Taller 03 Arboles

Estructura de datos

Juan Sebastián Rodríguez Pabon

Juian Pérez Gomes

David Rodríguez Jurado

Daniel Galviz

Dirigido a John Corredor

4 de octubre de 2024

Bogotá D.C

Pontificia Universidad JAVERIANA Bogotá

Introducción

Las estructuras de datos son esenciales en sistemas de información, ya que permiten gestionar y manipular elementos mediante operaciones de inserción, búsqueda y eliminación. Entre estas estructuras, los árboles binarios de búsqueda destacan por su capacidad para organizar datos de manera jerárquica y eficiente. Los árboles son estructuras de datos no lineales compuestas de varios nodos, organizados de tal forma que permiten obtener información de manera rápida y ordenada.

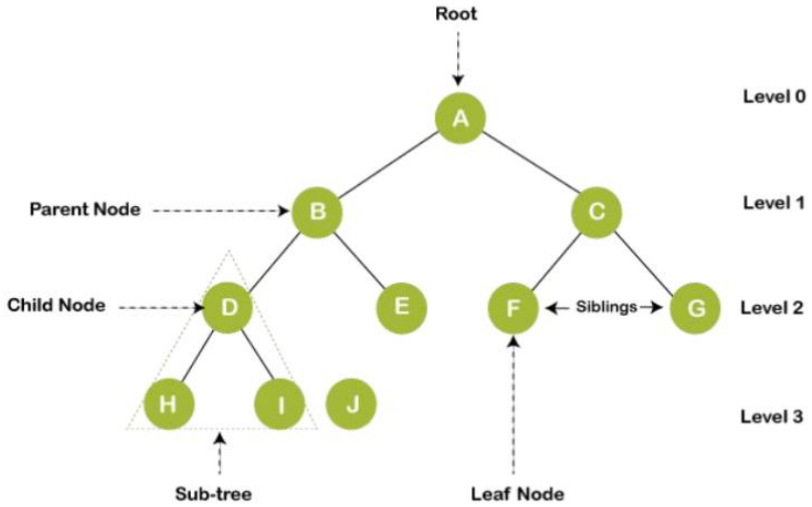
En el siguiente documento se analizarán distintos tipos de árboles binarios de búsqueda, explicando sus funciones específicas y cómo se implementan en diversos contextos.

Implementación

Un árbol comienza con un nodo raíz, el cual puede tener varios nodos hijos en el nivel siguiente. Los árboles crecen en niveles, y cada nodo puede tener hijos en función del tipo de árbol. Los árboles en sí abarcan una amplia variedad de estructuras y se dividen en múltiples categorías, las cuales se detallarán a continuación.

A continuación, se muestra un ejemplo básico de un árbol que ilustra sus componentes fundamentales. En el ejemplo, se destacan:

Raíz (Root): El nodo inicial o base.
Nodos (Node): Los elementos individuales del árbol.
Nodos hijos (Child Node): Son los nodos que dependen jerárquicamente de otros nodos.
Subárbol (Sub-tree): Secciones del árbol que constituyen en sí mismos estructuras jerárquicas.
Niveles: Las distintas capas de nodos que se extienden a partir de la raíz.



Árbol Binario

¿Qué es?
El árbol binario está formado por Nodos, necesariamente 2, donde cada nodo tiene una clave que indica su valor, el valor de los nodos del subárbol izquierdo es menor que el nodo raíz y el valor del subárbol derecho es mayor que el nodo raíz en un árbol binario de búsqueda.

Características
- Compuesto por dos números, es decir, dos nodos.
- Cada Nodo del árbol puede tener únicamente máximo 2 hijos.
- Ubicado de menor a mayor entre nodos.

¿Para que funciona?
El objetivo del árbol binario es organizar los datos de menor a mayor mediante cada nodo, siendo así una estructura jerárquica. Esto permite realizar operaciones de búsqueda, inserción y eliminación de manera eficiente y ordenada.

Recorridos
InOrder: Se pasa del subárbol izquierdo a la raíz y luego al subárbol derecho. PostOrder: Se pasa del subárbol izquierdo al subárbol derecho y luego a la raíz.
PreOrder: Desde la raíz al subárbol izquierdo y luego al subárbol derecho
NivelOrder: Muestra los valores de los nodos del árbol en el orden de nivel (nivel por nivel, de arriba hacia abajo).

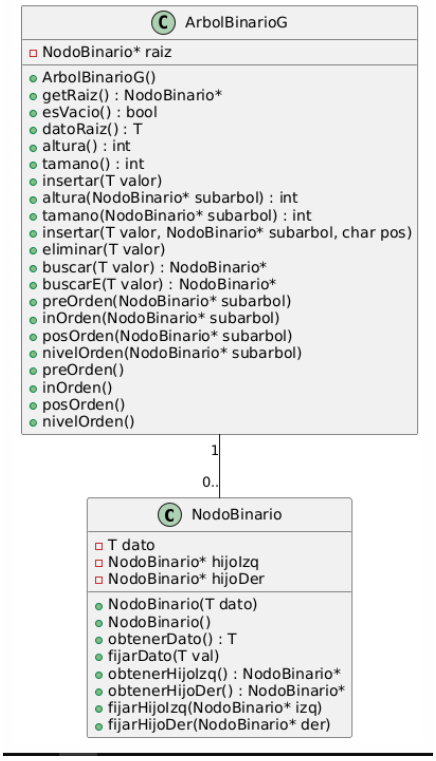
OperacionesBúsqueda: Se recorre nada nodo del árbol hasta finalmente encontrar el dato que se necesita hallar.
Inserción: Los nodos se pueden agregar en cualquier posición vacía del árbol.
Eliminación: Los nodos que se necesitan eliminar para agregar un dato nuevo en el árbol.

NOTA: En un árbol binario como solamente se tienen 2 hijos, al insertar o eliminar algún nodo quedaría desbalanceado, por esta razón se tiene que balancear el árbol según re requiera. Además el concepto de balanceo aplica únicamente para ciertos arboles que buscan mantener el equilibrio de este mismo.

Una vez ya se ha explicado el comportamiento y las funcionalidades principales de un árbol binario, se hará la implementación de este árbol Binario.

¿Cómo lo implementamos?

Se proporciona una interfaz de menú que permite al usuario realizar varias operaciones en el árbol binario (Como ya se mencionaron anteriormente)



Archivos de implementación

- NodoBinario.h: Define la clase NodoBinario, representa un nodo en el árbol binario. Cada nodo tiene un valor (dato) y dos punteros a sus hijos izquierdo y derecho (hijoIzq y hijoDer). La clase incluye métodos para obtener y establecer el valor y los hijos, calcular la altura y el tamaño del subárbol, insertar nuevos nodos, buscar nodos y realizar recorridos del árbol en diferentes órdenes.
- NodoBinario.hxx: Implementa los métodos definidos en NodoBinario.h. Aquí se define cómo se realizan las operaciones de inserción, búsqueda, y los recorridos (preorden, inorden, postorden y por nivel).
- ArbolBinario.h: Define la clase ArbolBinario, que representa el árbol binario en sí. La clase tiene un puntero a la raíz del árbol (raiz) y métodos para verificar si el árbol está vacío, obtener el dato de la raíz, calcular la altura y el tamaño del árbol, insertar y eliminar nodos, y realizar recorridos en diferentes órdenes.
- ArbolBinario.hxx: Implementa los métodos definidos en ArbolBinario.h. Aquí se define cómo se realizan las operaciones de inserción, eliminación y los recorridos del árbol.
- main.cpp: Contiene el punto de entrada del programa y proporciona un menú interactivo para el usuario. Permite al usuario insertar nodos en el árbol y realizar diferentes tipos de recorridos.

Pontificia Universidad JAVERIANA Bogotá

Funcionamiento

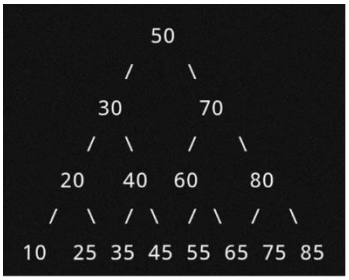
- Inicialización del Árbol: Al iniciar el programa, se crea un objeto de la clase ArbolBinario llamado arbolito. Este árbol se inicia vacío cada vez que el programa se ejecuta el programa.
- Menú Interactivo: El programa presenta un menú con las siguientes opciones:
- Insertar un nodo: Permite al usuario ingresar un valor para insertar en el árbol. El valor se inserta siguiendo las reglas del árbol binario de búsqueda.
- Recorrido en Preorden: Muestra los valores de los nodos del árbol en el orden de preorden (nodo, hijo izquierdo, hijo derecho).
- Recorrido en Inorden: Muestra los valores de los nodos del árbol en el orden de inorden (hijo izquierdo, nodo, hijo derecho).
- Recorrido en Postorden: Muestra los valores de los nodos del árbol en el orden de postorden (hijo izquierdo, hijo derecho, nodo).
- Recorrido en NivelOrden: Muestra los valores de los nodos del árbol en el orden de nivel (nivel por nivel, de arriba hacia abajo).

Operaciones en el Árbol

- Insertar: Se agrega un nuevo nodo al árbol. Si el árbol está vacío, el nuevo nodo se convierte en la raíz. De lo contrario, el nodo se inserta en la posición correcta según las reglas del árbol binario de búsqueda.
- Recorridos: Dependiendo de la opción seleccionada, el programa realiza un recorrido del árbol y muestra los valores de los nodos en el orden especificado.
- Salir del Programa: La opción 0 permite al usuario salir del programa.

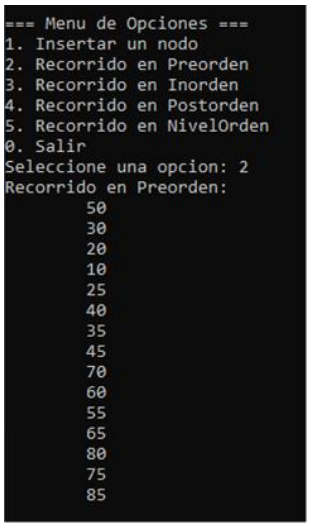
Plan de pruebas

Se ingresará un valor para cada nodo, en este caso los números ingresados serán: 50, 30, 70, 20, 40, 60, 80, 10, 25, 35, 45, 55, 65, 75 y 85

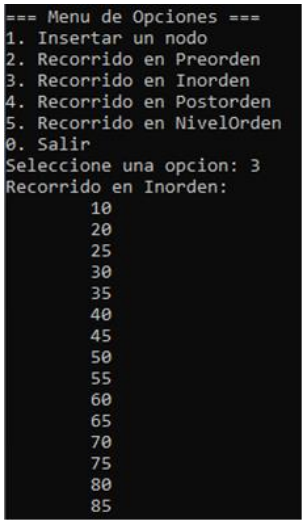


Como se mencionó anteriormente de acuerdo con los números, se puede evidenciar que siempre el valor del nodo izquierdo es menor a la raíz y el valor del nodo derecho es mayor que la raíz, propiedad esencial de este árbol.

Se puede observar que se tiene un menú en el cual el usuario puede seleccionar que tipo de recorrido desea hacer, en este caso se realizara el recorrido de PreOrder, el cual tiene un valor de 2 en el menú. El recorrido de PreOrder, es desde la raíz al subárbol izquierdo y luego al subárbol derecho.

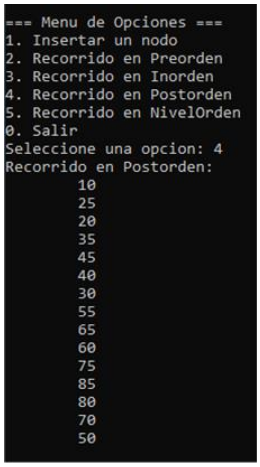


Entorno a este recorrido, se buscará realizar un recorrido en InOrder, el cual en el menú tiene un valor de 3.
El recorrido InOrder se pasa del subárbol izquierdo a la raíz y luego al subárbol derecho y tiene la siguiente estructura: 10, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75, 80, 85.

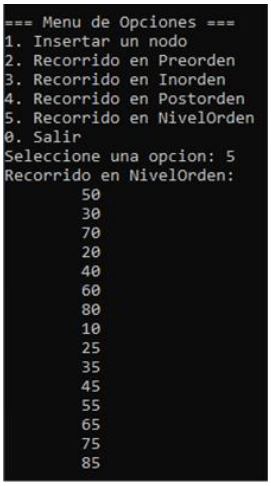


Entorno a este recorrido, se buscará realizar un recorrido en PostOrder, el cual en el menú tiene un valor de 4.

El recorrido PostOder, se pasa del subárbol izquierdo al subárbol derecho y luego a la raíz y tiene la siguiente estructura: 10, 25, 20, 35, 45, 40, 30, 55, 65, 60, 75, 85, 80, 70, 50.



Entorno a este recorrido, se buscará realizar un recorrido en NivelOrder, el cual en el menú tiene un valor de 5.
El recorrido NivelOder muestra los valores de los nodos del árbol en el orden de nivel (nivel por nivel, de arriba hacia abajo). Tiene la siguiente estructura: 50, 30, 70, 20, 40, 60, 80, 10, 25, 35, 45, 55, 65, 75, 85.



Pontificia Universidad JAVERIANA Bogotá

TADS – Árbol binario

TAD - NodoBinario

Objetivo
Representa un nodo en el árbol binario, con un valor de dato y enlaces a sus hijos izquierdo y derecho.

Nombre NodoBinario

Dominio
Elementos de tipo T (tipo genérico), punteros a NodoBinario para el hijo izquierdo y el hijo derecho.

Operaciones

1. **obtenerDato(): T**

   - **Descripción** Devuelve el valor almacenado en el nodo.
   - **Entrada** Ninguna.
   - **Salida** Dato del nodo.

2. **fijarDato(T dato): void**

   - **Descripción** Establece el valor del nodo.
   - **Entrada** Valor dato de tipo T**.**
   - **Salida** Ninguna.

3. **obtenerHijoIzq(): NodoBinario***

   - **Descripción** Devuelve el puntero al hijo izquierdo del nodo.
   - **Entrada** Ninguna.
   - **Salida** Puntero al nodo hijo izquierdo.

4. **fijarHijoIzq(NodoBinario izq): void***
   - **Descripción** Establece el hijo izquierdo del nodo.
   - **Entrada** Puntero izq a un NodoBinario.
   - **Salida** Ninguna.

5. **obtenerHijoDer(): NodoBinario***
   - **Descripción** Devuelve el puntero al hijo derecho del nodo.
   - **Entrada** Ninguna.
   - **Salida** Puntero al nodo hijo derecho.

6. **fijarHijoDer(NodoBinario der): void***
   - **Descripción** Establece el hijo derecho del nodo.
   - **Entrada** Puntero der a un NodoBinario.
   - **Salida**

Pontificia Universidad JAVERIANA Bogotá

TAD – ArbolBinarioG

Objetivo
Representar un árbol binario con funciones de inserción, eliminación y varios tipos de recorridos.

Nombre ArbolBinarioG

Dominio Nodo raíz de tipo NodoBinario, métodos de recorrido y modificación del árbol.

Operaciones

1. **getRaiz(): NodoBinario***
   - **Descripción** Devuelve la raíz del árbol.
   - **Entrada** Ninguna.
   - **Salida** Puntero a la raíz del árbol.

2. **esVacio(): bool**
   - **Descripción** Verifica si el árbol está vacío.
   - **Entrada** Ninguna.
   - **Salida** Retorna true si el árbol está vacío; de lo contrario, false.

3. **datoRaiz(): T**
   - **Descripción** Obtiene el dato de la raíz.
   - **Entrada** Ninguna.
   - **Salida** Dato de tipo T en la raíz del árbol.

4. **altura(): int**
   - **Descripción** Calcula la altura del árbol.
   - **Entrada** Ninguna.
   - **Salida** Altura del árbol en forma de entero.

5. **tamano(): int**
   - **Descripción** Calcula el tamaño (cantidad de nodos) del árbol.
   - **Entrada** Ninguna.
   - **Salida** Tamaño del árbol.

6. **insertar(T valor) void**
   - **Descripción** Inserta un nodo con el valor dado en la posición correcta según las reglas de un árbol binario de búsqueda.
   - **Entrada** Valor valor de tipo T.
   - **Salida** Ninguna.

7. **eliminar(T valor): void**
   - **Descripción** Elimina el nodo con el valor dado, ajustando el árbol según sea necesario.
   - **Entrada** Valor valor de tipo T que se desea eliminar.
   - **Salida** Ninguna.

8. **buscar(T valor): NodoBinario***
   - **Descripción** Busca un nodo con el valor especificado en el árbol.
   - **Entrada** Valor valor de tipo T.
   - **Salida** Puntero al nodo si se encuentra el valor; de lo contrario, nullptr.

9. **preOrden(): void**
   - **Descripción** Realiza el recorrido en PreOrden (raíz, subárbol izquierdo, subárbol derecho).
   - **Entrada** Ninguna.
   - **Salida** Muestra los valores de los nodos en el orden de PreOrden.

10. **inOrden(): void**
    - **Descripción** Realiza el recorrido en InOrden (subárbol izquierdo, raíz, subárbol derecho).
    - **Entrada** Ninguna.
    - **Salida** Muestra los valores de los nodos en el orden de InOrden.

11. **posOrden(): void**
    - **Descripción** Realiza el recorrido en PosOrden (subárbol izquierdo, subárbol derecho, raíz).
    - **Entrada** Ninguna.
    - **Salida** Muestra los valores de los nodos en el orden de PosOrden.

12. **nivelOrden(): void**
    - **Descripción** Realiza el recorrido en NivelOrden, mostrando los valores de los nodos en el orden de nivel, de arriba hacia abajo.
    - **Entrada** Ninguna.
    - **Salida** Muestra los valores de los nodos en el orden de NivelOrden.

TAD – MenuArbol

Objetivo
Proporcionar una interfaz de usuario que permita realizar operaciones en el árbol binario.

Nombre MenuArbol

Dominio Opciones de interacción con el usuario para insertar nodos y realizar recorridos en el árbol.

Operaciones

1. **mostrarMenu(): void**
   - **Descripción** Muestra el menú de opciones al usuario y permite seleccionar la operación deseada.
   - **Entrada** Ninguna.
   - **Salida** Ninguna.

2. **insertarNodo(T valor): void**
   - **Descripción** Permite al usuario insertar un valor en el árbol.
   - **Entrada** Valor valor de tipo T.
   - **Salida** Ninguna.

3. **realizarRecorrido(int opcion): void**
   - **Descripción** Ejecuta el recorrido seleccionado por el usuario (PreOrden, InOrden, PosOrden, o NivelOrden) y muestra los resultados.
   - **Entrada** Opción opcion que representa el tipo de recorrido.
   - **Salida** Ninguna.

4. **salir(): void**
   - **Descripción** Sale del programa.
   - **Entrada** Ninguna.
   - **Salida** Ninguna.

Conclusión

Al realizar la corrección de algunos errores del codigo Fuente, se pudo evidenciar que un arbol binario puede realizar funciones y recorridos de manera efectiva segun se requiera, en este caso los resultados fueron los esperados. Ademas se implemento una funcion main con interfaz para el usuario para que el programa sea usable, coherente y facil de entender con el fin de que algún usuario pueda jugar y apropiarse del concepto de arbol binario y todos sus componentes.

Árbol AVL

¿Qué es?

Es un tipo de árbol binario de búsqueda auto balanceado, en el cual cada nodo tiene un valor y se garantiza operaciones eficientes para buscar, insertar y eliminar. El balanceo del árbol es el pilar fundamental de este tipo de árbol y existen diferentes tipos de balanceos.

Características

- Árbol auto-balanceado O(logN).
- Inserciones y eliminaciones son siempre O (log N).
- La altura no debe ser mayor a 1 de lo contrario estaría desbalanceado.

¿Para que funciona?

El árbol AVL es esencial para realiza operaciones de buscar, insertar y eliminar nodos en el árbol, mediante diferentes tipos de rotaciones para mantener el equilibrio del árbol al momento de insertar o eliminar un nodo. El factor de balance funciona: Altura (subárbol izquierdo – subárbol derecho)

Rotaciones para equilibrar

Rotación derecha

Esta rotación se realiza cuando hay un desbalance en la parte izquierda del árbol debido a una inserción en el subárbol izquierdo del hijo izquierdo del nodo desbalanceado (caso izquierda-izquierda).
En este caso, el hijo izquierdo del nodo en desequilibrio pasa a ser el nuevo nodo padre, y el nodo desequilibrado se convierte en el hijo derecho de este nuevo padre.

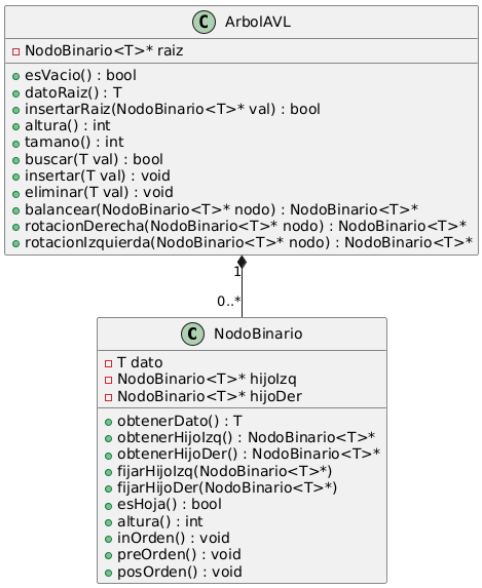
Rotación izquierda
Esta rotación ocurre cuando hay un desbalance en la parte derecha del árbol, específicamente cuando la inserción se realiza en el subárbol derecho del hijo derecho del nodo desbalanceado (caso derecha-derecha).
Aquí, el hijo derecho del nodo desequilibrado se convierte en el nuevo padre, y el nodo desbalanceado pasa a ser el hijo izquierdo de este nuevo padre.

Rotación izquierda – derecha (Rotación doble derecha)
Esta rotación se aplica cuando el desbalance ocurre en el subárbol derecho del nodo izquierdo del nodo desbalanceado (caso izquierda-derecha).
Primero, se realiza una rotación izquierda en el subárbol izquierdo, transformando el caso en un desbalance izquierda-izquierda. Luego, se hace una rotación derecha en el nodo desbalanceado para equilibrar el árbol.

Rotación derecha – izquierda (Rotación doble izquierda)
Esta rotación se usa cuando hay un desbalance en el subárbol izquierdo del hijo derecho del nodo desbalanceado (caso derecha-izquierda).
Primero, se hace una rotación derecha en el subárbol derecho, convirtiendo el caso en un desbalance derecha-derecha. Luego, se realiza una rotación izquierda en el nodo desbalanceado para equilibrar el árbol.

NOTA: SE DEBEN DE HACER ROTACIONES HASTA FINALMENTE PODER BALANCEAR EL ARBOL AVL

¿Cómo lo implementamos?



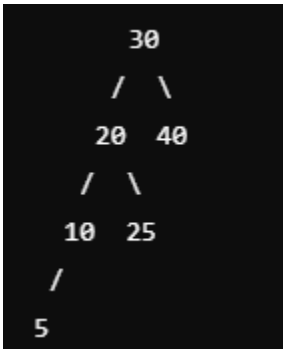
Clase NodoBinario
Representa un nodo en el árbol AVL.
Contiene los datos **(dato),** punteros al hijo izquierdo **(hijoIzq)** y al hijo derecho **(hijoDer).**
Métodos para obtener y fijar los nodos hijos (**obtenerHijoIzq, fijarHijoIzq, etc**.), verificar si un nodo es una hoja **(esHoja),** y calcular la altura del nodo **(altura).**
Métodos de recorrido como **inOrden, preOrden, y posOrden.**

Clase ArbolAVL
Representa el árbol AVL completo.
Operaciones principales: Inserción – Eliminación – Balanceo - Rotaciones
Inserción (insertarRec, insertar): Se asegura de que después de la inserción, el árbol permanezca balanceado.
Eliminación (eliminarRec, eliminar): Elimina un nodo manteniendo las propiedades AVL.
Balanceo (balanceo, balancear): Garantiza que el árbol se mantenga balanceado calculando el factor de balanceo y aplicando las rotaciones adecuadas.
Rotaciones (rotacionIzquierda, rotacionDerecha): Son las operaciones fundamentales para mantener el balance del árbol tras inserciones y eliminaciones.
Métodos de recorrido del árbol (preOrden, inOrden, posOrden, nivelOrden).
Factor de Balanceo: La función balanceo calcula el factor de balanceo de un nodo, que es la diferencia entre la altura de los subárboles izquierdo y derecho. Si esta diferencia excede uno, el árbol se reequilibra.
Como se menciono anteriormente, las rotaciones se usan para balancear arboles siendo que el nivel de los nodos del árbol no sea mayor a 1, entorno a esto se realizaron rotaciones para comprobar su eficiencia en arboles AVL.

¿Cómo se verifica que rotación usar?
Como se resaltó anteriormente, el nivel de los nodos del árbol no debe ser superior a 1, para esto se debe de mirar la altura del subárbol izquierdo y del subárbol derecho para verificar si no es superior a 1. Se evidencia que la altura del subárbol izquierdo es 3 porque se tiene el siguiente camino, 30 → 20 → 10 → 5, y la altura del subárbol derecho es 1 porque solo tiene un nodo (40).
Como se debe de calcular la diferencia, se debe de restar la altura del subárbol izquierdo con el subárbol derecho, es decir 3 - 1 = 2, lo cual significa que el árbol está desbalanceado en este punto.

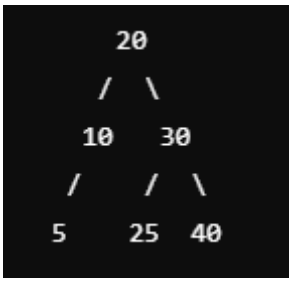
Se ingresarán valores de un árbol: 5, 10, 20, 25, 30 y 40 de tal manera que se evidencie que este desbalanceado en el subárbol izquierdo para poder realizar el balanceo.

Rotación derecha – izquierda **(Rotación doble izquierda)**



¿Cómo balancear el árbol por medio de Rotación derecha – izquierda (Rotación doble izquierda)?
El nodo 30 tiene un factor de balanceo de 2 porque la altura izquierda - altura derecha da un resultado de 2, lo que indica que el subárbol izquierdo tiene más altura que el subárbol derecho.
El nodo 20 tiene un factor de balanceo de 1 porque la diferencia entre las alturas de sus subárboles izquierdo y derecho.
Entonces la rotación que es necesaria de hacer es Rotación derecha – izquierda, como el árbol está desbalanceado hacia la izquierda, es decir el subárbol izquierdo es más alto, se necesita realizar una rotación derecha en el nodo 30 para que esta pueda ser balanceado.
En esta rotación el nodo 20 se convierte en la nueva raíz del subárbol, el nodo 30 se convierte en el hijo derecho del nodo 20 y el subárbol derecho de 20 (nodo 25) permanece como el hijo izquierdo de 30.

Realizando este procedimiento para balancear el árbol usando la **rotación derecha – izquierda (Rotación doble izquierda)** Se llegó al siguiente árbol balanceado.



Creación de TADS

TAD – NodoBinario

Objetivo
Representar un nodo en un árbol binario, el cual puede almacenar un valor genérico y referencias a sus hijos izquierdo y derecho.

Atributos

1. **dato: Tipo T**
   - Valor almacenado en el nodo.

2. **hijoIzq: NodoBinario<T>***
   - Referencia al nodo hijo izquierdo.

3. **hijoDer: NodoBinario<T>***
   - Referencia al nodo hijo derecho.

Operaciones

1. **NodoBinario()**
   - **Descripción** Constructor por defecto del nodo.
   - **Entrada** Ninguna.
   - **Salida** Crea un nodo con hijos izquierdo y derecho en nullptr.

2. **NodoBinario(T val)**
   - **Descripción** Constructor que inicializa el nodo con un valor específico.
   - **Entrada** Ninguna.
   - **Salida** Crea un nodo con el valor val, y los hijos en nullptr.

3. **~NodoBinario()**
   - **Descripción** Destructor que libera la memoria de los nodos hijos.
   - **Entrada** Ninguna**.**
   - **Salida** Libera la memoria de los nodos descendientes**.**

4. **obtenerDato()**
   - **Descripción** Devuelve el valor almacenado en el nodo.
   - **Entrada** El nodo debe tener un valor asignado.
   - **Salida** Retorna el valor dato.

5. **fijarDato(T val)**
   - **Descripción** Establece un nuevo valor en el nodo.
   - **Entrada** Ninguna.
   - **Salida** Asigna val a dato.

6. **obtenerHijoIzq()**
   - **Descripción** Devuelve el nodo hijo izquierdo.
   - **Entrada** Ninguna.
   - **Salida** Retorna hijoIzq.

7. **obtenerHijoDer()**
   - **Descripción** Devuelve el nodo hijo derecho.
   - **Entrada** Ninguna.
   - **Salida** Retorna hijoDer.

8. **fijarHijoIzq(NodoBinario<T> izq)***
   - **Descripción** Asigna un nodo hijo izquierdo.
   - **Entrada** Ninguna.
   - **Salida** Asigna izq a hijoIzq.

9. **fijarHijoDer(NodoBinario<T> der)***
   - **Descripción** Asigna un nodo hijo derecho.
   - **Entrada** Ninguna.
   - **Salida** Asigna der a hijoDer.

10. **esHoja()**
    - **Descripción** Verifica si el nodo es una hoja.
    - **Entrada** Ninguna.
    - **Salida** Retorna true si el nodo no tiene hijos, de lo contrario false**.**

11. **altura()**
    - **Descripción** Calcula la altura del subárbol a partir del nodo actual.
    - **Entrada** Ninguna**.**
    - **Salida** Retorna un valor entero que representa la altura.

12. **inOrden()**
    - **Descripción** Realiza el recorrido inorden del subárbol.
    - **Entrada** Ninguna.
    - **Salida** Imprime el valor de cada nodo en inorden.

13. **preOrden()**
    - **Descripción** Realiza el recorrido preorden del subárbol.
    - **Entrada** Ninguna.
    - **Salida** Imprime el valor de cada nodo en preorden.

14. **posOrden()**
    - **Descripción** Realiza el recorrido posorden del subárbol.
    - **Entrada** Ninguna.
    - **Salida** Imprime el valor de cada nodo en posorden.

15. **tamano()**
    - **Descripción** Calcula el tamaño del subárbol a partir del nodo actual.
    - **Entrada** Ninguna.
    - **Salida** Retorna un valor entero que representa el número total de nodos en el subárbol.

Pontificia Universidad JAVERIANA Bogotá

TAD - ArbolAVL

Nombre del TAD: ArbolAVL

Propósito

Representar un árbol AVL (árbol binario de búsqueda auto-balanceado) que almacena valores genéricos y asegura el balanceo a través de rotaciones.

Atributos

1. **raiz**: NodoBinario<T>*
   - Referencia al nodo raíz del árbol.

Operaciones

1. **ArbolAVL()**
   - **Descripción** Constructor que inicializa el árbol vacío.
   - **Entrada** Ninguna.
   - **Salida** El árbol se inicializa sin ningún nodo.
2. **~ArbolAVL()**
   - **Descripción** Destructor que libera la memoria del árbol.
   - **Entrada** Ninguna.
   - **Salida** Libera todos los nodos del árbol.
3. **esVacio()**
   - **Descripción** Verifica si el árbol está vacío.
   - **Entrada** Ninguna.
   - **Salida** Retorna true si raiz es nullptr.
4. **datoRaiz()**
   - **Descripción** Devuelve el valor de la raíz.
   - **Entrada** El árbol no debe estar vacío.
   - **Salida** Retorna el valor almacenado en raiz.
5. *insertarRaiz(NodoBinario<T> val)**
   - **Descripción** Inserta un nodo como raíz si el árbol está vacío.
   - **Entrada** El árbol debe estar vacío.
   - **Salida** Establece val como raiz del árbol.
6. **altura()**
   - **Descripción** Calcula la altura del árbol.
   - **Entrada** Ninguna.

- **Salida** Retorna la altura de raiz.

7. **tamano()**

   - **Descripción** Calcula el número de nodos en el árbol.
   - **Entrada** Ninguna.
   - **Salida** Retorna el tamaño del árbol.

8. **buscar(T val)**

   - **Descripción** Busca un valor en el árbol.
   - **Entrada** Ninguna.
   - **Salida** Retorna true si el valor está en el árbol, de lo contrario, false.

9. **preOrden()**

   - **Descripción** Realiza un recorrido en preorden del árbol.
   - **Entrada** Ninguna.
   - **Salida** Imprime los valores del árbol en preorden.

10. **inOrden()**

    - **Descripción** Realiza un recorrido en inorden del árbol.
    - **Entrada** Ninguna.
    - **Salida** Imprime los valores del árbol en inorden.

11. **posOrden()**

    - **Descripción** Realiza un recorrido en posorden del árbol.
    - **Entrada** Ninguna.
    - **Salida** Imprime los valores del árbol en posorden.

12. **nivelOrden()**

    - **Descripción** Realiza un recorrido por niveles del árbol.
    - **Entrada** Ninguna.
    - **Salida** Imprime los valores del árbol por niveles.

13. **balanceo(NodoBinario<T> nodo)***

    - **Descripción** Calcula el factor de balanceo de un nodo.
    - **Entrada** nodo no debe ser nullptr.
    - **Salida** Retorna el factor de balanceo.

14. **balancear(NodoBinario<T> nodo)***

    - **Descripción** Balancea el árbol a partir del nodo dado.
    - **Entrada** Ninguna.
    - **Salida** Retorna el nodo balanceado.

15. **insertar(T val)**

- **Descripción** Inserta un valor en el árbol y realiza el balanceo.
- **Entrada** Ninguna.
- **Salida** El árbol contiene el nuevo valor y se mantiene balanceado.

16. **eliminar(T val)**

- **Descripción** Elimina un valor del árbol y balancea el árbol.
- **Entrada** Ninguna.
- **Salida** El árbol se actualiza y se mantiene balanceado después de eliminar el valor.

17. rotacionDerecha(NodoBinario<T> y)*

- **Descripción** Realiza una rotación a la derecha en el nodo y.
- **Entrada** y debe tener un hijo izquierdo.
- **Salida** El subárbol se reestructura balanceando y.

18. rotacionIzquierda(NodoBinario<T> x)*

- **Descripción** Realiza una rotación a la izquierda en el nodo x.
- **Entrada** x debe tener un hijo derecho.
- **Salida** El subárbol se reestructura balanceando x.

19. obtenerSucesor(NodoBinario<T> nodo)*

- **Descripción** Encuentra el sucesor del nodo dado.
- **Entrada** El nodo debe tener un subárbol derecho.
- **Salida** Retorna el nodo sucesor en el árbol AVL.

Conclusión

De acuerdo con el árbol AVL se puede evidenciar que este es fundamental en funciones de buscar, insertar y eliminar puesto que al realizar algunas de estas operaciones se debe de balancear el árbol usando alguna rotación, por ejemplo, después de insertar el valor 5, el árbol AVL se desbalanceó, y la rotación derecha en el nodo 30 fue suficiente para reequilibrarlo. Así, el árbol sigue siendo un árbol de búsqueda binaria balanceado, manteniendo su propiedad AVL.

Quad-Tree

¿Qué es?

Es una estructura de datos que almacena datos y ademas subdivide el mundo a lo largo de planos alineados con los ejes, representando infinitas áreas cuadradas en donde los nodos internos de esta estructura son también finitos, áreas cuadradas que forman un cuadrante de la región de su nodo padre.

Características

- Cada nodo interno tiene 4 hijos
- Los nodos hojas son elementos finitos

- Los nodos hoja llenos se dividirán en cuatro cuadrantes y los datos se insertarán en los cuadrantes.

¿Para que funciona?
El árbol Quad-Tree es esencial para realizar planos 2D, es decir dividir recursivamente un plano bidimensional en un plano de 4 cuadrantes y que en cada cuadrante contenga un conjunto finito de elementos, teniendo 0 o 4 hijos cada nodo. Fundamental en la compresión de imágenes y entre mas profundidas se le de a la imagen, mayor claridad tendrá esta.

¿Cómo lo implementamos?
Se implementará realizando TADS el cual nos va a facilitar comprender a fondo la funcionalidad de este tipo de árbol y su respectiva implementación, realizando operaciones.

TADS Quad-Tree

TAD Nodo<T>

Descripción
Representa un nodo en el árbol Quad-Tree.

Atributos
- Pair<T,T> dato: Almacena el dato del nodo el cual representa un punto en el espacio.
- Nodo* NE: Apunta al nodo noreste.
- Nodo* SE: Apunta al nodo sureste.
- Nodo* NW: Apunta al nodo noroeste.
- Nodo* SW: Apunta al nodo suroeste.

Métodos
- Void insertar(pair<T,T> val): Inserta un nuevo valor en el quadtree, creando nodos hijos según corresponda.
- Void fijarDato(pair<T,T> val): Establece el valor del dato del nodo.

TAD Arbol<T>

Descripción
Representa el árbol Quad-Tree en su conjunto.

Atributos
- Nodo<T>* raiz: Apunta a la raíz (Nodo fuente) del árbol Quad-Tree.

Métodos
- Arbol(): Constructor que inicializa un árbol vacío.
- Arbol(pair<T,T> val): Constructor que inicializa el árbol con un valor.
- Bool esVacio(): Devuelve verdadero si el árbol está vacío.

- Pair<T,T> obtenerRaiz(): Devuelve el valor de la raíz del árbol.
- Void fijarRaiz(Nodo<T>* root): Establece la raíz del árbol.
- Void insertar(pair<T,T> val): Inserta un nuevo valor en el quadtree.
- Int altura(): Devuelve la altura del quadtree.
- Int tamano(): Devuelve el tamaño del quadtree.
- Nodo<T>* buscar(pair<T,T> val): Busca un nodo en el quadtree.
- Void preOrden(): Realiza un recorrido en preorden.
- Void posOrden(): Realiza un recorrido en postorden.

En la implementación del árbol Quad-Tree se realizó el siguiente diagrama el cual ejemplifica mejor la información proporcionada y que es importante al momento de realizar una correcta implementación. También para la implementación se usaron clases, atributos, métodos(recorrido) para poder realizar funciones implementadas en arboles anteriores para buscar, insertar y eliminar.

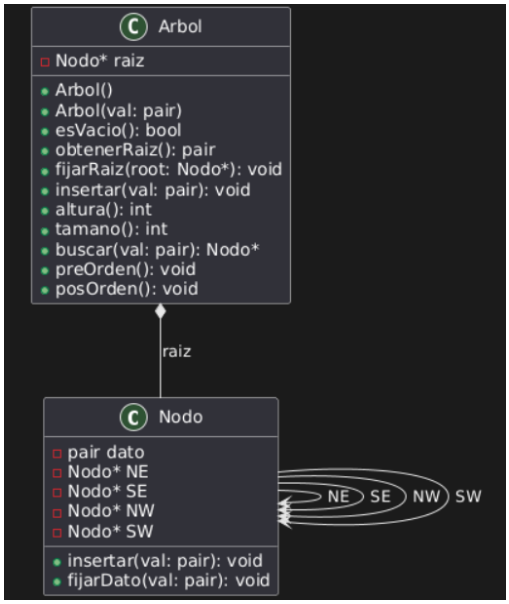


Diagrama Quad – Tree / Una descripción

Clase Nodo

Esta clase representa un nodo en una estructura de árbol, con una estructura orientada a direccionamientos en cuatro direcciones (NE, SE, NW, SW) para nuestro árbol QuadTree.

Atributos

- Pair dato: Representa el valor almacenado en el nodo en forma de par de coordenadas (Cualquier tipo de dato en pares).

- Nodo* NE, Nodo* SE, Nodo* NW, Nodo* SW: Son punteros a los nodos en las cuatro direcciones cardinales relativas:
- NE: Noreste.
- SE: Sureste.
- NW: Noroeste.
- SW: Suroeste.

Métodos
- Insertar (val: pair): Inserta un valor en el nodo, manteniendo la estructura cuadrante para nuestro Quad - Tree.
- FijarDato(val: pair): Permite fijar o actualizar el valor dato en el nodo.

Clase Árbol
Esta clase representa al árbol, manteniendo una referencia a la raíz y proporcionando operaciones de manipulación y recorrido.

Atributos
- Nodo* raiz: Puntero a la raíz del árbol.

Métodos
- Arbol(): Constructor de la clase Arbol.
- Arbol(val: pair): Constructor que inicializa el árbol con un valor inicial.
- EsVacio(): Verifica si el árbol está vacío.
- ObtenerRaiz(): Devuelve el valor de la raíz en forma de par.
- FijarRaiz(root: Nodo*): Permite establecer un nodo como raíz del árbol.
- Insertar(val: pair): Inserta un nuevo valor en el árbol.
- Altura(): Calcula la altura del árbol.
- Tamano(): Devuelve el número total de nodos en el árbol.
- Buscar(val: pair): Busca un valor específico dentro del árbol y devuelve el nodo que lo contiene.

Métodos de recorrido

- PreOrden(): Recorre el árbol en preorden (raíz - subárboles en orden específico).
- PosOrden(): Recorre el árbol en posorden (subárboles en orden específico - raíz).

Plan de pruebas

De acuerdo a los TADS implementados y de el diagrama realizado se mostrara su plan de pruebas con el fin de comprobar su correcto funcionamiento del árbol quad-tree, para esto se harán funciones de inertar, prueba de tamaño, buscar elemento, recorrer nodo.

Inserción

Se desea insertar los elementos: (10, 20), (15,25) y (5,30).

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

Prueba de tamaño

Se verifico que el tamaño del Quad-Tree se actualice correctamente cada vez que se haga algún operación o alteración.

```
Tamaño del árbol: 4
```

Búsqueda

Se busco el valor (15, 25) El cual ya se había insertado antes y se encontró correctamente, mostrando como resultado el valor que se estaba buscando.

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

Recorrido

- Se realizaron recorrido en PreOrder (Desde la raíz al subárbol izquierdo y luego al subárbol derecho) Y también un recorrido por PostOrder (Se pasa del subárbol izquierdo al subárbol derecho y luego a la raíz).

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

Conclusión

El árbol Quad – Tree es fundamental en planos bidimensionales los cuales se separan por cuadrantes y en cada uno de estos se tengas nodos, se evidencio que gracias a esta jerarquía se pueden realizar operaciones de inserción, búsqueda, recorridos, etc. Entorno a esto se hizo una buena practica para identificar el correcto funcionamiento de este.

Pontificia Universidad JAVERIANA Bogotá

Árbol Binario Ordenado

¿Qué es?
Un árbol binario es una estructura de datos que representan elementos en forma de nodos, teniendo cada nodo como máximo 2 hijos, uno izquierdo y otro derecho, este árbol debe de cumplir ciertas características para ser ordenado, es decir para todos los valores de los nodos en su subárbol izquierdo son menores que el valor del nodo y a su misma vez todos los valores de los nodos en su subárbol derecho son mayores que el valor del nodo.

Características
- El subárbol izquierdo es menor que el nodo raiz
- El subárbol derecho es mayor que el nodo raiz
- Es ordenado para encontrar valores fácilmente, dependiendo de si es menor o mayor que algun nodo.

¿Para que funciona?
El árbol Binario ordenado es esencial para realizar búsquedas de elementos dependiendo del valor a buscar, si es menor se ira por un camino y si es mayor se dirigirá a otro camino, a su misma vez al ser un árbol ordenado de menor a mayor se puede insertar elementos y eliminar elementos dependiendo del caso.

¿Cómo lo implementamos?
Se implementará realizando TADS el cual nos va a facilitar comprender a fondo la funcionalidad de este tipo de árbol y su respectiva implementación, realizando operaciones mencionadas anteriormente como buscar, insertar y eliminar.

Se implementaron los siguientes archivos para la realización del árbol binario ordenado, tales archivos son los TADS, descripciones de funciones y más componentes muy importantes.

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

Descripción de Clases y Métodos

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

Métodos Principales

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

ArbolBinario

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

Métodos Principales

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

Main.cpp

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

Ejemplo de Uso en Main.cpp

El archivo Main.cpp contiene la lógica de prueba, que utiliza el árbol binario para demostrar las funcionalidades descritas.

Consideraciones Finales

Gestión de memoria

Se tiene que asegúrar de manejar adecuadamente la memoria, al momento de realizar operaciones de buscar, inserter y eliminar para evitar fugas de memoria.

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

Conclusión

La implementación de este árbol binario ordenado permite una ejecución eficiente de las operaciones de inserción, eliminación y búsqueda, lo cual es fundamental en el manejo de grandes conjuntos de datos. Gracias a la estructura modular de las clases NodoBinario y ArbolBinario, cada un cumple funciones específicas, contribuyendo a la claridad y eficiencia del programa.

Árbol KD-Tree

¿Qué es?

Un árbol binario es una estructura de datos que representan puntos en distintos espacios en varias dimensiones, teniendo cada espacio dividido en espacios de k dimensiones según lo requiera, este árbol es batssnte importante a la hora de graficar coordenadas 2D y 3D debe de cumplir ciertas características para ser ordenado, es decir para todos los valores de los nodos en su subárbol izquierdo son menores que el valor del nodo y a su misma vez todos los valores de los nodos en su subárbol derecho son mayores que el valor del nodo.

Características

- Es más balanceado que un árbol de búsqueda binario normal
- Su complejidad es O (log n)
- Al tener mayores dimensiones, su eficiencia disminuye al realizar busquedas.

¿Para que funciona?

El árbol KD-Tree es esencial para realizar búsquedas de elementos al tener baja dimensionalidad, es decir, a mayor dimensionalidad, más difícil será realizar una búsqueda, el árbol KD-Tree es especial para realizar búsquedas de vecinos más cercanos, rango de búsqueda y problemas de búsqueda espacial.

¿Cómo lo implementamos?

Se implementará realizando TADS el cual nos va a facilitar comprender a fondo la funcionalidad de este tipo de árbol y su respectiva implementación, realizando operaciones como insertar, eliminar y buscar.

Se implementaron los siguientes archivos para la realización del árbol KD-Tree, tales archivos son los TADS, descripciones de funciones y más componentes muy importantes.

Pontificia Universidad JAVERIANA Bogotá

1. TAD Punto

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

Atributos esenciales

x: Representa la coordenada en el eje X.
y: Representa la coordenada en el eje Y.

Operaciones principales

- Punto& operator=(const Punto &p): Permite asignar un punto a otro.
- bool operator==(const Punto &p) const: Verifica si dos puntos son iguales.
- int distanciaEuclidiana(Punto val2): Calcula la distancia euclidiana entre este punto y otro.
- friend std::ostream& operator<<(std::ostream &o, const Punto &p): Sobrecarga del operador de salida para facilitar la impresión de un punto.

Propósito

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

2. TAD NodoKD

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

Atributos esenciales

Punto dato: El punto almacenado en el nodo.
NodoKD *hijoIzq: Puntero al nodo hijo izquierdo.
NodoKD *hijoDer: Puntero al nodo hijo derecho.

Operaciones principales

- NodoKD(): Constructor por defecto.
- NodoKD(Punto val): Inicializa el nodo con un punto específico.
- NodoKD(): Destructor que gestiona la memoria del nodo.
- bool esHoja(): Verifica si el nodo es una hoja, es decir, si no tiene hijos.
- Punto obtenerDato(): Obtiene el punto almacenado en el nodo.
- void fijarDato(Punto val): Asigna un nuevo valor de punto al nodo.
- NodoKD* obtenerHijoIzq(): Retorna el puntero al hijo izquierdo.
- NodoKD* obtenerHijoDer(): Retorna el puntero al hijo derecho.
- void fijarHijoIzq(NodoKD *izq): Establece el puntero al nuevo hijo izquierdo.
- void fijarHijoDer(NodoKD *der): Establece el puntero al nuevo hijo derecho.

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

3. TAD ArbolKD

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

Atributos esenciales

- NodoKD *raiz: Puntero al nodo raíz del árbol.

Operaciones principales
- ArbolKD(): Constructor que inicializa el árbol vacío.
- ArbolKD(Punto val): Constructor que inicializa el árbol con un nodo raíz.
- ArbolKD(): Destructor que libera los recursos del árbol.
- Punto datoRaiz(): Retorna el punto almacenado en la raíz.
- NodoKD* obtenerRaiz(): Devuelve el nodo raíz.
- void fijarRaiz(NodoKD *n_raiz): Establece un nuevo nodo raíz.
- bool esVacio(): Comprueba si el árbol está vacío.
- bool insertar(Punto val): Inserta un nuevo punto en el árbol.
- NodoKD* insertarRec(NodoKD *nodo, Punto val, bool &insertado, char dimension): Inserta recursivamente un nodo en el árbol.
- NodoKD* vecinoCercano(NodoKD *raiz, Punto val): Encuentra el punto más cercano al punto proporcionado.
- void vecinoCercanoRec(NodoKD *nodo, Punto val, char dimension, NodoKD *&mejorNodo, int &mejorDist): Método recursivo para buscar el vecino más cercano en el árbol.

Recorridos

- void preOrden(): Realiza un recorrido en preorden.

- void inOrden(): Realiza un recorrido inorden.

- void posOrden(): Realiza un recorrido posorden.

- void nivelOrden(): Realiza un recorrido por niveles.

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

Conclusión
La implementación del KD-Tree en 2D está bien estructurada y cumple con los requisitos de diseño y funcionalidad para cada TAD (Punto, NodoKD y ArbolKD). Cada uno de los archivos implementa de forma adecuada los métodos esenciales, como la asignación, comparación y cálculo de distancias en el TAD Punto, la gestión de nodos y sus hijos en el TAD NodoKD, y la estructura completa del árbol KD con operaciones de inserción y búsqueda en el TAD ArbolKD.
El archivo de prueba confirma que la implementación puede realizar correctamente las operaciones básicas de inserción y recorrido, lo cual es crucial para verificar la funcionalidad de este tipo de estructura de datos. En general, la implementación demuestra un diseño limpio y eficiente que facilita la manipulación y búsqueda de puntos en un espacio 2D, cumpliendo satisfactoriamente el propósito del KD-Tree.

Árbol de expresión

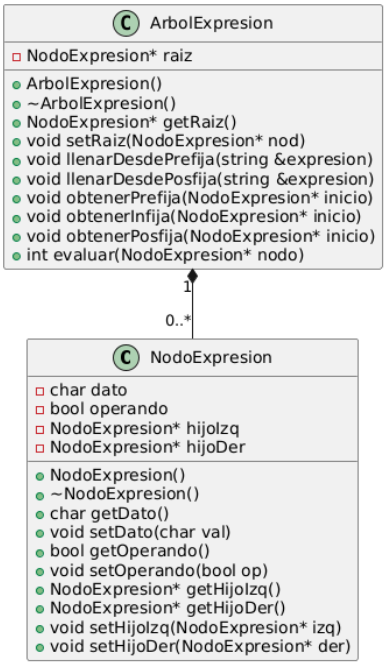
¿Qué es?
Un árbol de expresión es una estructura de datos que representan operaciones entre nodos base y nodos hojas, además los operadores están organizados de acuerdo con la precedencia y la jerarquía de la expresión, donde los operadores de mayor precedencia se encuentran más arriba en el árbol y los operadores de menor precedencia más abajo

Características
- Es muy recursivo, ya que cada subárbol representa una expresión matemática.
- Se evalúa desde las hojas hasta la raíz debido a sus operaciones.
- Debe de estar equilibrado.

¿Para que funciona?
El árbol de expresión es esencial para realizar operaciones que representan expresiones matemáticas y lógicas, cada Nodo del árbol representa un operador u operando de alguna expresión, siendo así un árbol especializado en matemáticas y sus operaciones.

¿Cómo lo implementamos?
Se implementará realizando TADS el cual nos va a facilitar comprender a fondo la funcionalidad de este tipo de árbol y su respectiva implementación, realizando operaciones que se mencionaran a continuación.

Se implementaron los siguientes archivos para la realización del árbol de expresión, tales archivos son los TADS, descripciones de funciones y más componentes muy importantes.



Clase ArbolExpresion
Es una clase que gestiona un árbol de expresión.

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

Métodos
- ArbolExpresion(): Constructor por defecto que inicializa el árbol vacío.
- ArbolExpresion(): Destructor.
- NodoExpresion* getRaiz(): Retorna el nodo raíz del árbol.
- void setRaiz(NodoExpresion* nod): Establece el nodo raíz.
- void llenarDesdePrefija(string &expresion): Llena el árbol usando una expresión en notación prefija.
- void llenarDesdePosfija(string &expresion): Llena el árbol usando una expresión en notación posfija.

Recorridos
- void obtenerPrefija(NodoExpresion* inicio): Imprime la expresión en notación prefija.
- void obtenerInfija(NodoExpresion* inicio): Imprime la expresión en notación infija.
- void obtenerPosfija(NodoExpresion* inicio): Imprime la expresión en notación posfija.
- int evaluar(NodoExpresion* nodo): Evalúa la expresión almacenada en el árbol, comenzando desde el nodo especificado.

Clase NodoExpresion
Representa un nodo del árbol de expresión.

Atributos

- char dato: Contiene el operador o el operando.
- bool operando: Indica si el nodo es un operando.
- NodoExpresion* hijoIzq: Puntero al hijo izquierdo.
- NodoExpresion* hijoDer: Puntero al hijo derecho.

Métodos

- NodoExpresion(): Constructor por defecto.
- NodoExpresion(): Destructor.
- char getDato(): Retorna el valor almacenado en el nodo.
- void setDato(char val): Establece un nuevo valor en el nodo.
- bool getOperando(): Retorna si el nodo es un operando.
- void setOperando(bool op): Establece si el nodo es un operando.
- NodoExpresion* getHijoIzq(): Retorna el hijo izquierdo del nodo.
- NodoExpresion* getHijoDer(): Retorna el hijo derecho del nodo.
- void setHijoIzq(NodoExpresion* izq): Establece el hijo izquierdo del nodo.
- void setHijoDer(NodoExpresion* der): Establece el hijo derecho del nodo.

Plan de pruebas

De acuerdo a los TADS implementados y del diagrama realizado se mostrara su plan de pruebas con el fin de comprobar su correcto funcionamiento del árbol de expresión, para esto se harán funciones de un nodo.

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

Conclusión

El proceso de construcción del árbol de expresión permite observar cómo se desglosan las operaciones y se estructura la evaluación de una expresión en notación postfija. Esta técnica es esencial en aplicaciones donde se necesita una evaluación precisa y eficiente de expresiones matemáticas o lógicas, además este tipo de árbol siempre será fundamental en operaciones. Es fundamental aprender a construir estos tipos de árboles frente s situaciones que se puedan presentar.

Pontificia Universidad JAVERIANA Bogotá

Árbol

¿Qué es?

Un árbol es una estructura de datos jerárquica compuesta por nodos, donde cada nodo puede tener cero o más nodos hijos. El nodo principal se llama "raíz", y los nodos sin hijos son "hojas".

Características

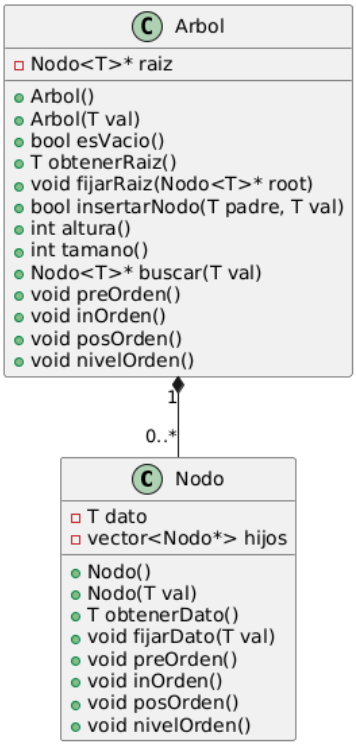
- **Raíz** Nodo principal del árbol.
- **Hojas** Nodos sin hijos.
- **Nivel Profundidad** en la jerarquía de un nodo.
- **Altura Longitud máxima** desde la raíz hasta una hoja.
- **Padre e hijo** Relación jerárquica entre nodos.
- Subárboles Estructura formada por un nodo y sus descendientes.

¿Para que funciona?

Representa relaciones estructuradas de sistema de archivos, permitiendo búsquedas eficientes en estructuras como árboles binarios de búsqueda. Y además, utilizado en algoritmos como los de grafos y procesamiento de datos jerárquicos.

¿Cómo lo implementamos?

Se implementará realizando TADS el cual nos va a facilitar comprender a fondo la funcionalidad de un árbol y su respectiva implementación, realizando operaciones que se mencionaran a continuación.



Clase Árbol

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

Atributos

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

Métodos

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

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

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

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

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

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

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

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

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

Recorridos

void preOrden(): Realiza un recorrido en preorden.

void inOrden(): Realiza un recorrido inorden.

void posOrden(): Realiza un recorrido en postorden.

void nivelOrden(): Realiza un recorrido por niveles.

Clase Nodo

Representa un nodo en el árbol.

Atributos

T dato: El valor almacenado en el nodo.

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

Métodos

Nodo(): Constructor por defecto.

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

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

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

Recorridos

void preOrden(): Recorre el subárbol en preorden.

void inOrden(): Recorre el subárbol en inorden.

void posOrden(): Recorre el subárbol en postorden.

void nivelOrden(): Recorre el subárbol por niveles.

Plan de pruebas

De acuerdo a los TADS implementados y del diagrama realizado se mostrará su plan de pruebas con el fin de comprobar su correcto funcionamiento del árbol de expresión, para esto se harán funciones de un nodo.

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

Insertar nodos 9 y 10 como hijos de 6

Insertar nodo 11 como hijo de 7

Conclusión

En un árbol de búsqueda binaria, cada nodo tiene un valor y, como máximo, dos hijos. El hijo izquierdo contiene valores menores que el nodo padre, y el hijo derecho contiene valores mayores, lo cual facilita la búsqueda binaria en el árbol. En su versión balanceada, como los árboles AVL o los árboles rojo-negro, los árboles de búsqueda binaria permiten realizar búsquedas, inserciones y eliminaciones en tiempo logarítmico O(log n), lo que es considerablemente más rápido que el tiempo lineal O(n) en listas, pilas o colas para estos mismos procesos en grandes conjuntos de datos.

En conclusión, los árboles de búsqueda binaria son herramientas poderosas y eficientes para almacenar y organizar datos en casos donde se necesita realizar operaciones frecuentes de búsqueda y ordenamiento, y resultan ideales en escenarios donde el costo de mantener el equilibrio del árbol se compensa con el beneficio de realizar consultas rápidas**.**

Bibliografía

1. GeeksforGeeks. (n.d.). Tree Data Structure. Recuperado de https://www.geeksforgeeks.org/binary-tree-data-structure/

2. Khan Academy. (n.d.). Trees. Recuperado de https://www.khanacademy.org/computing/computer-science/algorithms/trees

3. Weiss, M. A. (2013). Data structures and algorithm analysis in C++ (4th ed.). Pearson.