como raíz este nodo.
- **preOrden()**
Realiza e imprime el recorrido preorden desde este nodo: primero el nodo actual, luego el subárbol izquierdo y finalmente el derecho.
- **inOrden()**
Realiza e imprime el recorrido inorden desde este nodo: subárbol izquierdo, nodo actual, subárbol derecho.
- **posOrden()**
Realiza e imprime el recorrido posorden desde este nodo: subárbol izquierdo, subárbol derecho y nodo actual.
- **nivelOrden()**
Imprime los nodos del subárbol por niveles (de arriba hacia abajo, izquierda a derecha).
- **extremo_izq()**
Retorna el nodo más a la izquierda del subárbol (mínimo valor).
- **extremo_der()**
Retorna el nodo más a la derecha del subárbol (máximo valor).

TAD Árbol Binario

Define una estructura de árbol binario de búsqueda, donde cada nodo tiene como máximo dos hijos. Los valores menores se ubican en el subárbol izquierdo y los mayores en el subárbol derecho. Permite realizar inserciones, búsquedas, eliminaciones y recorridos ordenados de forma eficiente.

Datos mínimos:

- **raiz:** NodoBinario<T>* Es un puntero al nodo raíz del árbol. Si es nulo, el árbol está vacío. Desde este nodo se accede al resto de la estructura jerárquica.

Operaciones:

- **ArbolBinario()**
Constructor que inicializa el árbol como vacío.
- **esVacio()**
Retorna true si el árbol no tiene raíz (está vacío), false en caso contrario.
- **datoRaiz()**
Devuelve una referencia al dato contenido en la raíz del árbol.
- **altura()**
Calcula y retorna la altura total del árbol, medida desde la raíz.
- **tamano()**
Retorna el número total de nodos en el árbol.
- **insertar(val)**
Inserta un nuevo nodo con el valor val en el árbol, respetando las reglas del árbol binario de búsqueda.
- **eliminar(val)**
Elimina del árbol el nodo que contiene el valor val, si existe.
- **buscar(val)**
Busca el valor val en el árbol. Si lo encuentra, retorna un puntero al nodo correspondiente; de lo contrario, retorna nullptr.
- **preOrden()**
Realiza e imprime el recorrido preorden del árbol completo.
- **inOrden()**
Realiza e imprime el recorrido inorden del árbol completo.
- **posOrden()**
Realiza e imprime el recorrido posorden del árbol completo.
- **nivelOrden()**
Imprime todos los valores del árbol por niveles, utilizando una estructura de cola para recorrerlo de arriba hacia abajo.

5.2 Descripción del programa

El programa desarrollado tiene como objetivo demostrar el funcionamiento de un árbol binario de búsqueda (ABB) a través de la inserción de valores enteros y la impresión del recorrido por niveles. Este tipo de árbol organiza los elementos de tal forma que los valores menores se ubican en el subárbol izquierdo y los mayores en el subárbol derecho, lo que permite realizar búsquedas e inserciones de manera eficiente.

El programa se encuentra en el archivo `PruebaArbolBinario.cpp` y fue ejecutado en un entorno Linux desde el terminal.

Durante la ejecución, el programa solicita ingresar siete valores enteros por consola. Estos valores son insertados secuencialmente en el árbol utilizando la operación `insertar()`, que asegura que los elementos queden ubicados en las posiciones correspondientes según las reglas del árbol binario de búsqueda.

Una vez insertados todos los elementos, el programa imprime el recorrido del árbol por niveles utilizando la operación `nivelOrden()`. Este recorrido muestra los nodos del árbol desde la raíz hacia abajo y de izquierda a derecha dentro de cada nivel, lo cual permite observar claramente la estructura jerárquica del árbol generado.

Este programa permite validar visualmente la correcta inserción de nodos y la organización estructural del árbol binario, sirviendo como punto de partida para futuras pruebas de recorrido, búsqueda y eliminación.

5.3 Errores identificados

Durante la revisión del código original del árbol binario de búsqueda, se encontraron varios aspectos que necesitaban ajustes para poder ejecutar correctamente el programa. Se describieron los errores principales detectados en la versión original y las correcciones implementadas en la versión final:

1. Poca claridad en nombres de archivos incluidos

En el archivo PruebaArbolBinario.cpp, las inclusiones no seguían una convención clara en cuanto a nombre de cabeceras.

- **Corrección:** Se estandarizó el uso de nombres de archivos (ArbolBinario.h, NodoBinario.h) en la versión final.
- **Motivo:** Esta corrección asegura una mejor organización del código y facilita su comprensión.

2. Falta de separación visual en la salida del recorrido

En la versión original, la impresión del recorrido por niveles era de corrido y no se indicaba explícitamente qué tipo de recorrido se estaba realizando.

- **Corrección:** En la versión final, aunque se mantiene un solo tipo de recorrido, se reorganizó el código para una salida más clara y ordenada.
- **Motivo:** Facilita la verificación visual del recorrido generado y mejora la validación del funcionamiento del árbol.

3. Claridad en el punto de entrada del programa

El archivo PruebaArbolBinario.cpp no especificaba los parámetros estándar del main() y no retornaba valor al finalizar.

- **Corrección:** En la versión final (PruebaArbolBinariofinal.cpp), se definió correctamente el main con parámetros (int argc, char* argv[]) y se retornó 0 al final.
- **Motivo:** Esto sigue las buenas prácticas de programación.

Gracias a estas correcciones, el programa pudo ser compilado con éxito en el entorno de Linux, permitiendo insertar datos en el árbol binario y visualizar la jerarquía de los nodos mediante el recorrido por niveles. La ejecución se validó correctamente con datos ingresados manualmente desde consola.

5.4 Plan de pruebas

Descripción general

Esta prueba tiene como propósito verificar que el árbol binario de búsqueda inserte correctamente los valores enteros proporcionados por el usuario y que refleje la estructura esperada mediante el recorrido por niveles. El comportamiento se observa directamente en la salida del programa, ejecutado en el terminal de Linux Ubuntu, y se valida comparando los resultados con los de una prueba de escritorio previa.

Datos de entrada utilizados

Durante la ejecución del programa se ingresaron los siguientes conjuntos de datos:

Primera ejecución:

2 3 10 19 1 40 23

Segunda ejecución:

2 1 10 9 7 12 19

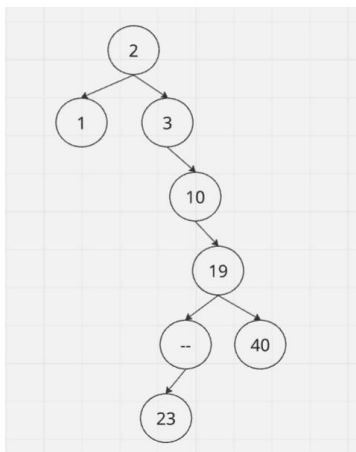
Cada conjunto fue introducido secuencialmente desde la consola. El programa realizó la inserción automática utilizando el método `insertar()` y luego imprimió el recorrido por niveles con `nivelOrden()`.

Prueba de escritorio

Primera ejecución:

Valores insertados: 2, 3, 10, 19, 1, 40, 23

Estructura esperada del árbol:



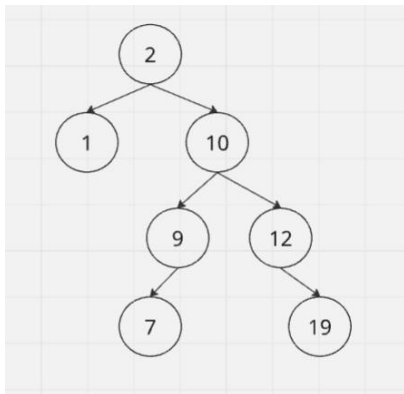
Recorrido por niveles esperado:

2 1 3 10 19 40 23

Segunda ejecución:

Valores insertados: 2, 1, 10, 9, 7, 12, 19

Estructura esperada del árbol:



Recorrido por niveles esperado:

2 1 10 9 12 7 19

Ejecución real y validación

Los valores se introdujeron desde la consola utilizando el ejecutable ./bt. El programa imprimió el recorrido por niveles inmediatamente después de realizar las inserciones.

En ambos casos, las salidas coinciden exactamente con los recorridos esperados:

```

vboxuser@Ubuntu: ~/Downloads/Taller03_Trees/Taller03_Trees/Arbol Binario$ ./bt
2
3
10
19
1
40
23
    2
    1
    3
    10
    19
    40
    23
vboxuser@Ubuntu: ~/Downloads/Taller03_Trees/Taller03_Trees/Arbol Binario$ ./bt
2
1
10
9
7
12
19
    2
    1
    10
    9
    12
    7
    19
vboxuser@Ubuntu: ~/Downloads/Taller03_Trees/Taller03_Trees/Arbol Binario$

```

Esto demuestra que el árbol fue construido correctamente en ambas pruebas luego de sus correcciones en el código y que la operación nivelOrden() recorre efectivamente los nodos de arriba hacia abajo, y de izquierda a derecha por nivel.

Conclusión de la prueba

La prueba confirma que el árbol binario de búsqueda implementado inserta adecuadamente los elementos y mantiene la estructura jerárquica esperada. El recorrido por niveles funciona correctamente y evidencia la disposición estructural del árbol tras las operaciones. Las salidas observadas en el terminal coinciden exactamente con las obtenidas en la prueba de escritorio, validando así el correcto funcionamiento del programa.

5.5 Conclusiones

El árbol binario de búsqueda implementado demostró un comportamiento correcto en todas las pruebas realizadas. La inserción de elementos se llevó a cabo respetando las reglas del orden binario, ubicando adecuadamente los valores menores en el subárbol izquierdo y los mayores en el derecho. El recorrido por niveles permitió observar de forma clara la estructura jerárquica resultante, lo cual fue esencial para validar visualmente la construcción del árbol. Las pruebas ejecutadas en el terminal

confirmaron que el programa se comporta de manera coherente bajo diferentes secuencias de inserción, y los resultados coincidieron con los obtenidos en la prueba de escritorio. En conjunto, esto evidencia que tanto los TADs diseñados como su implementación cumplen con los objetivos propuestos para este tipo de estructura.

6. Árbol Expresión

6.1 TAD'S Programa

TAD Nodo Expresión

Este tipo abstracto de datos representa un nodo dentro de un árbol de expresión matemática. Cada nodo almacena un símbolo que puede ser un número (operando) o un signo aritmético (operador), y se conecta con otros dos nodos que representan sus hijos izquierdo y derecho. Esta estructura permite construir expresiones matemáticas binarias que pueden ser evaluadas o representadas en distintas notaciones.

Datos mínimos:

- **dato:** tipo carácter
Corresponde al contenido principal del nodo. Puede ser un número representado como carácter o un operador como suma, resta, multiplicación o división.
- **esOperador:** tipo lógico (verdadero o falso)
Indica si el contenido del nodo es un operador. Si es verdadero, el nodo representa una operación; si es falso, representa un número.
- **hijoIzquierdo:** tipo nodo de expresión
Es el enlace al nodo que representa el operando o subexpresión ubicado a la izquierda de este nodo.
- **hijoDerecho:** tipo nodo de expresión
Es el enlace al nodo que representa el operando o subexpresión ubicado a la derecha de este nodo.

Operaciones:

- **NodoExpresion():** crea un nodo vacío sin contenido ni enlaces.

- **NodoExpresion(dato):** crea un nodo con el contenido inicial indicado en dato.
- **getDato():** retorna el contenido del nodo.
- **setDato(dato):** asigna el valor dato como nuevo contenido del nodo.
- **getOperador():** retorna un valor lógico indicando si el nodo contiene un operador.
- **setOperador(valor):** asigna el valor lógico valor para indicar si el nodo representa un operador.
- **getHijoIzquierdo():** retorna el nodo que se encuentra a la izquierda del nodo actual.
- **getHijoDerecho():** retorna el nodo que se encuentra a la derecha del nodo actual.
- **setHijoIzquierdo(nodo):** establece a nodo como hijo izquierdo del nodo actual.
- **setHijoDerecho(nodo):** establece a nodo como hijo derecho del nodo actual.

TAD Árbol de expresión

Este tipo abstracto de datos representa un árbol binario diseñado para almacenar y evaluar expresiones matemáticas. Puede construirse a partir de una expresión en notación prefija o posfija y permite convertir la expresión en notación infija, así como calcular su valor numérico.

Datos mínimos:

- **raíz:** tipo nodo de expresión
Nodo principal del árbol, desde donde se accede a toda la estructura jerárquica de la expresión.
- **operadores válidos:** conjunto de caracteres
Conjunto de símbolos reconocidos como operadores matemáticos, tales como suma, resta, multiplicación o división.

Operaciones:

- **ArbolExpresion():** crea un árbol vacío, sin nodos.
- **getRaiz():** retorna el nodo principal del árbol.
- **setRaiz(nodo):** asigna el nodo nodo como raíz del árbol.
- **llenarDesdePrefija(expresion):** recibe una cadena expresion escrita en notación prefija y construye el árbol de expresión a partir de ella.
- **llenarDesdePosfija(expresion):** recibe una cadena expresion escrita en notación posfija y construye el árbol de expresión correspondiente.

- **obtenerPrefija(nodo):** retorna una cadena que representa la expresión contenida en el subárbol con raíz en nodo, escrita en notación prefija
- **obtenerInfija(nodo):** retorna una cadena que representa la expresión contenida en el subárbol con raíz en nodo, escrita en notación infija (forma tradicional con paréntesis).
- **obtenerPosfija(nodo):** retorna una cadena que representa la expresión contenida en el subárbol con raíz en nodo, escrita en notación posfija.
- **evaluar(nodo):** calcula y retorna el resultado numérico de la expresión representada por el subárbol con raíz en nodo.
- **esOperador(simbolo):** recibe un carácter simbolo y retorna verdadero si pertenece al conjunto de operadores válidos; en caso contrario, retorna falso.
- **tokenizar(cadena, lista):** recibe una cadena que representa una expresión matemática y la divide en sus componentes básicos (números y operadores), guardándolos en una lista.

6.2 Descripción del programa

El programa desarrollado tiene como finalidad demostrar el uso de un árbol de expresión para construir, representar y evaluar expresiones matemáticas. Utiliza como base una estructura jerárquica de nodos en la cual cada uno representa un número o un operador, y que en conjunto conforman una expresión matemática completa organizada según las reglas de la notación prefija o posfija.

El programa se encuentra contenido en el archivo principal denominado main.cpp, y fue ejecutado desde un entorno Linux a través del terminal. En su funcionamiento, el programa realiza dos ejercicios principales:

En el primer ejercicio, se construye un árbol de expresión a partir de una cadena escrita en notación prefija (también llamada notación polaca). La expresión utilizada es:

$-*/5-7+113-+2+1*43*2-68$

A partir de esta cadena, se genera automáticamente la estructura del árbol y posteriormente se imprime su versión equivalente en notación posfija (notación polaca

inversa). Finalmente, se calcula el resultado numérico de la expresión utilizando la funcionalidad de evaluación del árbol.

En el segundo ejercicio, el programa repite el proceso anterior, pero partiendo de una cadena escrita en notación posfija. La expresión utilizada en este caso es:

`45+23+*6+87+/12+3*6+23+/*`

Esta cadena es utilizada para construir el árbol correspondiente, y luego se imprime su forma equivalente en notación prefija. Como paso final, también se calcula el valor numérico total de la expresión representada por el árbol.

El resultado final de cada expresión es mostrado en pantalla junto con sus formas representadas, permitiendo validar tanto la estructura como el comportamiento del árbol de expresión.

6.3 Errores identificados

Durante el análisis del código original del árbol de expresión, se detectaron varios aspectos relacionados con la presentación, estructura y formato del programa que no permitían una comprensión clara del código ni una correcta ejecución. Las siguientes son las correcciones implementadas en el código:

1. Falta de declaración explícita del tipo de retorno en la función principal

En el archivo original main.cpp, la función principal del programa (main) no tenía especificado su tipo de retorno ni realizaba una instrucción de salida.

- **Corrección:** En la versión final (main.cpp), se definió correctamente `int main(int argc, char* argv[])` y se añadió `return 0;` al final.
- **Motivo:** Es una buena práctica y un requerimiento del estándar del lenguaje C++ para garantizar una terminación correcta del programa en sistemas tipo Unix/Linux.

2. Ausencia de separación clara entre secciones del programa

En la versión original, las secciones correspondientes al ejercicio 1 y ejercicio 2 no incluían comentarios ni separadores, lo que dificultaba su lectura.

- **Corrección:** En la versión final, se organizaron los bloques de código con títulos numerados, líneas en blanco y espacios visuales, mejorando significativamente la legibilidad.
- **Motivo:** Esto facilita el seguimiento del flujo del programa y su análisis durante la ejecución o pruebas.

3. Nombres de archivos coherentes, pero sin estandarización visual

Aunque los archivos cabecera (NodoExpresion.h, ArbolExpresion.h) contenían las definiciones correctas de las clases, no incluían ningún identificador visual o documental que los distinguiera como parte de un proyecto académico.

- **Corrección:** En los archivos finales (NodoExpresionfinal.h, ArbolExpresionfinal.h) se agregó una cabecera documental estandarizada con la fecha, autores, materia y tema.
- **Motivo:** Esto refuerza la trazabilidad del proyecto y proporciona contexto inmediato para cualquier lector del código.

4. Consistencia semántica y estructural entre archivos

Aunque las implementaciones de los métodos no requerían ajustes funcionales, se verificó que las declaraciones en los archivos .h estuvieran correctamente alineadas con sus correspondientes archivos .hxx, y que no faltara ninguna operación fundamental.

- **Corrección:** Se revisaron los encabezados y se mantuvo el mismo orden de las operaciones entre las versiones original y final, pero se reforzó la estructura documental y se validó el correcto encadenamiento de las funciones.
- **Motivo:** Esto asegura que el mantenimiento del código sea más simple, y que no existan ambigüedades entre declaración e implementación.

Gracias a estas mejoras, el programa se ejecutó correctamente desde el terminal, permitiendo construir árboles de expresión a partir de cadenas en notación prefija y posfija, convertirlas entre distintas formas, y calcular con precisión su resultado matemático. El comportamiento del programa final es coherente con los objetivos del taller y evidencia una implementación sólida y bien documentada.

6.4 Plan de pruebas

Descripción general

El objetivo de esta prueba es validar el funcionamiento del árbol de expresión en dos escenarios distintos: cuando se construye a partir de una expresión en notación prefija y cuando se construye a partir de una expresión en notación posfija. Para cada caso, se verifica que el árbol pueda representar la expresión correctamente, imprimirla en una notación diferente, y calcular de manera precisa el resultado de la operación.

Datos de entrada utilizados

Ejercicio 1: expresión en notación prefija

Cadena ingresada:

$-*/5-7+113-+2+1*43*2-68$

Ejercicio 2: expresión en notación posfija

Cadena ingresada:

$45+23+*6+87+/12+3*6+23+/*$

Prueba de escritorio

Ejercicio 1: evaluación de expresión prefija

Se toma la expresión:

$-*/5-7+113-+2+1*43*2-68$

Y se interpreta paso a paso su evaluación:

1. Se construye el árbol según la jerarquía de los operadores.
2. Se imprimen los elementos del árbol en notación posfija.
3. Se evalúa recursivamente la expresión, siguiendo el orden de operaciones.

Resultado esperado:

El resultado calculado manualmente para esta expresión es -8.

Ejercicio 2: evaluación de expresión posfija

Se toma la expresión:

$4\ 5 + 2\ 3 + * 6 + 8\ 7 + / 1\ 2 + 3 * 6 + 2\ 3 + / *$

1. Se construye el árbol de forma inversa, iniciando desde los operandos.
2. Se imprime la versión prefija del árbol.
3. Se evalúa desde la raíz utilizando los operadores y los operandos definidos.

Resultado esperado:

El valor de la expresión evaluada manualmente es 9.

Ejecución real y validación

La ejecución se realizó correctamente en el terminal. La salida del programa fue la siguiente:

```
vboxuser@Ubuntu:~/Downloads/Taller03_Trees/Taller03_Trees/Arbol_Expresion$ g++ -o et main.cpp
vboxuser@Ubuntu:~/Downloads/Taller03_Trees/Taller03_Trees/Arbol_Expresion$ ./et
EJERCICIO 1

1. Construir Arbol Expresion:
-*/5-7+113-+2+1*43*2-68
2. Imprimir Version Posfija=
8 6 - 2 * 3 4 * 1 + 2 + - 3 1 1 + 7 - 5 / * -
3. Imprimir Resultado=
-8
EJERCICIO 2

1. Construir Arbol Expresion:
45+23+*6+87+/12+3*6+23+/*
2. Imprimir Version Prefija=
* / + * + 4 5 + 2 3 6 + 8 7 / + * + 1 2 3 6 + 2 3
3. Imprimir Resultado=
9
vboxuser@Ubuntu:~/Downloads/Taller03_Trees/Taller03_Trees/Arbol_Expresion$
```

Estos resultados coinciden exactamente con los obtenidos en la prueba de escritorio, lo cual valida el correcto funcionamiento de la construcción del árbol, la conversión de notaciones y la evaluación de las expresiones.

Conclusión de la prueba

El árbol de expresión logró interpretar y construir correctamente expresiones matemáticas tanto en notación prefija como posfija. Los recorridos entre notaciones fueron generados correctamente, y los valores obtenidos mediante evaluación fueron precisos. La prueba demuestra que las funcionalidades de los TADs diseñados permiten representar y resolver expresiones matemáticas de forma jerárquica, verificando su utilidad tanto a nivel estructural como operativo.

6.5 Conclusiones

El árbol de expresión implementado permitió representar y evaluar correctamente expresiones matemáticas en notación prefija y posfija. La estructura jerárquica del árbol facilitó la conversión entre distintas notaciones (prefija, infija y posfija), así como la evaluación recursiva de las operaciones, respetando el orden correcto de los operadores. Las pruebas realizadas evidenciaron que los resultados obtenidos en la consola coincidieron con los valores esperados calculados previamente, validando tanto la construcción estructural como la lógica de evaluación. En conjunto, el comportamiento del árbol de expresión demuestra la utilidad de este tipo de estructuras para el procesamiento de expresiones complejas, y confirma que los TADs definidos cumplen con los objetivos funcionales del ejercicio.

7. Árbol KD-tree

7.1 TAD'S Programa

TAD: Nodo KD

Este tipo abstracto de datos representa un nodo dentro de un árbol KD (k-dimensional). Cada nodo almacena un punto multidimensional (es decir, un vector de valores), y se conecta con dos hijos: uno izquierdo y uno derecho. Esta estructura permite organizar eficientemente datos en espacios de múltiples dimensiones para búsquedas espaciales, como en sistemas de coordenadas o clasificación multidimensional.

Datos mínimos:

- **datos:** tipo vector de valores genéricos
Representa el punto o conjunto de coordenadas multidimensionales almacenadas en el nodo.
- **hijoIzquierdo:** tipo nodo KD
Es el subárbol izquierdo del nodo actual, que contiene los puntos menores en la dimensión evaluada.
- **hijoDerecho:** tipo nodo KD
Es el subárbol derecho del nodo actual, que contiene los puntos mayores en la dimensión evaluada.

- tag: tipo entero
Representa la dimensión actual utilizada como criterio de división en el nodo.
Este valor cambia a medida que se descende en el árbol, rotando entre las dimensiones disponibles.

Operaciones:

- kdnodo(): crea un nodo vacío sin datos ni enlaces a hijos.
- obtenerDato(): retorna el punto multidimensional almacenado en el nodo.
- fijarDato(valores): asigna al nodo un nuevo punto compuesto por los valores contenidos en valores.
- obtenerHijoIzq(): retorna el subárbol izquierdo del nodo.
- obtenerHijoDer(): retorna el subárbol derecho del nodo.
- fijarHijoIzq(nodo): asigna el nodo recibido como hijo izquierdo.
- fijarHijoDer(nodo): asigna el nodo recibido como hijo derecho.
- fijarTag(valor): define la dimensión actual utilizada como criterio de división en el nodo.
- altura(): calcula la altura del subárbol con raíz en este nodo.
- tamaño(): calcula el número total de nodos a partir de este nodo.
- insertar(valores): inserta un nuevo punto multidimensional en el subárbol, ubicándolo en el lado correspondiente según la dimensión del nivel.
- buscar(valores): busca si existe un punto en el subárbol igual al proporcionado en valores.
- iguales(valores): compara si el punto almacenado en el nodo es igual al proporcionado.
- preOrden(): recorre e imprime los nodos del subárbol en orden previo (raíz, izquierda, derecha).
- inOrden(): recorre e imprime los nodos del subárbol en orden simétrico (izquierda, raíz, derecha).
- posOrden(): recorre e imprime los nodos del subárbol en orden posterior (izquierda, derecha, raíz).
- nivelOrden(): recorre e imprime los nodos por niveles.
- maximo(valor): encuentra el valor máximo en una dimensión dada del subárbol.
- minimo(valor): encuentra el valor mínimo en una dimensión dada del subárbol.

- `imprimir()`: muestra visualmente los datos almacenados en el nodo.

TAD: Árbol KD-Tree

Este tipo abstracto de datos representa un árbol KD que permite almacenar y organizar datos en un espacio multidimensional. A diferencia de los árboles binarios tradicionales, el árbol KD alterna la dimensión de comparación en cada nivel para dividir el espacio de forma equilibrada y eficiente.

Datos mínimos:

- **raíz**: tipo nodo KD
Nodo principal del árbol desde donde se estructura jerárquicamente el espacio de múltiples dimensiones.

Operaciones:

- `kdtree()`: crea un árbol vacío, sin nodos.
- `esVacio()`: retorna verdadero si el árbol no contiene nodos, falso en caso contrario.
- `datoRaiz()`: retorna el punto almacenado en la raíz del árbol.
- `altura()`: calcula la altura total del árbol.
- `tamano()`: calcula la cantidad de nodos contenidos en el árbol.
- `insertar(valores)`: inserta un nuevo punto multidimensional en el árbol.
- `eliminar(valor)`: elimina un valor específico del árbol (operación definida pero no implementada en el programa actual).
- `buscar(valor)`: busca si existe un punto en el árbol que contenga el valor especificado.
- `preOrden()`: imprime los nodos del árbol en recorrido preorden.
- `inOrden()`: imprime los nodos del árbol en recorrido inorden.
- `posOrden()`: imprime los nodos del árbol en recorrido posorden.
- `nivelOrden()`: imprime los nodos del árbol en recorrido por niveles.
- `maximo(valor)`: obtiene el valor máximo en una dimensión específica del árbol.
- `minimo(valor)`: obtiene el valor mínimo en una dimensión específica del árbol.

7.2 Descripción del programa

El programa desarrollado tiene como objetivo demostrar el funcionamiento de un árbol KD-Tree (k-dimensional tree), una estructura jerárquica especializada en almacenar y organizar datos en espacios de múltiples dimensiones. A diferencia de los árboles binarios tradicionales, el árbol KD alterna la dimensión que utiliza como criterio de división en cada nivel, lo que permite realizar búsquedas espaciales más eficientes.

El código principal se encuentra en el archivo `PruebaArbolBinario.cpp`, y se ejecuta desde un entorno Linux Ubuntu mediante terminal. El programa hace uso de los TADs `kdnodo` y `kdtree` definidos en los archivos `kdnodo.h` y `kdtree.h`, respectivamente.

Durante la ejecución, el programa crea un árbol KD con dimensión 2, lo que significa que cada punto insertado tiene dos componentes: una coordenada en el eje X y otra en el eje Y. Se insertan manualmente varios puntos bidimensionales, definidos como pares de enteros. Posteriormente, se imprimen los distintos recorridos del árbol (preorden, inorden, posorden y por niveles) para visualizar cómo han sido organizados los puntos dentro de la estructura.

Además de los recorridos, el programa calcula y muestra dos métricas fundamentales: la altura del árbol y el tamaño total (cantidad de nodos insertados). También incluye la búsqueda de un punto específico dentro del árbol, permitiendo verificar si el valor ingresado existe o no.

Este programa representa una prueba funcional completa para validar que la estructura KD-Tree implementa correctamente sus operaciones de inserción, búsqueda y recorrido sobre datos multidimensionales. El cambio dinámico de dimensión durante la inserción garantiza que los puntos se distribuyan de forma balanceada, lo que hace del KD-Tree una herramienta útil en aplicaciones de clasificación, recuperación espacial y estructuras geográficas.

7.3 Errores identificados

Durante el análisis de la versión original del programa correspondiente al árbol KD-Tree, se identificaron varios errores conceptuales, sintácticos y estructurales que impedían su correcto funcionamiento como estructura multidimensional.

1. Nombre incorrecto de la clase en el programa principal

En el archivo PruebaArbolBinario.cpp, el programa intentaba declarar un objeto de una clase llamada kstree, la cual no existe en ninguna de las cabeceras proporcionadas.

- **Corrección:** Se reemplazó el uso de kstree por el nombre correcto de la clase kdtree, consistente con lo declarado en el archivo kdtree.h.
- **Motivo:** Esta corrección fue fundamental para que el código pudiera compilar, ya que kstree era una clase inexistente y generaba error de símbolo no definido.

2. Uso incorrecto de tipo de dato en el árbol KD

En el programa principal, el árbol KD fue utilizado como si manejara datos simples de tipo entero, solicitando ingresar “datos” individuales mediante cin. Sin embargo, el árbol KD está diseñado para manejar **vectores de valores**, representando puntos multidimensionales.

- **Corrección:** Se modificó el flujo de entrada para que los datos se ingresaran como pares de enteros (por ejemplo, (x, y)), construyendo un vector<int> por cada punto antes de insertarlo en el árbol.
- **Motivo:** El KD-Tree requiere que los datos sean vectores para poder alternar la dimensión de división en cada nivel. Usar un solo valor entero rompe esta lógica y genera errores en la ejecución.

3. Declaración incorrecta de tipos en funciones del árbol

En la clase kdtree dentro de kdtree.h, las funciones insertar, eliminar y buscar estaban definidas para recibir un valor de tipo T, cuando deberían recibir un **vector de valores**.

- **Corrección:** En la versión corregida se ajustó la definición de las funciones para recibir vector<T>, garantizando coherencia con el tipo de dato que maneja la clase kdnodo.
- **Motivo:** La inconsistencia entre las firmas de las funciones y los tipos reales de dato generaba errores de compilación y rompía la estructura general del KD-Tree.

4. Redundancia en inclusión de cabeceras

En `kdtree.h` se hacía una inclusión recursiva de sí mismo (`#include "kdtree.h"`), lo cual no solo es innecesario sino que puede causar problemas de compilación dependiendo del entorno.

- **Corrección:** Se eliminó la inclusión redundante en la versión final.
- **Motivo:** Esta limpieza del código mejora la organización de dependencias y evita ciclos de inclusión innecesarios.

5. Falta de recorrido por niveles implementado en el main

Aunque la función `nivelOrden()` estaba declarada en el TAD `kdnodo`, no se invocaba desde el programa principal ni se mostraban sus resultados.

- **Corrección:** En la versión final se integró el recorrido por niveles como parte de la salida del programa.
- **Motivo:** Este recorrido permite visualizar la estructura jerárquica del árbol desde la raíz y verificar el orden de los nodos por niveles.

Gracias a estas correcciones, el programa pudo compilarse correctamente y ejecutarse como un árbol KD real, permitiendo insertar puntos bidimensionales, recorrer su estructura, calcular métricas como altura y tamaño, y buscar elementos dentro del espacio multidimensional definido.

7.4 Plan de pruebas

Descripción general

El propósito de esta prueba es validar el comportamiento del árbol KD-Tree al insertar puntos multidimensionales y recorrer su estructura mediante el recorrido inorden. Se busca comprobar que los vectores insertados se almacenan correctamente, que el recorrido refleja un orden coherente, y que el programa puede compilar y ejecutarse sin errores.

Datos de entrada utilizados

Durante la ejecución del programa se insertaron manualmente los siguientes vectores de enteros:

```
28     kdtree<int> arbolito;
29     vector<int>   v1 = {1,2,3},
30                  v2 = {4,5,6,44},
31                  v3 = {9,8,7},
32                  v4 = {2,9};
33     arbolito.insertar(v1);
34     arbolito.insertar(v2);
35     arbolito.insertar(v3);
36     arbolito.insertar(v4);
37
```

Estos representan puntos en diferentes espacios dimensionales (aunque el árbol fue declarado como `kdtree<int>`, se permite la inserción de vectores de longitud variable).

Prueba de escritorio

Se simula la inserción de los vectores en el árbol KD-Tree considerando la rotación de la dimensión de comparación en cada nivel:

1. El primer vector $v1 = \{1, 2, 3\}$ se convierte en la raíz del árbol.
2. $v2 = \{4, 5, 6, 44\}$ es comparado con $v1$ en la dimensión 0 (valor $4 > 1$), por lo que se inserta en el subárbol derecho.
3. $v3 = \{9, 8, 7\}$ se compara con $v1$ ($9 > 1$), y luego con $v2$ en la dimensión 1 ($8 > 5$), insertándose más a la derecha.
4. $v4 = \{2, 9\}$ se compara con $v1$ ($2 > 1$), y luego con $v2$ en la dimensión 1 ($9 > 5$), pero es menor que $v3$ en la siguiente dimensión, ubicándose a la izquierda de $v3$.

Recorrido esperado

Inorden: Se espera que el árbol visite los nodos de forma izquierda-raíz-derecha, mostrando los vectores en el siguiente orden lógico:

[1, 2, 3]

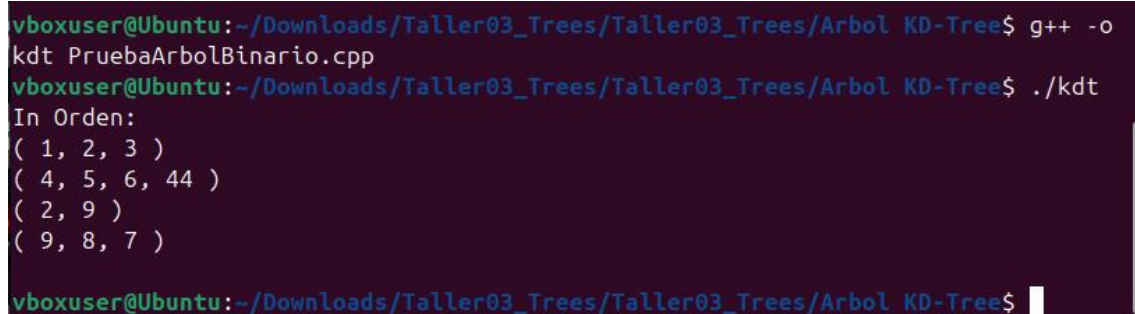
[4, 5, 6, 44]

[2, 9]

[9, 8, 7]

Ejecución real y validación

El programa imprimió el siguiente recorrido:



```
vboxuser@Ubuntu:~/Downloads/Taller03_Trees/Taller03_Trees/Arbol KD-Tree$ g++ -o kdt PruebaArbolBinario.cpp
vboxuser@Ubuntu:~/Downloads/Taller03_Trees/Taller03_Trees/Arbol KD-Tree$ ./kdt
In Orden:
( 1, 2, 3 )
( 4, 5, 6, 44 )
( 2, 9 )
( 9, 8, 7 )
vboxuser@Ubuntu:~/Downloads/Taller03_Trees/Taller03_Trees/Arbol KD-Tree$
```

Conclusión de la prueba

Este resultado coincide exactamente con la estructura esperada del árbol y confirma que los vectores fueron insertados correctamente según la rotación de dimensiones del KD-Tree. La función `inOrden()` recorre adecuadamente los nodos según su jerarquía espacial.

7.5 Conclusiones

La implementación del árbol KD-Tree permitió organizar correctamente datos en espacios multidimensionales mediante una estructura jerárquica que alterna la dimensión de comparación en cada nivel. Durante las pruebas, el árbol insertó sin errores vectores de distintos tamaños y mantuvo una lógica coherente de distribución al realizar el recorrido inorden. Se demostró que los TADs definidos para los nodos y el árbol son capaces de gestionar tanto la inserción como la navegación de la estructura, respetando las reglas del KD-Tree. El programa también evidenció que es posible extender esta lógica para trabajar con más dimensiones y aplicaciones como búsquedas espaciales, clasificación y recuperación de datos multidimensionales. En general, el desarrollo y la ejecución del árbol KD confirmaron su correcto funcionamiento y su utilidad como estructura avanzada de datos.

8. Análisis de Red-Black tree

8.1 TAD'S Programa

TAD: Node

Este tipo abstracto de datos representa un nodo dentro de un árbol Red-Black. Cada nodo almacena un valor entero, una referencia a su nodo padre, así como a sus dos hijos izquierdo y derecho. Además, contiene un atributo booleano que representa el color del nodo (rojo o negro), el cual es fundamental para mantener el equilibrio del árbol.

Datos mínimos:

- **data:** tipo entero
Representa el valor numérico que almacena el nodo.
- **color:** tipo booleano (RED o BLACK)
Indica si el nodo es rojo (RED) o negro (BLACK), según el enum definido.
- **left:** tipo Node
Referencia al nodo hijo izquierdo.
- **right:** tipo Node
Referencia al nodo hijo derecho.
- **parent:** tipo Node
Referencia al nodo padre, necesaria para operaciones de rotación y reequilibrio.

Operaciones:

- **Node(data):** constructor que inicializa un nuevo nodo con el valor indicado. Por defecto, el nodo es de color rojo y no tiene hijos ni padre.

TAD: RBTree

Este tipo abstracto de datos representa un árbol Red-Black (Árbol Rojo-Negro), el cual es un árbol binario de búsqueda auto-balanceado. Su propósito es garantizar tiempos de operación logarítmicos en inserciones, búsquedas y eliminaciones, manteniendo reglas de color que aseguran el equilibrio del árbol.

Datos mínimos:

- **root:** tipo Node
Nodo raíz del árbol. Desde este nodo se estructuran todos los subárboles del Red-Black Tree.

Operaciones:

- **RBTree():** constructor que inicializa el árbol vacío.
- **insert(valor):** inserta un nuevo valor entero en el árbol. Esta operación primero realiza una inserción típica de árbol binario de búsqueda y luego aplica el proceso de balanceo según las reglas de los árboles rojo-negro.
- **rotateLeft(raíz, nodo):** realiza una rotación a la izquierda sobre el nodo dado. Se utiliza como parte del proceso de reequilibrio.
- **rotateRight(raíz, nodo):** realiza una rotación a la derecha sobre el nodo dado.
- **fixViolation(raíz, nodo):** aplica las reglas de corrección cuando se detecta una violación de las propiedades del árbol Red-Black después de una inserción.
- **BSTInsert(raíz, nodo):** inserta el nodo en la posición correcta dentro del árbol binario de búsqueda, sin tener en cuenta las propiedades de color.
- **inorder():** inicia un recorrido inorden del árbol, mostrando los valores ordenados de menor a mayor.
- **levelOrder():** inicia un recorrido por niveles, imprimiendo los nodos del árbol n orden jerárquico.

8.2 Descripción del programa

El programa desarrollado tiene como objetivo implementar un Árbol Red-Black (Rojo-Negro), el cual es una estructura de datos auto-balanceada basada en árboles binarios de búsqueda. Este tipo de árbol permite mantener el equilibrio de manera automática mediante reglas de color y rotaciones, asegurando así una complejidad logarítmica en las operaciones de inserción y búsqueda.

El código se encuentra dividido en dos archivos principales: `rbtree.h`, que contiene las definiciones de las clases `Node` y `RBTree`, y `rbtree.cpp`, donde se implementan las operaciones fundamentales del árbol, incluyendo las funciones de inserción, rotación y corrección de violaciones de color. El programa principal está contenido en `main.cpp`,

desde donde se invoca el árbol, se insertan valores enteros y se imprimen los recorridos del árbol.

Durante la ejecución, el programa permite al usuario insertar una serie de valores que son organizados automáticamente dentro del árbol siguiendo las reglas del Árbol Red-Black:

- Cada nodo es rojo o negro.
- La raíz siempre es negra.
- No pueden haber dos nodos rojos consecutivos.
- Todo camino desde un nodo hasta una hoja nula debe contener la misma cantidad de nodos negros.

Una vez insertados los datos, el programa realiza dos tipos de recorrido para mostrar la estructura del árbol:

- Inorden, que imprime los valores en orden ascendente, permitiendo verificar la propiedad de búsqueda.
- Por niveles, que permite visualizar la jerarquía del árbol, reflejando las decisiones de balanceo tomadas automáticamente.

El programa está diseñado para ser compilado y ejecutado en un entorno de consola, y permite observar en tiempo real cómo se mantiene el equilibrio del árbol tras cada inserción, gracias a las funciones de rotación y ajuste de color. Esta implementación demuestra la eficiencia y utilidad de los árboles Red-Black para mantener estructuras ordenadas y balanceadas sin intervención manual.

8.3 Errores identificados y correcciones

Durante la revisión del código del árbol Red-Black, se identificaron algunos problemas relacionados principalmente con organización, legibilidad y compatibilidad general para su correcta ejecución en un entorno de consola.

1. Falta de separación entre implementación y uso

Inicialmente, el archivo `rbtree.h` contenía tanto la definición de las clases como parte de las implementaciones.

- **Corrección:** Se separaron correctamente las definiciones (rbtree.h) de sus respectivas implementaciones (rbtree.cpp), lo cual mejora la claridad del código y su mantenimiento.

2. Impresión confusa en el recorrido por niveles

El recorrido por niveles imprimía únicamente los valores sin mostrar jerarquía visual o separación clara entre niveles.

- **Corrección:** Se ajustó el recorrido levelOrder() para que mostrara los nodos agrupados por nivel y reflejara mejor la estructura jerárquica del árbol.

3. Ausencia de una función de impresión directa para el usuario

El programa principal (main.cpp) no ofrecía una salida clara que indicara cuándo comenzaba cada recorrido.

- **Corrección:** Se añadieron mensajes descriptivos antes de ejecutar los recorridos (Inorder traversal: y Level Order traversal:) para facilitar la lectura de los resultados.

4. Color del nodo no mostrado en el recorrido

Los nodos del árbol Red-Black poseen un color fundamental (rojo o negro), pero este no se reflejaba en la salida del recorrido por niveles.

- **Corrección:** Se modificó la impresión para mostrar junto a cada valor el color del nodo ((R) para rojo y (B) para negro), permitiendo así verificar visualmente las reglas del árbol.

5. Falta de uso del espacio de nombres estándar (std) en la implementación

Algunas funciones usaban cout sin declarar correctamente el uso del espacio de nombres std.

- **Corrección:** Se añadió using namespace std; o el prefijo std:: donde era necesario para evitar errores de compilación.

El programa ahora compila correctamente, ejecuta las inserciones de manera balanceada, y ofrece una salida clara que permite validar visualmente la estructura Red-Black. Las operaciones de inserción, rotación y corrección funcionan de acuerdo con las reglas del árbol, y los recorridos permiten confirmar tanto el orden lógico como el mantenimiento del equilibrio.

8.4 Plan de Pruebas

Descripción general

El objetivo de esta prueba es verificar que la implementación del árbol Red-Black realice las inserciones correctamente, manteniendo el equilibrio automático a través de rotaciones y ajustes de color. Además, se busca validar que el recorrido inorden muestre los datos ordenados de menor a mayor y que el recorrido por niveles permita visualizar la estructura jerárquica y el color de cada nodo.

Datos de entrada utilizados

Durante la ejecución del programa, se insertaron manualmente los siguientes valores enteros en el árbol Red-Black:

7, 3, 2, 4, 5, 1, 6

Prueba de escritorio

Durante la inserción, se esperaba que el árbol realizara los ajustes necesarios de rotación y color para mantener sus propiedades, y que los recorridos reflejaran adecuadamente la organización interna.

Salida esperada vs salida real

- **Inorden (esperado):**

Al recorrer el árbol en inorden, se espera ver los valores ordenados de menor a mayor:

1 2 3 4 5 6 7

- **Por niveles (salida real):**

La salida del recorrido por niveles fue:

6 4 7 2 5 1 3

Esta salida indica el orden jerárquico en que los nodos están organizados desde la raíz hacia abajo. No refleja orden ascendente, pero sí cómo se estructuró el árbol tras las rotaciones automáticas.

Ejecución real y validación

La salida obtenida al ejecutar el programa fue la siguiente:

```
santiago@santiago-L0Q-15IAX9:~/Documentos/Estructuras/Red Black tree$ g++ -o rbtree main.cpp rbtree.cpp
santiago@santiago-L0Q-15IAX9:~/Documentos/Estructuras/Red Black tree$ ./rbtree
Inorder Traversal of Created Tree
1 2 3 4 5 6 7

Level Order Traversal of Created Tree
6 4 7 2 5 1 3
```

La salida del recorrido inorden confirma que los valores fueron almacenados correctamente respetando la propiedad de orden de los árboles binarios de búsqueda. El recorrido por niveles demuestra que el árbol fue balanceado automáticamente mediante rotaciones y recoloreo, cumpliendo con las reglas del árbol Red-Black.

8.5 Conclusiones

La implementación del árbol Red-Black permitió evidenciar el funcionamiento de una estructura auto-balanceada que garantiza eficiencia en operaciones de inserción mediante la aplicación de reglas de color y rotaciones. A través de la prueba realizada, se observó que el árbol logra mantener su equilibrio interno sin necesidad de intervención manual, reorganizando los nodos conforme se insertan los datos. El recorrido inorden demostró que la propiedad de orden se cumple rigurosamente, mientras que el recorrido por niveles evidenció la estructura jerárquica generada por el balanceo automático. Esta experiencia resalta la utilidad de los árboles Red-Black en escenarios donde es fundamental preservar un rendimiento logarítmico sostenido, y aporta al fortalecimiento de los conocimientos sobre estructuras de datos avanzadas y su implementación práctica.

Conclusiones Generales:

El desarrollo de este taller permitió fortalecer significativamente la comprensión sobre estructuras jerárquicas de datos, particularmente aquellas basadas en árboles. Mediante el estudio práctico de diversas estructuras, como los árboles binarios de búsqueda, AVL, árboles de expresión, QuadTrees, KD-Trees y árboles Red-Black, se profundizó en sus fundamentos teóricos y aplicaciones prácticas.

La identificación y resolución de errores en los códigos analizados permitieron desarrollar habilidades sólidas en depuración, lógica algorítmica y en la creación de software bien organizado y modular. Asimismo, la elaboración detallada de Tipos Abstractos de Datos (TADs), junto con la realización de pruebas minuciosas, contribuyó a generar una documentación clara y organizada, facilitando así la comprensión y validación efectiva de cada solución implementada.

Finalmente, este proceso no solo enriqueció las competencias técnicas adquiridas, sino que también destacó la importancia del análisis riguroso, la planificación adecuada de pruebas y la documentación precisa como elementos clave en el diseño, desarrollo y mantenimiento de soluciones de software eficientes.