

Fecha: 26/03/2025

Taller 3

Autores: Santiago Mesa, Jeronimo Chaparro Tenorio.

Materia: Estructura de Datos.

Pontificia Universidad Javeriana

0.Índice

- 1.Resumen de lo solicitado
- 2.Objetivo
- 3.Introducción
- 4.Desarrollo
 - 4.1 Diseño de TADs
 - 4.2 Planes de Pruebas
- 5.Conclusión

1.Resumen de lo solicitado:

Se nos pide la revisión de seis tipos de árboles (Árbol general, Árbol Binario, Árbol AVL, Árbol de Expresión, Árbol KD y Árbol Quad), a su vez que realizar un plan de pruebas y una breve conclusión.

2.Objetivo:

Analizar los árboles entregados por el profesor, realizar los TADs de cada árbol y un plan de prueba para las funciones más relevantes de cada árbol.

3.Introducción:

Las estructuras de árboles son un tipo de estructura de datos jerárquica donde cada elemento, llamado nodo, está conectado a otros mediante enlaces llamados aristas. Se utilizan para representar relaciones estructuradas, como jerarquías, expresiones matemáticas y divisiones espaciales, para este taller nos vamos a enfocar en los siguientes seis tipos de árboles.

- **Árbol General:**

Es una estructura jerárquica donde cada nodo puede tener un número arbitrario de hijos. Se usa para representar relaciones complejas como

sistemas de archivos o estructuras jerárquicas.

- **Árbol Binario:**

Es un tipo de árbol donde cada nodo tiene como máximo dos hijos: izquierdo y derecho. Se utiliza en múltiples aplicaciones como estructuras de búsqueda y expresiones matemáticas.

- **Árbol AVL:**

Es un árbol binario de búsqueda balanceado donde la diferencia de altura entre los subárboles de cualquier nodo no puede ser mayor a 1. Se autorregula mediante rotaciones para optimizar las búsquedas.

- **Árbol de Expresión:**

Se usa para representar expresiones matemáticas o lógicas. Sus nodos internos son operadores, y sus hojas son operandos.

- **Árbol KD (K-Dimensional):**

Es una estructura para organizar puntos en un espacio k-dimensional. Se usa en aplicaciones como búsqueda espacial y machine learning, facilitando consultas de vecinos más cercanos.

- **Árbol Quad:**

Es una estructura de división espacial donde cada nodo tiene cuatro hijos. Se usa para representar imágenes, dividir espacios en gráficos computacionales y en sistemas de geolocalización.

- **árbol Red Black:**

Es un tipo de árbol binario de búsqueda auto-balanceado que utiliza colores (rojo y negro) en sus nodos para garantizar un equilibrio lógico. Esto permite realizar inserciones, eliminaciones y búsquedas en tiempo logarítmico, asegurando eficiencia incluso en el peor caso.

4.Desarrollo:

4.1 Diseño de TADs:

TAD Árbol

Datos Mínimos:

`raiz`, señala el nodo que corresponde a la raíz del árbol.

Comportamiento:

`Arbol()`, crea un árbol con raíz nula.

`Arbol(val)`, crea una árbol con raíz de valor val.
`~Arbol()`
`esVacio()`, booleano, devuelve verdadero si la raíz del árbol es nula.
`obtenerRaiz()`, Nodo, retorna un apuntador a la raíz del árbol.
`fijarRaiz(nraiz)`, recibe un apuntador al Nodo y lo asigna a la raíz del árbol.
`insertarNodo(padre,n)`, booleano, recibe el valor del padre, el cual se busca, y se le ingresa un Nodo hijo con valor n, si se inserta retorna verdadero.
`eliminarNodo(n)`, booleano ,elimina un nodo de valor n, retorna verdadero si se elimina.
`buscarNodo(val)`, retorna un apuntador al nodo de valor val.
`altura()`, entero, retorna la altura del árbol.
`tamano()`, entero, retorna el número de nodos en el árbol.
`preOrden()`, imprime en preorden el árbol.
`posOrden()`, imprime en posorden el árbol.
`inOrden()`, imprime en inorden el árbol.
`nivelOrden()`, imprime en nivel orden el árbol.

TAD Nodo

Datos Mínimos:

`T dato`,?, contenido del nodo

`desc`, contenedor, contiene un conjunto de apuntadores a nodos.

Comportamiento:

`Nodo()`, crea un nodo vacío.

`Nodo(const T& val)`, crea un nodo a partir de un valor

`~Nodo()`

`obtenerDato()`,?, retorna el dato que es contenido por el nodo

`fijarDato(val)`, reemplaza el valor de un nodo por val.

`obtenerDesc()`, contenedor, retorna la el conjunto de descendientes

`fijarDesc(listaDesc)`, fija el conjunto de Nodos listaDesc como el conjunto de descendientes del nodo

`adicionarDesc(nval)`, adiciona un Nodo creado a partir de val al conjunto de descendientes

`eliminarDesc(val)`, booleano, elimina el nodo de valor val del conjunto de descendientes, retorna verdadero si se elimina.

`buscarDesc(val)`, nodo*, retorna un apuntador al descendiente de valor val.

`limpiarLista()`, elimina la lista de descendientes.

`numDesc()`, entero, retorna el número de descendientes del nodo.

`insertarNodo(padre, n)`, inserta un nodo de valor `n`, en el nodo padre de alguno de los subárboles. retorna verdadero si se logra.

`eliminarNodo(n)`, elimina el nodo de valor `n` de alguno de los subárboles

`buscarNodo(val)`, apuntador, retorna un apuntador al nodo de valor `val` de alguno de los subárboles.

`altura()`, entero, retorna la altura del nodo

`tamano()`, entero, retorna el número de nodos del subárbol que tiene como raíz el nodo

`preOrden()`, imprime el árbol que tiene como raíz el nodo en PreOrden.

`posOrden()`, imprime el árbol que tiene como raíz el nodo en PosOrden

`inOrden()`, imprime el árbol que tiene como raíz el nodo en inOrden.

`nivelOrden(int nivel, int lvActual)`, imprime el árbol que tiene como raíz el nodo en NivelOrden.

TAD ArbolBinarioG

Datos Mínimos:

`raiz`, `NodoBinario`, señala al primer nodo del árbol binario.

Comportamiento:

`ArbolBinarioG()`,

`getRaiz()`, `NodoBinario`, retorna el `nodoBinario` correspondiente a la raíz del Árbol.

`esVacio()`, retorna verdadero si el árbol no posee nodo raíz.

`datoRaiz()`, `?`, retorna el contenido de la raíz.

`altura()`, entero, retorna la altura del árbol.

`tamano()`, entero, retorna el número de nodos que posee el árbol.

`insertar(valor)`, inserta un valor en el árbol siguiendo los parámetros de orden.

`altura(subarbol)`, entero, retorna la altura de un subárbol que es enviado por parámetro.

`tamano(subarbol)`, entero, retorna el número de nodos que posee un subárbol que es enviado por parámetro.

`insertar(valor, subárbol, pos)`, ingresa un `nodoBinario` de valor "`valor`", al subárbol en la posición indicada, `i` para hijo izquierdo, `d` para derecho.

`eliminar(valor)`, busca un valor en el árbol y lo elimina.

`buscar(valor)`, busca un valor en el árbol y retorna el apuntador si lo encuentra.

`buscarE(valor)`, busca un valor en el árbol y retorna el apuntador a su padre.

`preOrden(subarbol)`, realiza la impresión preorden de un subárbol

`inOrden(subarbol)`, realiza la impresión Inorden de un subárbol

`posOrden(subarbol)`, realiza la impresión posOrden de un subarbol

`nivelOrden(subarbol)`, realiza la impresión NivelOrden de un subárbol

`preOrden()`, realiza la impresión preorden del árbol

`inOrden()`, realiza la impresión preorden del árbol

`posOrden()`, realiza la impresión preorden del árbol

`nivelOrden()`, realiza la impresión preorden del árbol

TAD NodoBinario

Datos Mínimos:

`dato`, T, posee el contenido del nodo

`lzq`, NodoBinario,apuntador hacia el hijo izquierdo

`Dere`, NodoBinario,apuntador hacia el hijo derecho

Comportamiento:

`NodoBinario(dato)`, crea un nodo binario e inicializa su dato con el dato enviado por parámetro.

`NodoBinario()`, crea un nodo binario vacío.

//Setters y Getters

`obtenerDato()`

`fijarDato(val)`

`obtenerHijolzq()`

`obtenerHijoDer()`

`fijarHijolzq(izq)`

`fijarHijoDer(der)`

TAD ArbolExp

Datos Mínimos:

`raiz`, NodoExp , indica el inicio del arbol

`operadores`, conjunto de operadores validos para el arbol.

Comportamiento:

`ArbolExpresion()`, crea un arbolExpresion

`llenarDesdePrefija(expresion)`, recibe una expresión en Polaca y llena el arbol a partir de esta.

`llenarDesdePosfija (expresion)`, recibe una expresión en Polaca Inversa y llena el arbol a partir de esta.

`obtenerPrefija()`, cadena de caracteres, retorna una cadena de caracteres que expresa el árbol en Polaca.

`obtenerInfija()`,cadena de caracteres, retorna una cadena de caracteres que expresa el árbol en Infija.

`obtenerPosfija()`,cadena de caracteres, retorna una cadena de caracteres que expresa el árbol en Polaca Inversa.

`Prefija(subarbol)`,cadena de caracteres,retorna una cadena de caracteres que expresa el subárbol en Polaca.

`Posfija(subarbol)`,cadena de caracteres,retorna una cadena de caracteres que expresa el subárbol en Polaca inversa.

`Infija(subarbol)`,cadena de caracteres,retorna una cadena de caracteres que expresa el subárbol en Infija.

`evaluar()`,entero, retorna el resultado de la expresión contenida en el árbol.

`esOp(ope)`,booleano ,verifica si una cadena de caracteres esta en el conjunto de operadores validos, si lo está retorna true.

`eval(NodoExp *)`, entero,evalua la expresión contenida en un subarbol.

`llenarDesdePrefijaa(vector, pos, actual)`, `NodoExp*`, construye un árbol de expresión a partir de una notación prefija.

`llenarDesdePosfijaa(vector, pos, actual)`, `NodoExp*`, construye un árbol de expresión a partir de una notación posfija.

`tokenizar(s, vector)`, la divide en tokens una cadena para su posterior procesamiento en la construcción del árbol.

TAD NodoExp

Datos Mínimos:

`data`,string, posee el contenido del nodo

`left`, `NodoExp`,apuntador hacia el hijo izquierdo

`right`, `NodoExp`,apuntador hacia el hijo derecho

`op`, booleano, determina si el contenido es un operador(true), o un número(false).

Comportamiento:

`NodoExp(dato)`,crea un nodo binario e inicializa su dato con el dato enviado por parámetro.

`NodoExp()`crea un nodo binario vacío.

//Setters y Getters

`getData()`

`setData(val)`

`getLeft()`

`getRight()`
`setLeft(left)`
`setRight(right)`
`getOp()`
`setOp(left)`

TAD ArbolQuad

Datos Mínimos:

`raíz`: Puntero al nodo raíz del árbol.

Comportamiento:

`Arbol()`, Constructor por defecto.

`Arbol(pair<T,T> val)`: Constructor con un valor inicial.

`esVacio()`, booleano, Devuelve si el árbol está vacío.

`obtenerRaiz()`, pair<T,T> Retorna el dato de la raíz.

`fijarRaiz(Nodo<T>* root)`, Asigna una nueva raíz.

`insertar(pair<T,T> val)`, Inserta un valor en el árbol.

`eliminar(T& val)`, booleano, Elimina un nodo con el valor dado.

`buscar(pair<T,T> val)`, Nodo<T>*, Busca un nodo con el valor especificado.

`altura()`, entero, Calcula la altura del árbol.

`tamano()`, entero Calcula el número total de nodos.

`preOrden()`, `posOrden()`: Recorridos del árbol.

TAD NodoQuad

Datos Mínimos:

`dato`, Par de valores almacenados en el nodo.

`NW`, `NE`, `SW`, `SE`, Punteros a los hijos del nodo (para un QuadTree).

Comportamiento:

`Nodo()`, Constructor por defecto.

`Nodo(pair<T,T> val)`, Constructor con un valor inicial.

`obtenerDato()`, pair<T,T> Retorna el valor almacenado en el nodo.

`fijarDato(pair<T,T> val)`, Modifica el valor del nodo.

`insertar(pair<T,T> val)`, Inserta un nuevo valor en el nodo.

`buscar(pair<T,T> val)`, Nodo* Busca un nodo con el valor especificado.

`altura()`, entero, `tamano()`, entero, Calculan la altura y el tamaño del subárbol.

`preOrden()`, `posOrden()`, Métodos de recorrido.

TAD kdtree (Árbol KD)

Datos Mínimos:

`raiz`: Puntero al nodo raíz del árbol.

Comportamiento:

`kdtree()`, Constructor del árbol.

`esVacio()`, booleano, Devuelve true si el árbol está vacío.

`datoRaiz()`, T, Devuelve el dato almacenado en la raíz.

`altura()`, entero, Retorna la altura del árbol.

`tamano()`, entero, Retorna el número total de nodos en el árbol.

`insertar(T& val)`, Inserta un nuevo elemento en el árbol.

`eliminar(T& val)`, booleano, Elimina un nodo con el valor especificado.

`buscar(T& val)`, kdnodo<T>*, Busca un nodo con el valor dado y retorna un puntero a él.

`preOrden()`, `inOrden()`, `posOrden()`, `nivelOrden()`, Métodos para recorrer el árbol.

`maximo(int &maxi)`, `minimo(int &mini)`, Encuentran el valor máximo o mínimo en el árbol.

TAD kdnodo (Nodo del Árbol KD)

Datos Mínimos:

`datos`: Vector que almacena las coordenadas del punto en T dimensiones.

`hijoIzq`: Puntero al hijo izquierdo.

`hijoDer`: Puntero al hijo derecho.

`tag`: Índice del eje de división en el árbol KD.

Comportamiento:

`kdnodo()`, Constructor del nodo.

`obtenerDato()`, T, Devuelve el dato almacenado en el nodo.

`fijarDato(vector<T>& val)`, Asigna un nuevo valor al nodo.

`obtenerHijoIzq()`, `kdnodo<T>*`, `obtenerHijoDer()`, `kdnodo<T>*`,

Devuelven los punteros a los hijos.

`fijarHijoIzq(kdnodo<T> *izq)`, `fijarHijoDer(kdnodo<T> *der)`, Asignan nuevos hijos al nodo.

`fijarTag(int value)`, Asigna un índice de eje al nodo.

`altura()`, entero, Retorna la altura del subárbol con raíz en este nodo.

`tamano()`, entero, Retorna el número de nodos en el subárbol con raíz en este nodo.

`insertar(vector<T>& val)`, Inserta un nuevo punto en el subárbol.

`buscar(vector<T>& val)`, `kdnodo<T>*`, Busca un nodo con el valor dado y lo retorna.

`preOrden()`, `inOrden()`, `posOrden()`, `nivelOrden()`, Métodos para recorrer el subárbol.

`maximo(int &maxi)`, `minimo(int &mini)`, Encuentran el valor máximo o mínimo en el subárbol.

`imprimir()`, Imprime el contenido del nodo.

TAD RedBlackTree (Árbol Rojo-Negro)

Datos Mínimos:

- `root`: Puntero al nodo raíz del árbol.

Comportamiento:

- `RedBlackTree()`: Constructor del árbol. Inicializa la raíz como `nullptr`.
- `insert(int val)`: Inserta un nuevo nodo con valor `val` en el árbol, ajustando colores y haciendo rotaciones si es necesario para mantener las propiedades del árbol rojo-negro.

- `remove(int val)`: Elimina el nodo con valor `val` del árbol, si existe, y realiza correcciones para mantener las propiedades del árbol rojo-negro.
- `printTree()`: Muestra visualmente la estructura del árbol, indicando el valor y el color de cada nodo con indentación jerárquica.
- `inOrderTraversal()`: Recorre e imprime los nodos del árbol en orden (izquierda, raíz, derecha).
- `preOrderTraversal()`: Recorre e imprime los nodos en preorden (raíz, izquierda, derecha).
- `postOrderTraversal()`: Recorre e imprime los nodos en postorden (izquierda, derecha, raíz).

4.2 Planes de Pruebas:

Árbol General:

Caso de Prueba	Valores de Entrada	Resultado Esperado	Resultado Obtenido	Estado (Éxito/Falla)
PreOrden	<pre>Arbol<int> arbol(5); arbol.insert arNodo(5,6); arbol.insert arNodo(5,7); arbol.insert arNodo(5,8); arbol.insert arNodo(6,9); arbol.insert arNodo(6,10) ; arbol.insert arNodo(7,11) ; arbol.preOrd en();</pre>	5, 6, 9, 10, 7, 11, 8	5, 6, 9, 10, 7, 11, 8	✓ Éxito
PosOrden	<pre>Arbol<int> arbol(5); arbol.insert arNodo(5,6);</pre>	9, 10, 6, 11, 7, 8, 5	9, 10, 6, 11, 7, 8, 5	✓ Éxito

	<pre> arbol.insert arNodo (5, 7) ; arbol.insert arNodo (5, 8) ; arbol.insert arNodo (6, 9) ; arbol.insert arNodo (6, 10) ; arbol.insert arNodo (7, 11) ; arbol.posOrd en () ; </pre>			
nivelOrden	<pre> Arbol<int> arbol (5) ; arbol.insert arNodo (5, 6) ; arbol.insert arNodo (5, 7) ; arbol.insert arNodo (5, 8) ; arbol.insert arNodo (6, 9) ; arbol.insert </pre>	5, 6, 7, 8, 9, 10, 11	5, 6, 7, 8, 9, 10, 11	✓ Éxito

	<pre>arNodo(6,10) ; arbol.insert arNodo(7,11) ; arbol.nivelO rden(); }</pre>			
--	--	--	--	--

Árbol Binario:

Caso de Prueba	Valores de Entrada	Resultado Esperado	Resultado Obtenido	Estado (Éxito/Falla)
PreOrden	<pre>ArbolBinario <int> arbol; int a=1, b=2, c=3, d=4, e=5, f=6; arbol.insert ar(a); arbol.insert ar(b); arbol.insert ar(c); arbol.insert ar(d); arbol.insert ar(e); arbol.insert ar(f); arbol.preOrd en();</pre>	1, 2, 3, 4, 5, 6	1, 2, 3, 4, 5, 6	✓ Éxito
PosOrden	<pre>ArbolBinario <int> arbol; int a=1, b=2, c=3, d=4, e=5,</pre>	6, 5, 4, 3, 2, 1	6, 5, 4, 3, 2, 1	✓ Éxito

	<pre> f=6; arbol.insert ar(a); arbol.insert ar(b); arbol.insert ar(c); arbol.insert ar(d); arbol.insert ar(e); arbol.insert ar(f); arbol.posOrden(); </pre>			
nivelOrden	<pre> ArbolBinario <int> arbol; int a=1, b=2, c=3, d=4, e=5, f=6; arbol.insert ar(a); arbol.insert ar(b); arbol.insert ar(c); arbol.insert ar(d); </pre>	1, 2, 3, 4, 5, 6	1, 2, 3, 4, 5, 6	✓ Éxito

	<pre> arbol.insert ar(e); arbol.insert ar(f); arbol.nivelO rden(); </pre>			
inOrden	<pre> ArbolBinario <int> arbol; int a=1, b=2, c=3, d=4, e=5, f=6; arbol.insert ar(a); arbol.insert ar(b); arbol.insert ar(c); arbol.insert ar(d); arbol.insert ar(e); arbol.insert ar(f); arbol.inOrde n(); </pre>	1, 2, 3, 4, 5, 6	1, 2, 3, 4, 5, 6	✓ Éxito

inOrden eliminando el dato 3 (c)	<pre> ArbolBinario <int> arbol; int a=1, b=2, c=3, d=4, e=5, f=6; arbol.insert ar(a); arbol.insert ar(b); arbol.insert ar(c); arbol.insert ar(d); arbol.insert ar(e); arbol.insert ar(f); arbol.elimin ar(c); arbol.inOrde n(); } </pre>	1, 2, 4, 5, 6	1, 2, 4, 5, 6	✓ Éxito
--	--	---------------	---------------	---------

Árbol AVL:

Caso de Prueba	Valores de Entrada	Resultado Esperado	Resultado Obtenido	Estado (Éxito/Falla)
PreOrden	<pre> ArbolBinario AVL<int> arbol; int a=1, b=2, c=3, d=4, e=5, f=6; arbol.insert ar(a); arbol.insert ar(b); arbol.insert ar(c); arbol.insert ar(d); arbol.insert ar(e); arbol.insert ar(f); cout << endl << "Preorden: " << endl; arbol.preOrd en(arbol.get Raiz()); </pre>	3, 2, 1, 5, 4, 6	3, 2, 1, 5, 4, 6	✓ Éxito
PosOrden	<pre> ArbolBinario </pre>	1, 2, 4, 6, 5, 3	1, 2, 4, 6, 5, 3	✓ Éxito

	<pre> AVL<int> arbol; int a=1, b=2, c=3, d=4, e=5, f=6; arbol.insert ar(a); arbol.insert ar(b); arbol.insert ar(c); arbol.insert ar(d); arbol.insert ar(e); arbol.insert ar(f); cout << endl <<"Posorden: " << endl; arbol.posOrd en(arbol.get Raiz()); </pre>			
nivelOrden	<pre> ArbolBinario AVL<int> arbol; </pre>	3, 2, 5, 1, 4, 6	3, 2, 5, 1, 4, 6	✓ Éxito

	<pre> int a=1, b=2, c=3, d=4, e=5, f=6; arbol.insert ar(a); arbol.insert ar(b); arbol.insert ar(c); arbol.insert ar(d); arbol.insert ar(e); arbol.insert ar(f); cout << endl <<"Nivel Raiz: " << endl; arbol.nivelO rden(arbol.g etRaiz()); </pre>			
inOrden	<pre> ArbolBinario AVL<int> arbol; </pre>	1, 2, 3, 4, 5, 6	1, 2, 3, 4, 5, 6	✓ Éxito

	<pre> int a=1, b=2, c=3, d=4, e=5, f=6; arbol.insert ar(a); arbol.insert ar(b); arbol.insert ar(c); arbol.insert ar(d); arbol.insert ar(e); arbol.insert ar(f); cout << endl << "Inorden: " << endl; arbol.inOrden(arbol.getRoot()); </pre>			
inOrden eliminando el dato 3 (c)	<pre> ArbolBinario AVL<int> arbol; int a=1, b=2, c=3, d=4, e=5, f=6; arbol.insert </pre>	1, 2, 4, 5, 6	1, 2, 4, 5, 6	✓ Éxito

	<pre>ar(a); arbol.insert ar(b); arbol.insert ar(c); arbol.insert ar(d); arbol.insert ar(e); arbol.insert ar(f); arbol.elimin ar(c); cout << endl << "Inorden: " << endl; arbol.inOrde n(arbol.getR aiz());</pre>			
--	--	--	--	--

Árbol de Expresión:

Caso de Prueba	Valores de Entrada	Resultado Esperado	Resultado Obtenido	Estado (Éxito/Falla)
obtenerPosfija	<pre> cout<<"1. Construir Arbol Expresion: "<<endl; cout<<"-*/5- 7+113-+2+1*4 3*2-68"<< endl; ArbolExpresi on* arbexp =new ArbolExpresi on(); string exp="-*/5-7+ 113-+2+1*43* 2-68"; arbexp->llen arDesdePrefi ja(exp); cout<<"2. Imprimir Version Posfija"<<"= " << endl; arbexp->obte nerPosfija(a rbexp->getRa iz()); </pre>	<pre> 8 6 - 2 * 3 4 * 1 + 2 + - 3 1 1 + 7 - 5 / * - </pre>	<pre> 8 6 - 2 * 3 4 * 1 + 2 + - 3 1 1 + 7 - 5 / * - </pre>	<div>✓</div> Éxito

<p>obtenerPrefija</p>	<pre> cout<<"1. Construir Arbol Expresion: "<<endl; ArbolExpresion* arbexp2 =new ArbolExpresion(); cout<< "45+23+*6+87 +/12+3*6+23+ /*"<< endl; string exp2="45+23+ *6+87+/12+3* 6+23+/*"; arbexp2->lle narDesdePosf ija(exp2); cout<<"2. Imprimir Version Prefija"<<"= " << endl; arbexp2->obt enerPrefija(arbexp2->get Raiz()); </pre>	<p>• / + * + 4 5 + 2 3 6 + 8 7 / + * + 1 2 3 6 + 2 3</p>	<p>• / + * + 4 5 + 2 3 6 + 8 7 / + * + 1 2 3 6 + 2 3</p>	<p>✓ Éxito</p>
-----------------------	--	--	--	----------------

Árbol KD:

Caso de Prueba	Valores de Entrada	Resultado Esperado	Resultado Obtenido	Estado (Éxito/Falla)
PreOrden	<pre> kdtree<int> arbol; vector<int> a = {5, 4}; vector<int> b = {3, 2}; vector<int> c = {4, 1}; vector<int> d = {8, 7}; vector<int> e = {1, 2}; vector<int> f = {9, 5}; arbol.insert ar(a); arbol.insert ar(b); arbol.insert ar(c); arbol.insert ar(d); arbol.insert ar(e); arbol.insert ar(f); </pre>	<p>(5, 4)</p> <p>(3, 2)</p> <p>(4, 1)</p> <p>(1, 2)</p> <p>(8, 7)</p> <p>(9, 5)</p>	<p>(5, 4)</p> <p>(3, 2)</p> <p>(4, 1)</p> <p>(1, 2)</p> <p>(8, 7)</p> <p>(9, 5)</p>	<p>✓ Éxito</p>

	<pre> cout<<endl; cout<<"Pre Orden:"<<endl; arbol.preOrden(); </pre>			
PosOrden	<pre> kdtree<int> arbol; vector<int> a = {5, 4}; vector<int> b = {3, 2}; vector<int> c = {4, 1}; vector<int> d = {8, 7}; vector<int> e = {1, 2}; vector<int> f = {9, 5}; arbol.insert ar(a); arbol.insert ar(b); arbol.insert ar(c); </pre>	(1, 2) (4, 1) (3, 2) (9, 5) (8, 7) (5, 4)	(1, 2) (4, 1) (3, 2) (9, 5) (8, 7) (5, 4)	<input checked="" type="checkbox"/> Éxito

	<pre> arbol.insert ar(d); arbol.insert ar(e); arbol.insert ar(f); cout<<endl; cout<<"In Orden: "<<endl; arbol.inOrde n(); </pre>			
InOrden	<pre> kdtree<int> arbol; vector<int> a = {5, 4}; vector<int> b = {3, 2}; vector<int> c = {4, 1}; vector<int> d = {8, 7}; vector<int> e = {1, 2}; vector<int> f = {9, 5}; arbol.insert ar(a); </pre>	<p>(1, 2)</p> <p>(4, 1)</p> <p>(3, 2)</p> <p>(5, 4)</p> <p>(9, 5)</p> <p>(8, 7)</p>	<p>(1, 2)</p> <p>(4, 1)</p> <p>(3, 2)</p> <p>(5, 4)</p> <p>(9, 5)</p> <p>(8, 7)</p>	<p>✓ Éxito</p>

	<pre>arbol.insert ar(b); arbol.insert ar(c); arbol.insert ar(d); arbol.insert ar(e); arbol.insert ar(f); cout<<endl; cout<<"In Orden: "<<endl; arbol.inOrde n();</pre>			
--	---	--	--	--

quadtree:

Caso de Prueba	Valores de Entrada	Resultado Esperado	Resultado Obtenido	Estado (Éxito/Falla)
PreOrden	<pre>Arbol<int> arbol; pair<int, int> a = {5, 4}; pair<int, int> b = {3, 2}; pair<int, int> c = {4, 1}; pair<int, int> d = {8, 7}; pair<int, int> e = {1, 2}; pair<int, int> f = {9, 5}; arbol.insert ar(a); arbol.insert ar(b); arbol.insert ar(c); arbol.insert</pre>	<pre>(5, 4) (8, 7) (9, 5) (3, 2) (1, 2) (4, 1)</pre>	<pre>(5,4) (8,7) (9,5) (3,2) (1,2) (4,1)</pre>	✓ Éxito

	<pre> ar(d); arbol.insert ar(e); arbol.insert ar(f); cout<<endl; cout<<"Pre Orden: "<<endl; arbol.preOrd en(); </pre>			
PosOrden	<pre> Arbol<int> arbol; pair<int, int> a = {5, 4}; pair<int, int> b = {3, 2}; pair<int, int> c = {4, 1}; pair<int, int> d = {8, 7}; pair<int, int> e = {1, 2}; </pre>	(9, 5) (8, 7) (1, 2) (4, 1) (3, 2) (5, 4)	(9,5) (8,7) (1,2) (4,1) (3,2) (5,4)	<input checked="" type="checkbox"/> Éxito

```
pair<int,
int> f = {9,
5};

arbol.insert
ar(a);

arbol.insert
ar(b);

arbol.insert
ar(c);

arbol.insert
ar(d);

arbol.insert
ar(e);

arbol.insert
ar(f);

cout<<endl;

cout<<"Pos
Orden:
"<<endl;

arbol.posOrd
en();
```

Árbol Red Black:

Caso de Prueba	Valores de Entrada	Resultado Esperado	Resultado Obtenido	Estado (Éxito/Falla)
PreOrden	<pre> tree.insert(11); tree.insert(19); tree.insert(7); tree.insert(14); tree.insert(17); tree.insert(31); tree.insert(26); tree.insert(45); std::cout << "Red-Black Tree structure:" << std::endl; tree.printTr ee(); // Remove a node from the tree </pre>	17 11 7 14 26 19 31	17 11 7 14 26 19 31	 Éxito

	<pre> tree.remove(45); std::cout << "\nAfter deleting node 5:" << std::endl; tree.printTree(); std::cout << "PreOrder: "; tree.preOrderTraversal(); </pre>			
PosOrden	<pre> std::cout << "PostOrder: "; tree.postOrderTraversal(); </pre>	7 14 11 19 31 26 17	7 14 11 19 31 26 17	<input checked="" type="checkbox"/> Éxito
in order	<pre> std::cout << "\nInOrder: "; tree.inOrderTraversal(); </pre>	7 11 14 17 19 26 31	7 11 14 17 19 26 31	<input checked="" type="checkbox"/> Éxito

5.Conclusión:

Cada tipo de árbol tiene una estructura y propósito específicos:

- **Árbol General:** Es la forma más flexible, donde cada nodo puede tener cualquier número de hijos (no presenta balanceo propio).
- **Árbol Binario:** Cada nodo tiene a lo sumo dos hijos, facilitando operaciones eficientes (no presenta balanceo propio).
- **Árbol AVL:** Un árbol binario de búsqueda balanceado que mantiene alturas equilibradas para optimizar búsquedas, inserciones y eliminaciones.
- **Árbol de Expresión:** Representa expresiones matemáticas, donde los nodos internos son operadores y las hojas son operandos.
- **Árbol KD:** Usado en espacios multidimensionales para búsquedas eficientes, especialmente en gráficos y bases de datos.
- **Árbol Quad:** Divide el espacio en cuatro regiones, útil para indexación en imágenes y mapas.
- **Árbol Red Black:** Árbol binario de búsqueda auto-balanceado que mantiene equilibrio utilizando colores (rojo y negro) y reglas específicas. Garantiza una altura logarítmica, lo que permite búsquedas, inserciones y eliminaciones eficientes incluso en el peor caso.

Cada tipo de árbol tiene ventajas específicas según la aplicación, desde organizar datos hasta optimizar búsquedas y representar expresiones. Para el caso de los programas brindados algunos árboles requieren un mejor orden y cambios de variables para hacer más cómoda su comprensión, a su vez que corregir ciertos errores que presentan.