ACTIVIDAD 1

UNIVERSIDAD TECNOLÓGICA METROPOLITANA

CUARTO CUATRIMESTRE
TERCER PARCIAL
GRUPO: 4D - DSM

ASIGNATURA: ESTRUCTURA DE DATOS

INTEGRANTES:
KAIROS ISAI LIZARRAGA DIAZ
DOCENTE
MIRIAN CANCHE

Contenido

Introducción

Los árboles son una de las estructuras de datos más fundamentales en ciencias computacionales, útiles para organizar información jerárquica y realizar búsquedas eficientes. Dentro de los árboles, los árboles binarios destacan por su estructura sencilla y versatilidad en algoritmos. En este documento se exploran los tipos de árboles, los recorridos en árboles binarios, las herramientas disponibles en lenguajes como Python, Java y C#, y casos prácticos de implementación.

Tipos de arboles

Tipos de árboles binarios

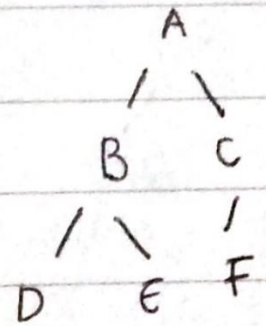
Árboles generales

- Estructura de datos no lineal que consiste en un conjunto de nodos organizados jerárquicamente. El nodo principal es la raíz, que puede tener múltiples hijos.

Características:

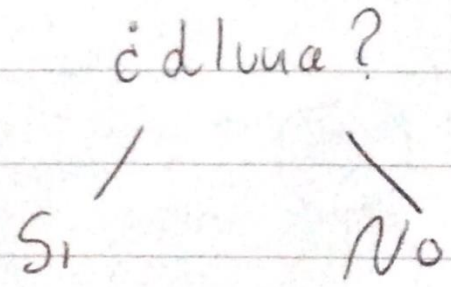
- la raíz es el nodo principal
- cada nodo puede tener más hijos

Ejemplo:



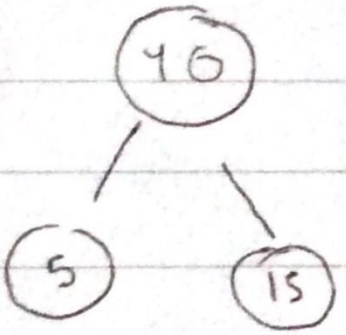
Árbol de decisión: Es una representación gráfica de decisión y sus posibles consecuencias. Cada nodo interno representa una decisión, cada rama representa el resultado de una decisión y cada hoja representa el resultado final.

Ejemplo:



- Árbol ordenado: Es una estructura en la que los nodos hijos están organizados según un orden específico. Este orden puede ser ascendente o descendente, dependiendo de la aplicación.

Ejemplo:



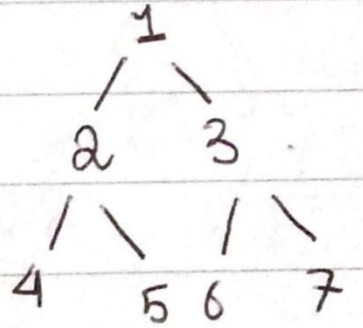
Árbol Binario Completo

tipo de árbol en el que dos niveles excepto posiblemente uno que es el último, están completamente llenos y están lo más a la izquierda posible

Características:

- lo más a la izquierda posible
- todos los nodos llenos

Ejemplo:



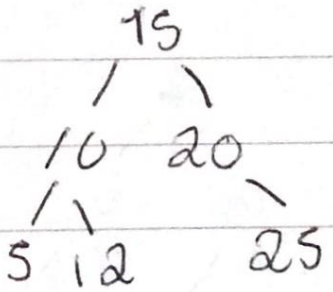
Árbol Binario de Búsqueda

Árbol binario en el que para cada nodo, los valores de todos los nodos en su árbol izquierdo son menores, y los valores de todos los nodos en su árbol derecho son mayores.

Características:

- Izquierda son menores
- derecha mayores

Ejemplo



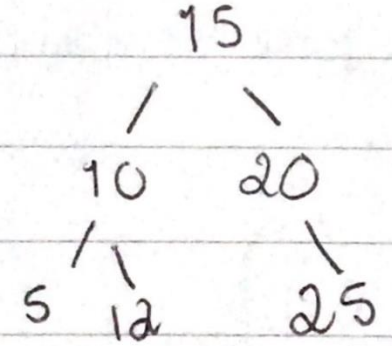
Árbol binario

- Tipo específico de árbol en el que cada nodo tiene, como máximo dos hijos. Estos hijos se denominan hijo izquerdo y derecho.

Características

- máximo 2 hijos
- Ideales para búsquedas

Ejemplo:



Tipos de recorridos

Recorrido en anchura

Recorre el árbol nivel por nivel comenzando desde la raíz. Este tipo de recorrido es útil cuando se quiere explorar todos los nodos de un nivel antes de pasar al siguiente nivel.

Algoritmo:

1) Colocar la raíz en una cola
2) Mientras la cola no este vacía
   - Sacar un nodo de la cola
   - Procesar el nodo
   - Agregar a sus hijos

Pasos del algoritmo

1) la cola contiene el nodo [15]

2) Sacamos 15 de la cola, procesamos 15 y añadimos sus hijos 5 y 12 a la cola

3) Sacamos 10, procesamos 10 y añadimos sus hijos 5 y 12

4) Sacamos 20, procesamos 20 y añadimos a su hijo 25 a la cola

5) Sacama 5, procesamos 5

6) Sacamos 12, procesamos 12, no tiene hijos

7) Sacamos 25, procesamos 25,

Resultado BFS: 15, 10, 20, 5, 12, 25

Recorrido en Preorden

Se procesa el nodo raiz primero, luego recorre el subárbol izquierdo y finalmente el subárbol derecho

Algoritmo:

1. procesar el nodo actual
2. Recorrer el subárbol izquierdo en preorden
3. Recorrer el subárbol derecho

Pasos:

1. Procesar 15
2. Recorrer el subárbol izquierdo: procesar 10,
3. Recorrer el subárbol derecho de 10: 5
4. Recorrer el subárbol derecho de 10: 12

Resultado: 15, 10, 5, 12, 20, 25

Recorrido en Inorden

Recorrido en inorden: primero recorre el subárbol izquierdo, luego procesa el nodo raíz y finalmente recorre el subárbol derecho.

Algoritmo:

1) recorrer el subárbol izquierdo
2) procesar el nodo actual
3) recorrer el subárbol derecho en inorden

Pasos

1. Recorrer el subárbol izq de 15
2. Recorrer el subárbol izq de 10
3. Procesar 10
4. Recorrer el subárbol der. de 10
5. Procesar 15
6. Recorrer el subárbol der. de 15

Resultado: 5, 10, 12, 15, 20, 25

Recorrido postorden

El recorrido postorden es una forma de recorrer los nodos de un árbol binario. En este tipo de recorrido, se visitan los nodos en el siguiente orden:

Algoritmo:

1. Primero se recorre el subárbol izq
2. Luego se recorre el subárbol der.
3. Finalmente, se procesa el nodo raíz

Pasos:

1. Identificar la estructura del árbol
2. Inicia en la raíz A
3. Desde B va a su arbol izquierdo
4. Regresa a B y va a su árbol derecho
5. Regresa a B y se procesa
6. Regresa a la raíz A y va a su árbol derecho C

Resultado: D, E, B, C, A

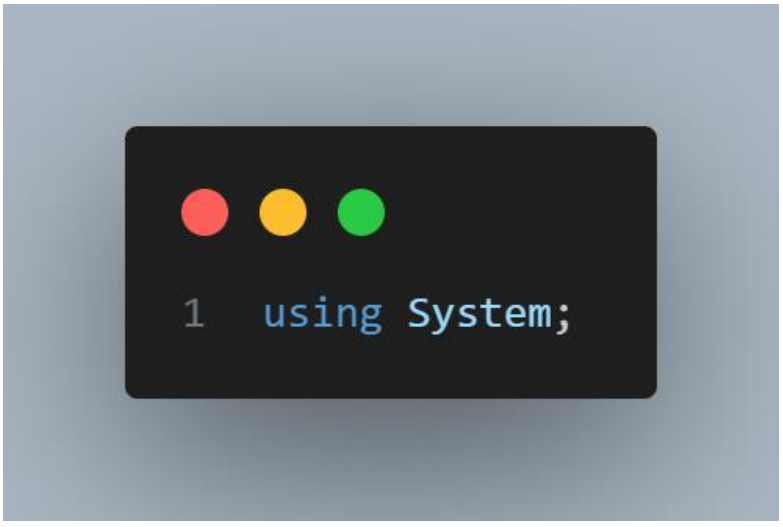
1. C#

En C#, puedes manejar árboles binarios utilizando estructuras de datos personalizadas o bibliotecas de terceros. Algunas bibliotecas útiles incluyen:

- **System.Collections.Generic**: En la biblioteca estándar de C#, se pueden usar clases como SortedDictionary<TKey, TValue> y SortedSet<T> para manejar datos de manera ordenada, aunque no implementan un árbol binario tradicional de forma explícita, pueden utilizarse para casos donde se requieran búsquedas rápidas, inserciones y eliminaciones.

- **C5 (C5 Collections Library)**: C5 es una poderosa colección de estructuras de datos que incluye árboles binarios, árboles de búsqueda balanceados (como Red-Black Trees) y más.

- **NDS (Node Data Structures)**: Esta es una biblioteca más sencilla para trabajar con árboles binarios y otras estructuras de datos.



2. Java

En Java, se pueden utilizar bibliotecas estándar o bibliotecas de terceros para trabajar con árboles binarios.

- **Java Collections Framework (JCF)**: Aunque JCF no tiene una implementación directa de árboles binarios, puedes usar clases como TreeSet y TreeMap, que implementan internamente un **Red-Black Tree**, un tipo de árbol binario balanceado.

- **Google Guava**: Guava incluye algunas colecciones avanzadas, pero no tiene una implementación directa de árboles binarios. Sin embargo, puede usarse junto con TreeMap de Java para construir árboles de manera eficiente.

- **Apache Commons Collections**: Esta biblioteca ofrece implementaciones adicionales de estructuras de datos que se pueden usar para manejar árboles binarios, incluyendo TreeList y TreeSet.

3. JavaScript

En JavaScript, no hay bibliotecas estándar para árboles binarios, pero puedes usar bibliotecas de terceros que implementan árboles binarios o crear una implementación personalizada.

- **Bintrees**: Una biblioteca que ofrece árboles binarios de búsqueda, así como otros tipos de árboles binarios balanceados.

- **JSDataStructures**: Esta es otra biblioteca útil para estructuras de datos, que incluye árboles binarios de búsqueda, entre otras cosas.

4. Python

En Python, existen varias bibliotecas que permiten trabajar con árboles binarios y otras estructuras de datos.

- **Binarytree**: Esta es una biblioteca de Python para la creación y manipulación de árboles binarios. Permite visualizar árboles, así como construir árboles binarios de búsqueda.

- **bintrees**: Aunque está descontinuada, esta biblioteca ofrece una implementación eficiente de árboles binarios balanceados como **Red-Black Trees** y **AVL Trees**.

- **sortedcontainers**: Una excelente alternativa para trabajar con árboles binarios balanceados en Python.

Caso 1: Sistema de Gestión de Inventarios (Árbol Binario de Búsqueda)

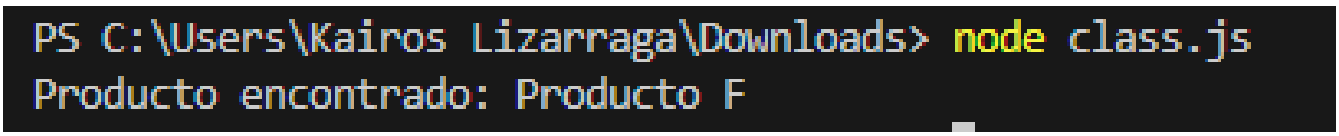
Imagina que estás implementando un sistema de gestión de inventarios para una tienda. Los productos deben estar ordenados por su código, y la tienda necesita un método rápido para buscar si un producto específico está disponible.

En este caso, utilizamos un **Árbol Binario de Búsqueda (BST)** para almacenar los productos, donde cada nodo representa un producto y el valor del nodo es el código del producto. Los productos estarán ordenados de menor a mayor según su código para optimizar la búsqueda

```
class ProductNode {
    constructor(code, name) {
        this.code = code;
        this.name = name;
        this.left = null;
        this.right = null;
    }
}

// Función para buscar un producto por su código
function searchProduct(node, targetCode) {
    if (!node) return null; // Producto no encontrado

    if (node.code === targetCode) return node.name; // Producto encontrado

    if (targetCode < node.code) {
        return searchProduct(node.left, targetCode); // Buscar en el subárbol izquierdo
    } else {
        return searchProduct(node.right, targetCode); // Buscar en el subárbol derecho
    }
}

// Construcción del árbol de productos
const root = new ProductNode(50, 'Producto A');
root.left = new ProductNode(30, 'Producto B');
root.right = new ProductNode(70, 'Producto C');
root.left.left = new ProductNode(20, 'Producto D');
root.left.right = new ProductNode(40, 'Producto E');
root.right.left = new ProductNode(60, 'Producto F');
root.right.right = new ProductNode(80, 'Producto G');

// Buscar producto por código
const targetCode = 60;
const product = searchProduct(root, targetCode);
console.log(product ? `Producto encontrado: ${product}` : 'Producto no encontrado');

```

Paso a Paso:

1. El árbol comienza en el nodo raíz (50).
2. Dado que **60 > 50**, se mueve al subárbol derecho (70).
3. Dado que **60 < 70**, se mueve al subárbol izquierdo (60).
4. Encuentra el producto: **"Producto F"**.



Caso 2: Sistema de Navegación de Rutas (Árbol Binario)

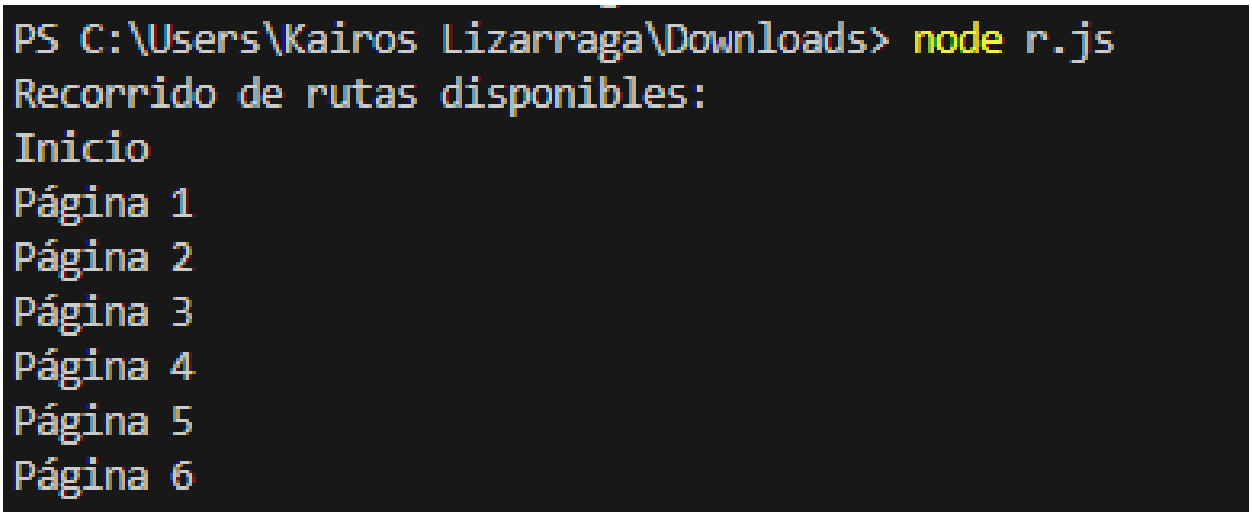
Imagina que estás diseñando un sistema de navegación para un sitio web o una aplicación móvil donde los usuarios deben seguir un conjunto de rutas para llegar a una página específica. Las rutas son jerárquicas, y cada página tiene un conjunto limitado de páginas hijas.

En este caso, utilizamos un **árbol binario** para representar las rutas, donde cada nodo representa una página, y las ramas representan las opciones para navegar a otras páginas.

```
class PageNode {
    constructor(name) {
        this.name = name;
        this.left = null;
        this.right = null;
    }
}

// Función para realizar un recorrido en anchura (nivel por nivel)
function breadthFirstSearch(root) {
    if (!root) return;
    let queue = [root]; // Cola para almacenar los nodos a procesar

    while (queue.length > 0) {
        let currentNode = queue.shift(); // Extraer el primer nodo de la cola
        console.log(currentNode.name); // Mostrar la página actual

        if (currentNode.left) queue.push(currentNode.left); // Añadir hijo izquierdo a la cola
        if (currentNode.right) queue.push(currentNode.right); // Añadir hijo derecho a la cola
    }
}

// Construcción del árbol de navegación
const root = new PageNode("Inicio");
root.left = new PageNode("Página 1");
root.right = new PageNode("Página 2");
root.left.left = new PageNode("Página 3");
root.left.right = new PageNode("Página 4");
root.right.left = new PageNode("Página 5");
root.right.right = new PageNode("Página 6");

// Recorrido en anchura (nivel por nivel)
console.log("Recorrido de rutas disponibles:");
breadthFirstSearch(root);

```

Paso a Paso:

1. Comienza desde el nodo raíz ("Inicio").
2. Primero, se visitan las páginas en el primer nivel ("Página 1" y "Página 2").
3. Luego, se visitan las páginas en el siguiente nivel ("Página 3", "Página 4", "Página 5", y "Página 6").
4. El recorrido es en **anchura**, lo que significa que procesamos todos los nodos de un nivel antes de pasar al siguiente.



Caso 3: Gestión de Categorías de Productos (Árbol Binario)

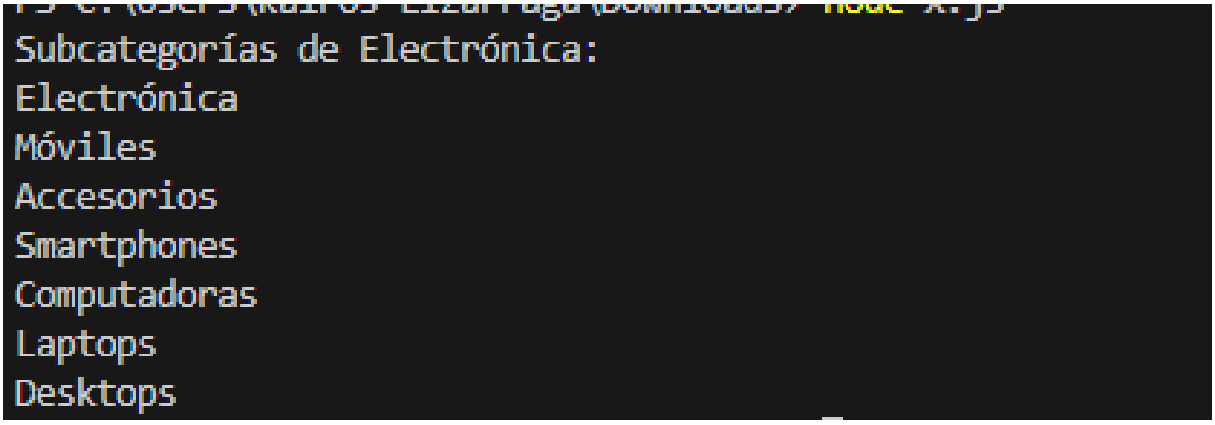
Imagina que tienes un sistema de comercio electrónico donde los productos están organizados en categorías. Las categorías son jerárquicas y cada categoría puede tener subcategorías de productos.

En este caso, un **árbol binario** puede representar esta jerarquía. Por ejemplo, la categoría principal puede ser "Electrónica", y debajo de ella hay subcategorías como "Móviles" y "Computadoras".

```
    constructor(name) {
        this.name = name;
        this.left = null;
        this.right = null;
    }
}

// Función para realizar un recorrido en preorden (de arriba hacia abajo)
function preOrderTraversal(node) {
    if (!node) return;
    console.log(node.name); // Mostrar la categoría actual
    preOrderTraversal(node.left); // Recorrer el subárbol izquierdo
    preOrderTraversal(node.right); // Recorrer el subárbol derecho
}

// Construcción del árbol de categorías
const root = new CategoryNode("Electrónica");
root.left = new CategoryNode("Móviles");
root.right = new CategoryNode("Computadoras");
root.left.left = new CategoryNode("Accesorios");
root.left.right = new CategoryNode("Smartphones");
root.right.left = new CategoryNode("Laptops");
root.right.right = new CategoryNode("Desktops");

// Recorrido en preorden para mostrar las subcategorías de Electrónica
console.log("Subcategorías de Electrónica:");
preOrderTraversal(root);

```

Paso a Paso:

1. Comienza en la categoría raíz ("Electrónica").
2. Se visita primero la categoría "Móviles", luego sus subcategorías ("Accesorios" y "Smartphones").
3. Luego, se pasa a la categoría "Computadoras" y sus subcategorías ("Laptops" y "Desktops").
4. El recorrido es en **preorden**, lo que significa que primero se visita la categoría y luego sus subcategorías.



Conclusión

Los **árboles** son estructuras jerárquicas fundamentales en la informática, ampliamente utilizadas para organizar datos y facilitar su acceso eficiente. Los distintos **tipos de árboles** (binarios, binarios de búsqueda, AVL, B-Trees, entre otros) están diseñados para resolver problemas específicos, desde búsquedas rápidas hasta la optimización del espacio en almacenamiento.

Los **recorridos de árboles** (preorden, inorden y postorden) son métodos clave para procesar los nodos y extraer información de ellos. Cada tipo de recorrido tiene un propósito único:

- **Preorden**: Procesa la raíz antes de sus hijos, útil para clonar estructuras o evaluar expresiones.
- **Inorden**: Procesa los nodos en orden ascendente (en árboles binarios de búsqueda), ideal para obtener datos ordenados.
- **Postorden**: Procesa la raíz después de sus hijos, usado frecuentemente en eliminación de nodos o evaluaciones de expresiones complejas.

En resumen, la elección del tipo de árbol y el método de recorrido depende del problema que se busca resolver. Comprender estas herramientas permite diseñar soluciones más eficientes, adaptadas a las necesidades de cada aplicación.

Referencias

Libros

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to algorithms* (3ra ed.). MIT Press.

Knuth, D. E. (1998). *The art of computer programming: Fundamental algorithms* (Vol. 1, 3ra ed.). Addison-Wesley.

Shaffer, C. A. (2013). *Data structures and algorithm analysis* (3ra ed.). Virginia Tech. Recuperado de https://people.cs.vt.edu/shaffer/Book/

Artículos y Recursos en Línea

GeeksforGeeks. (s.f.). *Tree Traversals (Inorder, Preorder and Postorder)*. Recuperado de https://www.geeksforgeeks.org/tree-traversals-inorder-preorder-and-postorder/

Programiz. (s.f.). *Binary Tree Data Structure*. Recuperado de https://www.programiz.com/dsa/binary-tree

TutorialsPoint. (s.f.). *Binary Tree - Data Structure*. Recuperado de https://www.tutorialspoint.com/data_structures_algorithms/tree_data_structure.html

Videos Educativos

FreeCodeCamp. (2020). *Tree Traversals in Data Structures* [Video]. YouTube. Recuperado de https://www.youtube.com/watch?v=fAAZixBzIAI

MyCodeSchool. (2013). *Tree Traversals Explained* [Video]. YouTube. Recuperado de https://www.youtube.com/watch?v=gm8DUJJhmY4