Implementación y Pruebas de Árboles en C++

Autores: Jeison Alfonso, Douglas Díaz
Fecha: 03/04/2025
Curso: Estructuras de Datos
Institución: Pontificia Universidad Javeriana

Índice

Objetivos

- Revisar y corregir las implementaciones modularizadas de las siguientes estructuras de árboles en C++:
    1. QuadTree
    2. KD-Tree
    3. Árbol Rojo-Negro (Red-Black Tree)
    4. Árbol
    5. Árbol Binario
    6. Árbol Binario AVL
    7. Árbol Binario Ordenado
    8. Árbol de Expresión
- Identificar y solucionar errores en la lógica de cada estructura, tales como:
    - Cálculo correcto de la altura y el tamaño.
    - Alternancia de dimensiones en el KD-Tree.
    - Balanceo automático en el Árbol Rojo-Negro y AVL.
    - Inserción y eliminación de nodos en árboles binarios y AVL.
    - Evaluación de expresiones aritméticas en el Árbol de Expresión.
    - Manejo correcto de la raíz (o nodo centinela) en estructuras generales.
- Desarrollar programas principales (mains) que demuestren el funcionamiento correcto de cada estructura, mediante inserciones, recorridos y operaciones específicas.
- Documentar cada archivo con comentarios claros y explicativos que faciliten su lectura y mantenimiento.
- Elaborar un informe que contenga el análisis, las pruebas realizadas y las conclusiones sobre cada implementación, incluyendo su TAD correspondiente.

Introducción

Las estructuras de datos especializadas son fundamentales para el manejo eficiente de la información en aplicaciones complejas. Este informe se centra en la revisión, corrección, documentación y prueba de ocho estructuras de árbol implementadas en C++, cada una con propósitos particulares.

1. **QuadTree**: Una estructura que divide un espacio bidimensional en cuatro cuadrantes, ideal para el manejo de gráficos y sistemas de información geográfica.
2. **KD-Tree**: Diseñada para almacenar puntos en espacios k-dimensionales (en este caso 2D), alternando la dimensión de comparación, lo cual facilita búsquedas multidimensionales eficientes.
3. **Árbol Rojo-Negro**: Un árbol binario de búsqueda auto-balanceado que garantiza tiempos de operación logarítmicos mediante el uso de colores y rotaciones.
4. **Árbol Binario**: Una implementación base para estructuras jerárquicas, útil como punto de partida para árboles más específicos.
5. **Árbol Binario Ordenado**: Variante del árbol binario con inserción ordenada, que permite búsquedas eficientes siguiendo la propiedad BST (Binary Search Tree).
6. **Árbol AVL**: Una mejora del árbol binario de búsqueda que mantiene el equilibrio mediante rotaciones simples y dobles, asegurando un rendimiento óptimo en operaciones de búsqueda, inserción y eliminación.
7. **Árbol de Expresión**: Permite representar y evaluar expresiones aritméticas complejas utilizando notación prefija y posfija. Es útil en aplicaciones como compiladores, calculadoras y procesadores de fórmulas.

Se presenta el análisis detallado y la documentación técnica de cada módulo (archivos `.h`, `.hxx`, `.cpp`), así como la definición de sus respectivos TADs (TAD Nodo y TAD Árbol) y las pruebas realizadas para validar el correcto funcionamiento de cada implementación.

Desarrollo

3.1. Análisis y Documentación de Archivos

3.1.1. QuadTree

Documentación de nodo.h / Nodo.hxx:
- Funcionalidad:
Define la clase plantilla Nodo para el QuadTree. Cada nodo almacena un par de valores (por ejemplo, coordenadas X e Y) y tiene cuatro punteros para sus hijos: NW, NE, SW y SE.
- Errores Detectados:
• La función altura() en algunas versiones no retornaba el valor calculado.
• La función tamano() contaba incorrectamente los nodos utilizando el cálculo de la altura.

- Mejoras Realizadas:
  • Se corrigió altura() para retornar el valor correcto, comparando recursivamente la altura de cada subárbol.
  • Se reimplementó tamano() para que sume el nodo actual y, de forma recursiva, los nodos de cada subárbol.
- Documentación:
  Cada método cuenta con comentarios que explican su propósito, parámetros y valor de retorno.

Documentación de quadtree.h / quadtree.hxx:
- Funcionalidad:
  Encapsulan la clase Arbol (o QuadTree) que administra la estructura del QuadTree. Se maneja la raíz (tipo Nodo<T>) y se delegan las operaciones de inserción, búsqueda, recorridos (preorden y postorden) y consultas (altura y tamano) a los métodos de Nodo.
- Errores Detectados:
  • Se asumía que la raíz siempre existía, lo cual causaba errores al insertar en un árbol vacío.
- Mejoras Realizadas:
  • Se agregó verificación para crear el nodo raíz en el método insertar() si éste es nulo.
  • Se mejoró el manejo de consultas en caso de árbol vacío.
- Documentación:
  Los métodos están documentados con comentarios detallados que describen el flujo de ejecución y la interacción con los nodos.

3.1.2. KD-Tree

Documentación de nodo.h / Nodo.hxx:
- Funcionalidad:
  Define la clase Nodo para el KD-Tree. Cada nodo almacena un punto (por ejemplo, un par de coordenadas) y tiene dos punteros: izquierdo y derecho. Durante la inserción y búsqueda se alterna la dimensión de comparación según la profundidad.
- Errores Detectados:
  • Inicialmente faltaba la variable 'profundidad' para alternar entre comparar la primera y la segunda componente.
- Mejoras Realizadas:
  • Se implementaron los métodos insertar y buscar de manera recursiva utilizando la variable profundidad para determinar la dimensión de comparación.
- Documentación:
  Se añaden comentarios que explican cómo se utiliza la profundidad para alternar la dimensión y el proceso recursivo.

Documentación de kdtree.h / kdtree.hxx:
- Funcionalidad:
  Encapsulan la clase KDTree, que administra la raíz y delega las operaciones en la clase Nodo. Se definen métodos para insertar puntos, buscar un punto específico y realizar recorridos (preorden y postorden).
- Errores Detectados:

• Era necesario asegurar que, al insertar, se maneje el caso de un árbol vacío.
- Mejoras Realizadas:
• Se agregó verificación para la inicialización de la raíz y se pasa la profundidad inicial (0) al método recursivo de inserción.
- Documentación:
Cada método se documenta detalladamente, enfatizando el uso de la variable 'profundidad' para alternar la comparación de dimensiones.

Documentación de prueba_kd.cpp:
- Funcionalidad:
Programa principal para probar el KD-Tree. Se inicializa el árbol, se insertan varios puntos, se realizan recorridos y se busca un punto específico. Además, se muestran propiedades como la altura y el tamano.
- Documentación:
Incluye comentarios al inicio y a lo largo del código para explicar cada etapa (inserción, recorrido, búsqueda).

3.1.3. Árbol Rojo-Negro

Documentación de rbtree.h / rbtree.hxx:
- Funcionalidad:
Define la estructura del nodo (RBNode) y la clase RBTree para implementar un árbol rojo-negro. Cada nodo almacena un dato, un color (RED o BLACK) y punteros al padre, hijo izquierdo y derecho. El árbol se mantiene balanceado mediante rotaciones y un proceso de fixup después de cada inserción.
- Errores Detectados:
• Era necesario implementar correctamente el balanceo tras la inserción utilizando rotaciones (izquierda y derecha) y el procedimiento insertFixup.
- Mejoras Realizadas:
• Se implementaron las funciones leftRotate, rightRotate y insertFixup para garantizar que se mantengan las propiedades del árbol rojo-negro.
- Documentación:
Cada función cuenta con comentarios detallados que explican los casos manejados y cómo se corrige el árbol.

Documentación de prueba_rb.cpp:
- Funcionalidad:
Programa de prueba para el Árbol Rojo-Negro que inserta nodos, realiza un recorrido InOrder (mostrando el dato y el color de cada nodo) y busca un nodo específico.
- Documentación:
Se incluyen comentarios que describen el propósito del programa y la función de cada bloque de código.

3.1.4. Árbol

Documentación de nodo.h / nodo.hxx:
- Funcionalidad:

Define la clase Nodo para un Arbol. Cada nodo almacena un dato y punteros a sus hijos izquierdo y derecho.
- Errores Detectados:
  • Se debe garantizar que la inserción respete el orden del árbol.
- Mejoras Realizadas:
  • Se implementó el método insertar de forma recursiva, comparando el dato actual para ubicar el nuevo nodo en el lado correcto.
  • Se agregaron métodos para realizar recorridos (preorden, postorden), calcular la altura y el tamano, y buscar un nodo específico.
- Documentación:
  Cada método está documentado con comentarios que explican su funcionamiento, parámetros y valores de retorno.

Documentación de arbol.h / arbol.hxx:
- Funcionalidad:
  Encapsulan la clase Arbol que administra el árbol. Se gestiona la raíz y se delegan las operaciones a los métodos de la clase Nodo.
- Errores Detectados:
  • Es fundamental manejar el caso de árbol vacío al insertar o consultar.
- Mejoras Realizadas:
  • Se implementaron verificaciones para crear la raíz si ésta es nula y se definieron adecuadamente los métodos de consulta.
- Documentación:
  Se describen cada uno de los métodos con comentarios que explican el flujo de datos y la delegación de operaciones a los nodos.

Documentación de prueba_arbol.cpp:
- Funcionalidad:
  Programa de prueba para el árbol. Se demuestra la inserción de datos, los recorridos (preorden y postorden), la búsqueda de un nodo y el cálculo de la altura y el tamano.
- Documentación:
  Incluye comentarios explicativos en cada bloque de código.

TAD_Arbol.txt:
- Funcionalidad:
  Documento que especifica el Tipo Abstracto de Datos (TAD) para el árbol binario, listando las operaciones fundamentales y sus comportamientos esperados.
- Documentación:
  Contiene la especificación detallada de cada operación.

3.1.5. Árbol Binario AVL

Funcionalidad:

La clase `ArbolBinarioAVL` representa un árbol binario de búsqueda auto-balanceado mediante el criterio AVL (Adelson-Velsky y Landis). Está implementada como una clase plantilla que permite almacenar datos de cualquier tipo. Este árbol mantiene su balance

por medio de rotaciones simples y dobles, que se aplican automáticamente durante las operaciones de inserción y eliminación.

Estructura:

- La clase contiene un puntero a su nodo raíz (`raiz`) de tipo `NodoBinarioAVL<T>`.
- Cada nodo mantiene su dato, punteros a los hijos izquierdo y derecho, y es manejado por métodos internos que aseguran el balanceo.

Métodos principales:

- **`insertar(T& val)`**: Inserta un nuevo valor y ajusta el balance del árbol utilizando rotaciones si es necesario.
- **`eliminar(T& val)`**: Elimina un valor del árbol y realiza las rotaciones requeridas para mantener el balance.
- **`buscar(T& val)`**: Verifica si un valor existe en el árbol.
- **`altura(NodoBinarioAVL<T>* inicio)`**: Calcula recursivamente la altura del subárbol.
- **`tamano(NodoBinarioAVL<T>* inicio)`**: Cuenta el número de nodos en el subárbol.
- **Rotaciones**:
  - `giroDerecha()`, `giroIzquierda()`: Rotaciones simples.
  - `giroIzquierdaDerecha()`, `giroDerechaIzquierda()`: Rotaciones dobles.
- **Recorridos**:
  - `preOrden`, `inOrden`, `posOrden`, `nivelOrden`.

Mejoras realizadas:

- Se implementaron correctamente las rotaciones simples y dobles para mantener el equilibrio AVL.
- Se verificaron las condiciones de balanceo en cada operación de inserción y eliminación.
- Se incluyó la validación del caso base de árbol vacío y el control de altura de subárboles.

3.1.6. Árbol Binario ordenado

La clase ArbolBinario implementa un árbol binario de búsqueda (BST) mediante plantillas en C++. Su objetivo es organizar y almacenar datos de forma ordenada para facilitar las operaciones de inserción, búsqueda y eliminación. Esta clase puede utilizarse con cualquier tipo de dato que tenga operadores de comparación definidos.

Estructura:

- Se basa en el nodo NodoBinario<T>, el cual contiene un dato de tipo T y punteros a sus hijos izquierdo y derecho.
- La clase ArbolBinario administra la raíz del árbol (raiz) y ofrece métodos para modificar y consultar su estructura.
- Todas las operaciones principales (insertar, buscar, eliminar) están implementadas de forma recursiva, con excepción del recorrido por niveles.

Métodos principales:

- **insertar(val, nodo)**: Inserta el valor val en el subárbol que inicia en nodo. Si el árbol está vacío, se crea la raíz.
- **buscar(val)**: Retorna true si el valor está presente en el árbol, false en caso contrario.
- **eliminar(val)**: Elimina el nodo que contiene el valor val, reorganizando el árbol si es necesario.
- **altura(nodo)**: Retorna la altura del subárbol cuya raíz es nodo.
- **tamano(nodo)**: Retorna la cantidad de nodos en el subárbol.
- **datoRaiz()**: Retorna el dato almacenado en la raíz del árbol.
- **esVacio()**: Retorna true si la raíz es nula, lo que indica que el árbol está vacío.

Recorridos disponibles:

- **preOrden(nodo)**: Visita primero la raíz, luego el subárbol izquierdo y luego el derecho.
- **inOrden(nodo)**: Visita el subárbol izquierdo, luego la raíz y finalmente el derecho.
- **posOrden(nodo)**: Visita primero los subárboles izquierdo y derecho, y finalmente la raíz.
- **nivelOrden(nodo)**: Recorre el árbol por niveles usando una cola (BFS).

Mejoras implementadas:

- Se incluyeron verificaciones para manejar correctamente el árbol vacío al insertar.
- La eliminación contempla los tres casos estándar de BST: nodo hoja, nodo con un solo hijo y nodo con dos hijos (usando el sucesor mínimo).
- El código cuenta con documentación estructurada usando comentarios tipo @brief para facilitar el mantenimiento y comprensión.

3.1.7. Árbol Binario

La clase ArbolBinario implementa una estructura de árbol binario genérico que permite insertar, eliminar, buscar y recorrer nodos sin necesariamente seguir la lógica de un árbol binario de búsqueda (BST). Esta flexibilidad permite organizar los datos según criterios propios del usuario.

Estructura:

- El árbol está compuesto por nodos de tipo NodoBinario<T>, que almacenan un valor genérico y apuntadores a sus hijos izquierdo y derecho.
- La clase mantiene un único puntero a la raíz del árbol (raiz) y ofrece una interfaz para operar sobre él.
- Las operaciones básicas como inserción y recorrido se delegan, en su mayoría, a la clase NodoBinario.

Métodos principales:

- **insertar(val)**: Inserta un valor en el árbol siguiendo una lógica definida en los nodos (usualmente estilo BST).
- **eliminar(val)**: Elimina el nodo que contiene el valor val, reorganizando si es necesario.
- **buscar(val)**: Retorna un puntero al nodo que contiene el valor val, o nullptr si no se encuentra.
- **esVacio()**: Verifica si la raíz es nula.
- **datoRaiz()**: Retorna una referencia al valor almacenado en la raíz (se asume que el árbol no está vacío).
- **altura() y tamano()**: Calculan, respectivamente, la altura total y la cantidad de nodos del árbol.
- **Recorridos**:
    - preOrden(), inOrden(), posOrden(), nivelOrden(): imprimen los valores en el orden correspondiente, a partir de la raíz.

Mejoras implementadas:

- Se incluyen métodos auxiliares en los nodos que permiten realizar las operaciones recursivamente.
- La implementación del recorrido por niveles se apoya en una estructura tipo cola.
- Los métodos están documentados con comentarios descriptivos que explican el propósito y comportamiento esperado.

3.1.8. Árbol Expresión

La clase ArbolExpresion representa una estructura de árbol binario especializada en almacenar expresiones matemáticas en forma de árbol. Esta estructura permite construir el árbol a partir de expresiones en notación prefija o posfija (también llamadas notación polaca y polaca inversa, respectivamente), generar sus representaciones equivalentes en otras notaciones y evaluar el resultado de la expresión.

Estructura:

- El árbol se compone de nodos NodoExpresion, los cuales almacenan un carácter (ya sea un operando o un operador), una marca booleana que indica si es un operando, y punteros a los hijos izquierdo y derecho.
- La clase ArbolExpresion posee un único apuntador a la raíz del árbol, desde el cual se realizan todas las operaciones.

Principales métodos de ArbolExpresion:

- **llenarDesdePrefija(expresion)**: Construye el árbol a partir de una expresión en notación prefija.
- **llenarDesdePosfija(expresion)**: Construye el árbol a partir de una expresión en notación posfija.
- **obtenerPrefija(inicio)**, **obtenerInfija(inicio)**, **obtenerPosfija(inicio)**: Generan las representaciones prefija, infija y posfija de un subárbol.
- **evaluar(nodo)**: Evalúa recursivamente la expresión contenida en el subárbol con raíz en nodo.
- **siOperando(char)**: Verifica si un carácter corresponde a un operando (dígito) o no.

Principales métodos de NodoExpresion:

- **Getters y Setters** para el dato, hijos izquierdo y derecho, y la marca de operando.
- Define un nodo que puede representar un número (hoja del árbol) o un operador (nodo interno).

Ejemplo de uso (main.cpp):

El archivo principal (main.cpp) crea árboles a partir de expresiones en prefija y posfija, muestra sus equivalentes en otras notaciones y evalúa el resultado numérico de dichas expresiones. Sirve como prueba funcional completa de la implementación.

3.2. TAD's (Tipos Abstractos de Datos)

3.2.1. TAD para Nodo y Árbol

TAD Nodo (Árbol Binario):
Datos Mínimos:
- dato: almacena el valor del nodo.
- izq, der: punteros a los hijos izquierdo y derecho.

Operaciones:
Nodo(), crea un nodo con dato vacío y punteros nulos.
Nodo(val), crea un nodo con el dato val.
obtenerDato(), retorna el dato almacenado.
fijarDato(val), asigna el dato val al nodo.
insertar(val), inserta un nuevo nodo en el subárbol de forma ordenada.
preOrden(), posOrden(), inOrden(), recorre el subárbol en el orden correspondiente.
altura(), retorna la altura del subárbol.
tamano(), retorna el número total de nodos en el subárbol.
buscar(val), retorna un apuntador al nodo con el dato val.

TAD Árbol (Árbol):
Datos Mínimos:
- raiz: señala el nodo que corresponde a la raíz.

Operaciones:
Arbol(), crea un árbol con raíz nula.
Arbol(val), crea un árbol con raíz de valor val.
~Arbol()
esVacio(), retorna verdadero si la raíz es nula.
obtenerRaiz(), retorna el dato de la raíz.
fijarRaiz(nraiz), asigna el nodo nraiz como raíz.
insertarNodo(val), inserta un nuevo dato en el árbol.
eliminarNodo(val), elimina el nodo con el dato val.
buscarNodo(val), retorna el apuntador al nodo con el dato val.
altura(), tamano(), preOrden(), posOrden(), inOrden(), nivelOrden().

3.2.2. TAD para Nodo y QuadTree

TAD Nodo (QuadTree):
Datos Mínimos:
- dato: almacena un par (x, y).
- NW, NE, SW, SE: punteros a los hijos correspondientes a cada cuadrante.

Operaciones:
Nodo(), crea un nodo con punteros nulos.
Nodo(val), crea un nodo con el par de valores val.
obtenerDato(), retorna el par almacenado.
fijarDato(val), asigna el par val al nodo.

insertar(val), inserta el nuevo par en el cuadrante adecuado.
preOrden(), posOrden(), recorre el subárbol en el orden correspondiente.
altura(), retorna la altura del subárbol.
tamano(), retorna el total de nodos en el subárbol.
buscar(val), retorna el apuntador al nodo con el par val.

TAD QuadTree:
Datos Mínimos:
- raiz: señala el nodo raíz, que almacena un par (x, y).

Operaciones:
QuadTree(), crea un QuadTree con raíz nula.
QuadTree(val), crea un QuadTree con raíz de valor val.
~QuadTree()
esVacio(), retorna verdadero si la raíz es nula.
obtenerRaiz(), retorna el nodo o dato de la raíz.
fijarRaiz(nodo), asigna el nodo como raíz.
insertar(val), inserta un par en el QuadTree según la posición (cuadrante).
buscar(val), retorna el nodo que contiene el par val.
altura(), tamano(), preOrden(), posOrden().

3.2.3. TAD para Nodo y KD-Tree

TAD Nodo (KD-Tree):
Datos Mínimos:
- dato: almacena un punto (por ejemplo, (x, y)).
- izquierdo, derecho: punteros a los hijos.

Operaciones:
Nodo(), crea un nodo con punteros nulos.
Nodo(val), crea un nodo con el punto val.
obtenerDato(), retorna el punto almacenado.
fijarDato(val), asigna el punto val al nodo.
insertar(val, profundidad), inserta el punto val en el subárbol, alternando la dimensión según la profundidad.
preOrden(), posOrden(), recorre el subárbol.
altura(), tamano(), retorna las propiedades del subárbol.
buscar(val, profundidad), retorna el nodo con el punto val.

TAD KD-Tree:
Datos Mínimos:
- raiz: señala el nodo raíz, que almacena un punto (x, y).

Operaciones:
KDTree(), crea un KD-Tree con raíz nula.
KDTree(val), crea un KD-Tree con raíz de valor val.
~KDTree()

esVacio(), retorna verdadero si la raíz es nula.
obtenerRaiz(), retorna el dato de la raíz.
insertar(val), inserta un punto en el KD-Tree (con profundidad inicial 0).
buscar(val), retorna el nodo que contiene el punto val (usando profundidad).
altura(), tamano(), preOrden(), posOrden().

3.2.4. TAD para RBNode y Árbol Rojo-Negro

TAD RBNode (Árbol Rojo-Negro):
Datos Mínimos:
- data: almacena el dato del nodo.
- color: almacena el color (RED o BLACK).
- parent, left, right: punteros al padre y a los hijos.

Operaciones:
RBNode(data), crea un nodo con el dato dado y color RED por defecto.
obtenerDato(), retorna el dato.
fijarDato(data), asigna el dato al nodo.
(Las operaciones de recorrido, altura, etc., se delegan en el árbol).

TAD Árbol Rojo-Negro:
Datos Mínimos:
- raiz: señala el nodo que corresponde a la raíz; se utiliza un nodo centinela NIL para representar hojas.

Operaciones:
RBTree(), crea un árbol rojo-negro con la raíz inicializada a NIL.
RBTree(val), crea un árbol con raíz de valor val.
~RBTree()
esVacio(), retorna verdadero si la raíz es NIL.
obtenerRaiz(), retorna el nodo o dato de la raíz.
insertar(key), inserta un nuevo dato en el árbol rojo-negro, realizando las rotaciones y balanceos necesarios (leftRotate, rightRotate, insertFixup) para mantener las propiedades.
eliminar(key), (si se implementa) elimina el nodo con el dato key y retorna verdadero si se elimina.
buscar(key), retorna el nodo que contiene el dato key.
altura(), tamano(), inOrden(), recorre el árbol mostrando el dato y el color de cada nodo.

3.2.5. TAD para Arbol Binario AVL

TAD Nodo (Árbol AVL)
Datos Mínimos:
- dato: valor almacenado por el nodo.
- hijoIzq, hijoDer: punteros a los hijos izquierdo y derecho del nodo.

Operaciones:

- NodoBinarioAVL(): crea un nodo vacío con punteros nulos.
- NodoBinarioAVL(val): crea un nodo con el valor val.
- getDato(): retorna el valor almacenado en el nodo.
- setDato(val): asigna el valor val al nodo.
- getHijoIzq(), getHijoDer(): retornan los punteros a los hijos.
- setHijoIzq(nodo), setHijoDer(nodo): asignan hijos izquierdo o derecho.

TAD Árbol (Árbol AVL)
Datos Mínimos:

- raiz: apunta al nodo raíz del árbol.

Operaciones:

- ArbolBinarioAVL(): crea un árbol con raíz nula.
- ~ArbolBinarioAVL(): destructor del árbol.
- esVacio(): retorna verdadero si la raíz es nula.
- getRaiz(), setRaiz(nodo): permite acceder o modificar la raíz.
- insertar(val): inserta el valor val y balancea si es necesario.
- eliminar(val): elimina el nodo con valor val y reequilibra el árbol.
- buscar(val): retorna verdadero si el valor se encuentra en el árbol.
- altura(nodo): retorna la altura del subárbol con raíz en nodo.
- tamano(nodo): retorna la cantidad de nodos del subárbol.
- **Rotaciones**:
  - giroDerecha, giroIzquierda: rotaciones simples.
  - giroIzquierdaDerecha, giroDerechaIzquierda: rotaciones dobles.
- **Recorridos**:
  - preOrden(nodo), inOrden(nodo), posOrden(nodo), nivelOrden(nodo).

3.2.6. TADs Arbol Binario ordenado

TAD Nodo (NodoBinario)
Datos Mínimos:

- dato: almacena el contenido del nodo (tipo genérico T).
- izq, der: apuntadores a los hijos izquierdo y derecho del nodo.

Comportamiento:

- NodoBinario(): constructor que inicializa los hijos como nulos.
- NodoBinario(dato): constructor que asigna un dato al nodo.
- obtenerDato(): retorna el dato almacenado.
- fijarDato(val): modifica el valor almacenado en el nodo.
- obtenerHijoIzq(), obtenerHijoDer(): retornan los punteros a los hijos.
- fijarHijoIzq(izq), fijarHijoDer(der): asignan punteros a los hijos izquierdo y derecho respectivamente.

TAD Árbol (ArbolBinario)
Datos Mínimos:

- raiz: puntero al nodo raíz del árbol.

Comportamiento:

- ArbolBinario(): constructor que crea un árbol con raíz nula.
- getRaiz(): retorna el nodo correspondiente a la raíz del árbol.
- esVacio(): retorna true si el árbol no tiene raíz.
- datoRaiz(): retorna el dato contenido en la raíz (asume que el árbol no está vacío).
- altura(): retorna la altura total del árbol.
- tamano(): retorna el número total de nodos del árbol.
- insertar(valor): inserta un valor nuevo en el árbol, manteniendo el orden BST.
- altura(subarbol): retorna la altura de un subárbol específico.
- tamano(subarbol): retorna la cantidad de nodos en un subárbol.
- insertar(valor, subarbol): inserta un valor a partir de un nodo dado.
- eliminar(valor): busca y elimina el nodo con el valor dado.
- buscar(valor): retorna true si el valor está presente en el árbol.
- preOrden(subarbol): imprime en preorden a partir de un nodo dado.
- inOrden(subarbol): imprime en inorden a partir de un nodo dado.
- posOrden(subarbol): imprime en posorden desde un nodo específico.
- nivelOrden(subarbol): imprime por niveles desde un nodo dado.
- preOrden(), inOrden(), posOrden(), nivelOrden(): versiones que inician desde la raíz del árbol.

3.2.7. TADs Arbol Binario

TAD Nodo (NodoBinario)
Datos mínimos:
- dato: contiene el valor del nodo.
- hijoIzq, hijoDer: apuntadores a los hijos izquierdo y derecho del nodo.

Comportamiento:
- NodoBinario(): constructor que inicializa los punteros a nulo.
- NodoBinario(dato): constructor que inicializa el nodo con el dato proporcionado.
- obtenerDato(): retorna el dato del nodo.
- fijarDato(val): asigna un nuevo valor al nodo.
- obtenerHijoIzq(), obtenerHijoDer(): devuelven los punteros a los hijos.
- fijarHijoIzq(izq), fijarHijoDer(der): asignan los punteros a los hijos.
- altura(): retorna la altura del subárbol que cuelga del nodo.
- tamano(): retorna el número total de nodos desde este nodo.
- insertar(val): inserta un valor en el subárbol desde este nodo.
- buscar(val): retorna el nodo que contiene val o nullptr si no existe.
- **Recorridos**:
  - preOrden(): imprime nodo → hijo izq → hijo der.
  - inOrden(): imprime hijo izq → nodo → hijo der.
  - posOrden(): imprime hijo izq → hijo der → nodo.
  - nivelOrden(): imprime nivel por nivel (BFS).
- extremo_izq(), extremo_der(): retornan el nodo más a la izquierda o derecha del subárbol.

TAD Árbol (ArbolBinario)
Datos mínimos:

- raiz: apunta al nodo raíz del árbol.

Comportamiento:

- ArbolBinario(): constructor que inicializa el árbol vacío.
- esVacio(): retorna true si la raíz es nullptr.
- datoRaiz(): retorna el valor contenido en la raíz.
- altura(): retorna la altura del árbol.
- tamano(): retorna la cantidad de nodos del árbol.
- insertar(val): inserta un valor en el árbol (delegando en el nodo).
- eliminar(val): elimina un nodo por valor.
- buscar(val): retorna un puntero al nodo que contiene el valor (si existe).
- **Recorridos**:
    - preOrden(), inOrden(), posOrden(), nivelOrden(): imprimen los valores del árbol desde la raíz.

3.2.8. TADs Arbol Expresión

TAD NodoExp (NodoExpresion)

Datos mínimos:

- data: carácter que representa un dígito u operador.
- left, right: apuntadores a los nodos hijo izquierdo y derecho.
- op: valor booleano que indica si el nodo representa un operador (true) o un operando (false).

Comportamiento:

- NodoExp(dato): crea un nodo e inicializa su contenido.
- NodoExp(): crea un nodo vacío.
- getData(): retorna el dato almacenado.
- setData(val): asigna un nuevo dato al nodo.
- getLeft(), getRight(): retornan los hijos izquierdo y derecho.
- setLeft(nodo), setRight(nodo): asignan los hijos izquierdo y derecho.
- getOp(): retorna si el nodo es un operador.
- setOp(bool): establece si el nodo representa un operador.

TAD ArbolExp (ArbolExpresion)

Datos mínimos:

- raiz: apuntador al nodo raíz del árbol.
- operadores: conjunto de símbolos válidos como operadores (e.g., +, -, *, /).

Comportamiento:

- ArbolExpresion(): constructor que crea un árbol vacío.
- llenarDesdePrefija(expresion): construye el árbol desde una cadena en notación prefija.
- llenarDesdePosfija(expresion): construye el árbol desde una cadena en notación posfija.
- obtenerPrefija(): retorna la representación de la expresión en notación prefija.
- obtenerInfija(): retorna la representación de la expresión en notación infija.

- obtenerPosfija(): retorna la representación de la expresión en notación posfija.
- Prefija(subarbol), Infija(subarbol), Posfija(subarbol): versiones que operan sobre un subárbol dado.
- evaluar(): calcula el resultado final de la expresión completa.
- eval(nodo): evalúa el subárbol con raíz en nodo.
- esOp(op): verifica si un símbolo pertenece al conjunto de operadores válidos.
- llenarDesdePrefijaa(q, pos, actual), llenarDesdePosfijaa(q, pos, actual): versiones alternativas para construir el árbol desde un vector de tokens.
- tokenizar(s, q): divide la cadena en tokens para facilitar el análisis.

3.3. Pruebas

Para cada estructura se desarrollaron programas principales (mains) que demuestran:

- La inserción de datos o puntos en el árbol.
- Los recorridos (preorden, postorden, inOrden o nivelorden) que permiten verificar el orden, la correcta partición o balanceo.
- La búsqueda de nodos específicos.
- El cálculo y la visualización de propiedades como la altura y el tamano.

Los resultados obtenidos confirman que, tras las correcciones y mejoras implementadas, todas las estructuras funcionan de manera robusta y cumplen con los objetivos de eficiencia y organización de datos.

Conclusiones

- Se detectaron y corrigieron errores en la lógica de cada estructura: en el QuadTree y KD-Tree se ajustaron los métodos de cálculo (altura y tamaño) y la alternancia de dimensiones; en el Árbol Rojo-Negro se implementó correctamente el balanceo mediante rotaciones y fixup; en el Árbol Binario se gestionó la inserción y el manejo de árbol vacío.

- La documentación detallada y los comentarios en cada archivo facilitan la comprensión y el mantenimiento del código.

- Las pruebas realizadas demuestran que, con las correcciones propuestas, las implementaciones son robustas y adecuadas para aplicaciones que requieren la organización y búsqueda eficiente de datos.

- En el Árbol Binario General, se validó el uso de métodos recursivos para el cálculo de altura, tamaño, recorridos y búsqueda, demostrando que puede ser adaptado a diversas lógicas más allá del orden binario clásico.

- En el Árbol Binario Ordenado, se aseguraron las propiedades de los árboles de búsqueda, y se comprobó el correcto funcionamiento de inserciones, búsquedas y eliminaciones siguiendo el orden BST.

- El Árbol AVL incorpora mecanismos automáticos de balanceo mediante rotaciones simples y dobles, logrando una estructura equilibrada que garantiza tiempos de operación logarítmicos.

- El Árbol de Expresión permite representar y evaluar expresiones aritméticas de forma estructurada, transformando notaciones prefija y posfija en árboles que luego pueden recorrerse y evaluarse de forma eficiente, siendo útil para la interpretación de lenguajes matemáticos o expresiones algebraicas.

Referencias

- Implementaciones originales proporcionadas en el archivo .zip.
- Material bibliográfico y documentación sobre estructuras de datos en C++.
- “Data Structures and Algorithms in C++” de Michael T. Goodrich, Roberto Tamassia, y David M. Mount (si aplica).
- Documentación y artículos especializados sobre Árboles Rojo-Negros.