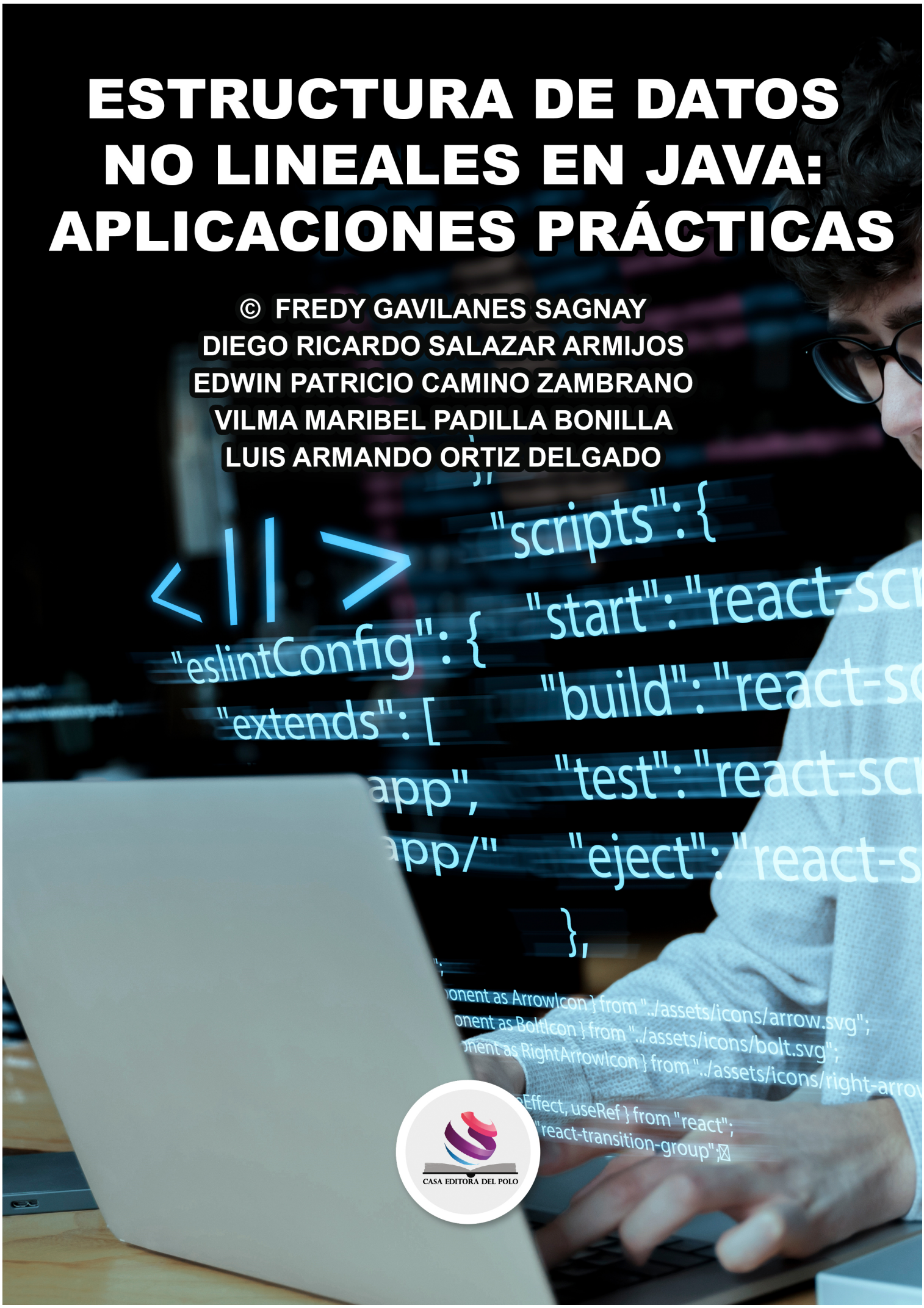
ESTRUCTURA DE DATOS NO LINEALES EN JAVA: APLICACIONES PRÁCTICAS

© FREDY GAVILANES SAGNAY
DIEGO RICARDO SALAZAR ARMIJOS
EDWIN PATRICIO CAMINO ZAMBRANO
VILMA MARIBEL PADILLA BONILLA
LUIS ARMANDO ORTIZ DELGADO



CASA EDITORA DEL POLO

ESTRUCTURA DE DATOS NO LINEALES EN JAVA: APLICACIONES PRÁCTICAS

Fredy Gavilanes-Sagnay
Diego Ricardo Salazar Armijos
Edwin Patricio Camino Zambrano
Vilma Maribel Padilla Bonilla
Luis Armando Ortiz Delgado

CASA EDITORA DEL POLO

Estructura de datos no lineales en java: Aplicaciones prácticas

Fredy Gavilanes-Sagnay
Departamento en Informática y Ciencias de la Computación, Docente de la **Escuela Politécnica Nacional. Quito, Ecuador.**

Diego Ricardo Salazar Armijos
Departamento de Ciencias de la Computación Santo Domingo, Docente de la Universidad de las Fuerzas Armadas ESPE

Edwin Patricio Camino Zambrano
Departamento de Ciencias de la Computación Santo Domingo, Docente de la Universidad de las Fuerzas Armadas ESPE

Vilma Maribel Padilla Bonilla
Docente de la Unidad Educativa Luis A Martínez

Luis Armando Ortiz Delgado
Departamento de Ciencias de la Computación, Docente de la Universidad de las Fuerzas Armadas-ESPE

CASA EDITORA DEL POLO

Casa Editora del Polo - CASEDELPO CIA. LTDA.
Departamento de Edición

Editado y distribuido por:

Editorial: Casa Editora del Polo
Sello Editorial: 978-9942-816
Manta, Manabí, Ecuador. 2019
Teléfono: (05) 6051775 / 0991871420
Web: www.casedelpo.com
ISBN: 978-9942-621-74-0
DOI: https://doi.org/10.23857/978-9942-621-74-0

© Primera edición
© Mayo - 2024
Impreso en Ecuador

Revisión, Ortografía y Redacción:
Lic. Jessica Mero Vélez

Diseño de Portada:
Michael Josué Suárez-Espinar

Diagramación:
Ing. Edwin Alejandro Delgado-Veliz

Director Editorial:
Lic. Henry Darío Suárez-Vélez

Constancia de Arbitraje

La Casa Editora del Polo, hace constar que este libro proviene de una investigación realizada por los autores, siendo sometido a un arbitraje bajo el sistema de doble ciego (peer review), de contenido y forma por jurados especialistas. Además, se realizó una revisión del enfoque, paradigma y método investigativo; desde la matriz epistémica asumida por los autores, aplicándose las normas APA, Sexta Edición, proceso de anti plagio en línea Plagiarisma, garantizándose así la cientificidad de la obra.

Comité Editorial

Contenido

PRÓLOGO

En el vasto y dinámico campo de la ciencia computacional, la eficiente manipulación y gestión de datos son aspectos fundamentales. En este contexto, las estructuras de datos no lineales juegan un papel crucial al permitirnos representar y organizar datos de manera flexible y eficiente. Desde los tradicionales árboles binarios hasta las más complejas estructuras como los grafos y los heaps de Fibonacci, estas estructuras ofrecen una amplia gama de herramientas para abordar una variedad de problemas computacionales.

Este libro sobre estructuras de datos no lineales es una guía completa y exhaustiva que explora los fundamentos teóricos y prácticos de estas estructuras, proporcionando a los lectores un profundo entendimiento de sus características, aplicaciones y algoritmos asociados. Desde estudiantes universitarios hasta profesionales de la informática, este libro está diseñado para ser una referencia invaluable para cualquiera que desee dominar este fascinante campo.

El libro comienza con una introducción a las estructuras de datos no lineales, presentando los conceptos básicos y motivando su importancia en la ciencia computacional moderna. A partir de ahí, se adentra en diversas estructuras, como los árboles binarios, los árboles de búsqueda balanceados, los grafos y los heaps, explorando sus propiedades, operaciones y algoritmos asociados en detalle.

A medida que avanza, el libro aborda temas más avanzados, como la optimización de operaciones, el diseño eficiente de algoritmos y las aplicaciones prácticas en campos como la inteligencia artificial, la bioinformática y la optimización combinatoria. Cada capítulo está cuidadosamente elaborado con ejemplos ilustrativos, pseudocódigos claros y ejercicios prácticos para ayudar a los lectores a consolidar su comprensión y aplicar los conocimientos adquiridos.

El objetivo de este libro no es solo proporcionar información teórica, sino también fomentar un enfoque práctico y orientado a problemas en el análisis y diseño de algoritmos para estructuras de datos no lineales. A través de esta combinación de teoría y práctica, los lectores serán capaces de desarrollar habilidades sólidas en el diseño y aplicación de estructuras de datos no lineales para resolver una amplia gama de desafíos computacionales.

Esperamos que este libro sea una herramienta útil y valiosa para estudiantes, académicos e ingenieros en su viaje hacia el dominio de las estructuras de datos no lineales y su aplicación en la resolución de problemas del mundo real.

CAPÍTULO I

INTRODUCCIÓN A LAS ESTRUCTURAS DE DATOS NO LINEALES

Las estructuras de datos son fundamentales en el campo de la informática, ya que proporcionan una manera de organizar y almacenar datos eficientemente. A diferencia de las estructuras lineales, como los arrays y listas enlazadas, que organizan los datos de manera secuencial, las estructuras de datos no lineales, tales como árboles y grafos, presentan relaciones más complejas entre los elementos (Goodrich et al., 2014). Estas estructuras permiten representar jerarquías, redes y otras relaciones complejas que son cruciales en el desarrollo de algoritmos y aplicaciones avanzadas.

Las estructuras de datos no lineales se caracterizan por su capacidad para representar relaciones múltiples entre sus elementos, a diferencia de una secuencia lineal simple. Esta característica es esencial para modelar diversas formas de datos complejos que encontramos en aplicaciones reales, como las redes sociales, sistemas de navegación y organización de la información (Cormen et al., 2009).

Los árboles son una de las estructuras de datos no lineales más fundamentales. Un árbol está compuesto por nodos conectados por enlaces o aristas, donde un único nodo sin padre, conocido como la raíz, da origen a una jerarquía de nodos hijos. Los árboles se utilizan ampliamente para representar relaciones jerárquicas, como estructuras organizacionales, sistemas de archivos y bases de datos (Silberschatz et al., 2020).

Los grafos extienden la idea de estructuras no lineales para incluir un conjunto aún más amplio de relaciones, permitiendo representar redes de datos. Un grafo consiste en un conjunto de vértices conectados por aristas, que pueden ser dirigidas o no dirigidas. Los grafos son fundamentales en el campo de la informática y tienen aplicaciones en la modelación de redes de comunicaciones, algoritmos de navegación y la teoría de grafos es un área de estudio en constante evolución en ciencias de la computación (Bang-Jensen & Gutin, 2009).

Los heaps, o montículos, son otra forma de estructura de datos no lineal que se utilizan para implementar colas de prioridad. Son árboles binarios que cumplen con la propiedad de montículo, lo que permite el acceso y eliminación eficiente del elemento mínimo o máximo. Este tipo de estructura de datos es crucial en algoritmos de ordenación y sistemas de gestión de colas (Knuth, 1998).

Las estructuras de datos no lineales son fundamentales en la solución de problemas complejos en informática. Por ejemplo, los árboles binarios de búsqueda permiten la búsqueda eficiente de elementos, mientras que los grafos son esenciales en la planificación de rutas y análisis de redes. La comprensión profunda de estas estructuras es crucial para los científicos de la computación que desarrollan nuevas herramientas y tecnologías (Sedgewick & Wayne, 2011).

El estudio y la implementación de estructuras de datos no lineales presentan desafíos únicos, especialmente en términos de eficiencia de algoritmos y manejo de grandes volúmenes de datos. La investigación continua en esta área promete innovaciones en cómo procesamos y analizamos datos complejos, con un potencial significativo para avances tecnológicos (Leiserson et al., 2020).

1.1. Definición y características de las Estructuras de Datos No Lineales

Las estructuras de datos son fundamentos esenciales para la informática, ya que permiten organizar, almacenar y gestionar datos de manera eficiente. Mientras que las estructuras lineales como las listas y colas organizan los datos de manera secuencial, las estructuras de datos no lineales presentan una disposición que permite representar relaciones más complejas entre los elementos. Estas estructuras, incluyendo árboles y grafos, son cruciales para modelar jerarquías y redes, soportando así algoritmos complejos y aplicaciones de software avanzadas (Goodrich, Tamassia, & Goldwasser, 2014).

1.1.1. Características Principales

Las estructuras de datos no lineales se distinguen por varias características clave:

Relaciones Jerárquicas:

Los árboles, una forma común de estructuras no

lineales, representan relaciones jerárquicas donde elementos subordinados (hijos) están vinculados a elements superiores (padres) en un modelo no secuencial (Cormen, Leiserson, Rivest, & Stein, 2009).

Conexiones Complejas:

Los grafos permiten representar redes de datos con conexiones complejas entre nodos, incluyendo relaciones bidireccionales y ponderaciones para modelar caminos y redes (Bang-Jensen & Gutin, 2009).

Almacenamiento No Secuencial:

A diferencia de las listas o colas, donde el acceso a los datos suele ser secuencial, las estructuras de datos no lineales pueden permitir acceso directo a múltiples elementos simultáneamente, gracias a su organización inherentemente no secuencial (Sedgewick & Wayne, 2011).

Eficiencia en Operaciones Específicas:

Las operaciones como la búsqueda, inserción y eliminación pueden ser más eficientes en ciertas estructuras de datos no lineales, debido a la posibilidad de dividir y conquistar o saltar directamente a subconjuntos de datos, como se ve en la búsqueda binaria en árboles binarios de búsqueda (Knuth, 1998).

1.1.2. Tipos de Estructuras de Datos No Lineales

Árboles: Organizan los nodos en un sistema jerárquico, permitiendo representar estructuras de datos con relaciones de uno a muchos. Los árboles binarios, los árboles AVL, y los árboles rojo-negro son ejemplos clave que facilitan operaciones de búsqueda y ordenamiento eficiente.

Grafos: Ideal para modelar relaciones complejas entre pares de elementos con estructuras que pueden ser cíclicas o acíclicas, dirigidas o no dirigidas. Los grafos son fundamentales para algoritmos de redes, caminos mínimos y planificación de rutas.

Importancia y Aplicaciones

Las estructuras de datos no lineales son esenciales en la informática y la ingeniería de software, ofreciendo soluciones óptimas para una variedad de problemas complejos. La organización de información en bases de datos, el análisis de redes sociales, y los algoritmos de búsqueda en Internet son solamente algunas de las aplicaciones donde estas estructuras son fundamentales. Además, su capacidad para representar de manera eficiente relaciones complejas hace que sean herramientas indispensables en la ciencia de datos, inteligencia artificial y la modelización de sistemas complejos (Leiserson, Rivest, Cormen, & Stein, 2020).

Desafíos y Consideraciones

A pesar de sus ventajas, trabajar con estructuras de datos no lineales presenta desafíos. La complejidad en su diseño e implementación, especialmente en lo que respecta a garantizar la eficiencia y estabilidad de las operaciones sobre grandes volúmenes de datos, es una consideración importante. Además, el análisis y optimización de algoritmos que operan sobre estas estructuras requieren un entendimiento profundo de teoría computacional y matemática aplicada (Aho, Ullman, & Hopcroft, 1983).

1.2. Importancia y Aplicaciones

Las estructuras de datos no lineales, como árboles y grafos, son fundamentales en el dominio de la informática, permeando numerosas aplicaciones que van desde el diseño de algoritmos eficientes hasta el modelado de relaciones complejas en datos de la vida real (Cormen, Leiserson, Rivest, & Stein, 2009). Su importancia se deriva principalmente de su capacidad para organizar datos de una manera que refleje relaciones específicas y proporcione acceso eficiente a ellos, una necesidad prevalente en muchas áreas tecnológicas y científicas.

1.2.1. Importancia en Ciencias de la Computación

La capacidad de representar datos de manera jerárquica o en red permite resolver problemas computacionales complejos que serían ineficientes o incluso imposibles

de manejar usando estructuras de datos lineales. Los árboles binarios, por ejemplo, son críticos para la búsqueda y ordenación eficiente, fundamentales en la programación de bases de datos y sistemas de archivos (Knuth, 1998). Los grafos, por otro lado, son esenciales en el análisis de redes, desde la planificación de redes de transporte hasta la modelización de redes sociales y sistemas de recomendaciones (Newman, 2010).

1.2.2. Aplicaciones en el Mundo Real

Bases de Datos:

Las estructuras de datos no lineales son cruciales en el diseño e implementación de sistemas de gestión de bases de datos. Los árboles B y árboles B+, en particular, se utilizan para almacenar información que se puede buscar, insertar o eliminar eficientemente (Sedgewick & Wayne, 2011).

Sistemas de Navegación y Mapas:

El uso de grafos permite modelar redes viales y rutas, facilitando la programación de algoritmos que encuentran caminos óptimos, esenciales para los sistemas de navegación y mapas digitales (Dijkstra, 1959).

Procesamiento de Lenguaje Natural (PLN):

Los árboles sintácticos, que son un tipo de estructura de datos no lineal, juegan un papel importante en el PLN al ayudar en la descomposición y análisis de la estructura

gramatical de los lenguajes (Jurafsky & Martin, 2009).

Inteligencia Artificial (IA) y Aprendizaje Automático:

En IA, las estructuras de datos no lineales, como los árboles de decisión, son vitales para construir y entrenar modelos que pueden realizar clasificaciones o predicciones basadas en datos de entrada complejos (Breiman, 2001).

Desafíos en la Implementación

A pesar de su amplia aplicación y beneficios, las estructuras de datos no lineales también presentan desafíos significativos en términos de su implementación y optimización. La complejidad de la gestión de la memoria, el balanceo de árboles para garantizar operaciones eficientes y la necesidad de algoritmos sofisticados para el análisis de grafos son ejemplos de tales desafíos (Cormen et al., 2009).

1.3. Comparación con las Estructuras Lineales

Comparar estructuras de datos lineales con no lineales implica examinar cómo cada tipo organiza, almacena y permite el acceso a los datos. Echemos un vistazo a las diferencias fundamentales, ventajas y desventajas de cada una, así como sus aplicaciones típicas.

1.3.1. Organización y Almacenamiento

Lineales: Las estructuras lineales, como los arrays, listas enlazadas, pilas y colas, organizan los datos de

manera secuencial. En estas estructuras, cada elemento está en una posición que tiene un siguiente y, excepto para el primero y último, un elemento previo. Esto facilita el acceso secuencial a los datos, pero puede limitar la eficiencia al realizar búsquedas o actualizaciones, especialmente en listas largas.

No Lineales: Las estructuras de datos no lineales, como los árboles y grafos, organizan los datos de manera jerárquica o en redes. Permiten representar relaciones complejas, como jerarquías, redes sociales o mapas de rutas. Estas estructuras son excelentes para búsquedas rápidas, inserciones, y eliminaciones, gracias a su capacidad para segmentar los datos y reducir el espacio de búsqueda.

1.3.2. Complejidad

Lineales: En general, las estructuras lineales son más simples de implementar y entender. La inserción y eliminación pueden ser rápidas para estructuras como las listas enlazadas, pero la búsqueda puede ser lenta, especialmente si la lista no está ordenada.

No Lineales: Las estructuras no lineales, aunque más complejas en su implementación, ofrecen mayores beneficios en términos de eficiencia para operaciones específicas gracias a su estructura organizativa. Por ejemplo, los tiempos de búsqueda, inserción y eliminación en un árbol binario de búsqueda, si está bien balanceado, son significativamente más rápidos que en una lista.

1.3.3. Uso de Memoria

Lineales: Tienen el beneficio de ser compactas, especialmente los arrays, que almacenan datos de manera contigua, facilitando el acceso por índice. Sin embargo, el redimensionamiento (como en el caso de los arrays dinámicos) puede ser costoso.

No Lineales: Pueden ser más eficientes en términos de memoria debido a su estructura, especialmente cuando se trata de representar conjuntos de datos complejos. Sin embargo, incurren en un sobrecosto de memoria debido a la necesidad de almacenar punteros adicionales (como los punteros a hijos en árboles binarios).

1.3.4. Aplicaciones Típicas

Lineales: Son ideales para aplicaciones que requieren acceso secuencial, como implementaciones de buffers, colas de impresión, o algoritmos de procesamiento de secuencias. También son preferidas cuando se necesita un acceso rápido a elementos a través de índices, como en el caso de arrays.

No Lineales: Son esenciales en aplicaciones que requieren representar relaciones complejas y realizar búsquedas optimizadas, como bases de datos (por ejemplo, índices B-tree), manipulación de datos jerárquicos (como sistemas de archivos), y algoritmos para rutas óptimas en redes de transporte o mapas (grafos).

1.3.5. Ventajas y Desventajas

Lineales: La principal ventaja es su simplicidad y la eficiencia en el acceso por índice. La desventaja es que no son óptimas para representar relaciones complejas o realizar operaciones de inserción y eliminación eficientes en conjuntos de datos grandes sin un orden predefinido.

No Lineales: Su ventaja radica en la eficiencia de operaciones específicas como búsqueda y actualización en conjuntos de datos con estructuras complejas. La principal desventaja es la complejidad en su implementación y el mantenimiento del balance o la estructura óptima para asegurar eficiencia.

1.3.6. Ejemplo de Encapsulación en Java

Para ilustrar la encapsulación en Java, consideremos un ejemplo de una clase “Cuenta Bancaria” que encapsula información sobre una cuenta bancaria y proporciona métodos para acceder y modificar esa información. Los atributos privados de la clase podrían incluir el número de cuenta, el saldo y el titular de la cuenta.

Clase Cuenta Bancaria

```
public class CuentaBancaria {
   private int numeroCuenta;
   private double saldo;
   private String titular;
   public void depositar(double monto) {
      // Lógica para realizar el depósito
   public void retirar(double monto) {
      // Lógica para realizar el retiro
   }
   public double obtenerSaldo() {
      return saldo;
   }
}
```

1.1.1. Aplicación 2:

Implementación de una Tipo de Dato Abstracto en Java denominada Cola con un TDA Automóviles

Clase Cola Automoviles

```
import java.util.LinkedList;
import java.util.Queue;
public class ColaDeAutomoviles {
   public static void main(String[] args) {
      // Crear una cola de automóviles
      Queue<String> colaDeAutomoviles = new LinkedList<>();
      // Agregar automóviles a la cola
      colaDeAutomoviles.offer("Automóvil 1");
      colaDeAutomoviles.offer("Automóvil 2");
      colaDeAutomoviles.offer("Automóvil 3");
      colaDeAutomoviles.offer("Automóvil 4");
      colaDeAutomoviles.offer("Automóvil 5");
      // Imprimir la cola de automóviles
      System.out.println("Cola de automóviles: " + colaDeAutomoviles);
      // Obtener y eliminar el automóvil del frente de la cola
      String automovilFrente = colaDeAutomoviles.poll();
      System.out.println("Automóvil del frente: " + automovilFrente);
      // Imprimir la cola de automóviles después de eliminar el automóvil del frente
        System.out.println("Cola de automóviles después de eliminar el automóvil del
frente: " + colaDeAutomoviles);
      // Obtener el automóvil del frente sin eliminarlo
      String automovilFrenteSinEliminar = colaDeAutomoviles.peek();
            System.out.println("Automóvil del frente sin eliminarlo: " +
automovilFrenteSinEliminar);
      // Imprimir la cola de automóviles nuevamente
      System.out.println("Cola de automóviles actualizada: " + colaDeAutomoviles);
   }
}
```


En este programa, se utiliza la clase LinkedList de Java como la implementación de la cola. Primero, se crea una instancia de LinkedList llamada colaDeAutomoviles, que representa la cola de automóviles.

Luego, se agregan algunos automóviles a la cola utilizando el método offer, que agrega el elemento al final de la cola.

Después de eso, se imprime la cola de automóviles utilizando el método toString() de la lista enlazada.

A continuación, se utiliza el método poll para obtener y eliminar el automóvil del frente de la cola. El automóvil eliminado se asigna a la variable automovilFrente.

Después de eliminar el automóvil del frente, se imprime la cola de automóviles actualizada.

Luego, se utiliza el método peek para obtener el automóvil del frente sin eliminarlo. El automóvil del frente se asigna a la variable automovilFrenteSinEliminar.

Finalmente, se imprime la cola de automóviles actualizada una vez más.

CAPÍTULO II

ESTRUCTURA DE DATOS NO LINEALES: ÁRBOLES

El término “árbol de datos” se refiere a una estructura de datos fundamental en el campo de las ciencias computacionales, utilizada para organizar información de manera jerárquica. Los árboles permiten representaciones eficientes de relaciones parentesco, crucial para operaciones de búsqueda, inserción y borrado. Diversos tipos, como árboles binarios, AVL, árboles rojo-negro y B-trees, son piedras angulares para el diseño de algoritmos y sistemas (Cormen, Leiserson, Rivest, & Stein, 2009).

Un árbol de datos es una colección de nodos conectados por aristas. Solo existe un único camino entre dos nodos, evitando cualquier ciclo. Cada árbol tiene un nodo raíz desde el cual se derivan más nodos. Los nodos sin hijos se conocen como hojas. La profundidad de un nodo se mide por la longitud del camino desde la raíz hasta el nodo (Knuth, 1998).

Los árboles de datos encuentran aplicación en un amplio rango de áreas, incluyendo:

Bases de Datos: Los B-trees son cruciales para indexar en bases de datos, permitiendo búsqueda, inserción y borrado eficientes de grandes volúmenes de datos (Comer, 1979).

Sistemas de Archivos: Los árboles facilitan la organización y acceso rápido a los archivos y directorios en un sistema de archivos.

Procesamiento de Lenguaje Natural: Árboles sintácticos permiten analizar y entender estructuras gramaticales.

Gráficos y Visualización: Los árboles son usados en la representación de jerarquías y estructuras organizacionales.

• Ventajas de Usar Árboles de Datos

Eficiencia: Los árboles permiten búsquedas, inserciones y borrados rápidos, especialmente en estructuras balanceadas.

Flexibilidad: Se pueden modificar para resolver una variedad de problemas computacionales.

Intuitivos: La estructura jerárquica de un árbol es fácil de entender y visualizar.

• Desafíos en el Uso de Árboles

Mantenimiento: Los árboles, especialmente los balanceados, requieren cuidado en su mantenimiento para asegurar su eficiencia operacional.

Complejidad de Implementación: Algunos tipos de árboles, como los AVL y rojo-negro, tienen reglas complejas para su balanceo.

2.1. Árboles Generales

Empieza describiendo qué es un árbol en el contexto de las estructuras de datos. Un árbol es una colección de nodos que simula una jerarquía: una sola entidad

conocida como la "raíz" da origen a más nodos, formando relaciones padres-hijos con reglas específicas (Cormen, Leiserson, Rivest, & Stein, 2009). Los árboles generales son un tipo específico de árbol donde no hay restricciones en el número de hijos que un nodo puede tener, a diferencia de los árboles binarios.

2.1.1. Definición de Árboles Generales

Un árbol general es una estructura de datos no lineal que permite que cada nodo tenga cualquier número de hijos. Se contrasta con los árboles binarios e introduce la idea base para estructuras más complejas como árboles B y árboles B+ usados en bases de datos y sistemas de archivos (Knuth, 1998).

2.1.2. Características de los Árboles Generales

Nodos: Incluyen una raíz, nodos internos y hojas.

Raíz: El único nodo sin padre.

Nodos Internos: Nodos con al menos un hijo.

Hojas: Nodos sin hijos.

Altura: La longitud del camino más largo desde la raíz hasta una hoja.ç

Profundidad: La longitud del camino desde la raíz hasta el nodo.

2.1.3. Representación de Árboles Generales

Listas de Adyacencia: Representación directa de la estructura del árbol con listas anidadas.

Arrays de Hermanos-Padre: Representación indirecta que mantiene vínculos entre nodos y sus hermanos.

2.1.4. Operaciones Básicas con Árboles Generales

Descripción de operaciones como inserción, búsqueda y eliminación. Cada operación en un árbol general puede requerir un enfoque diferente comparado con árboles binarios, debido a la naturaleza no restringida del número de hijos (Goodrich, Tamassia, & Goldwasser, 2014).

2.1.5. Aplicaciones de Árboles Generales

Representación de Estructuras Jerárquicas: Como sistemas de archivos donde directorios pueden contener un número variado de archivos o subdirectorios.

Sistemas de Gestión de Bases de Datos: Árboles generales proporcionan los fundamentos para algoritmos de indexación eficientes mediante árboles B y B+.

Árboles de Análisis: Usados en compiladores para representar la estructura de programas.

2.1.6. Ventajas de los Árboles Generales

Incluye ventajas como flexibilidad en la representación de datos y eficiencia para operaciones que involucran

recorridos jerárquicos o representaciones naturales de estructuras como organigramas.

2.1.7. Desafíos y Limitaciones

Manejo de Memoria: Es más complejo debido a la variabilidad en el número de hijos.

Complejidad de Algoritmos: Algunas operaciones pueden ser menos eficientes que en árboles binarios debido a la necesidad de manejar múltiples hijos.

2.2. Árboles Binarios

Los árboles binarios son una estructura de datos fundamental en informática que permite organizar y almacenar datos de manera jerárquica. En este artículo, exploraremos en profundidad qué son los árboles binarios, cómo funcionan, sus características principales, así como sus aplicaciones en diversos campos de la informática.

2.2.1. Definición y Características

Un árbol binario es una estructura de datos jerárquica en la que cada nodo tiene como máximo dos hijos, denominados hijo izquierdo y hijo derecho. El nodo superior se conoce como nodo raíz, mientras que los nodos que no tienen hijos se denominan nodos hoja. Cada nodo puede contener un valor o dato, y los nodos están conectados mediante enlaces o punteros.

La característica clave de los árboles binarios es que

cada nodo puede tener como máximo dos hijos, lo que los hace ideales para representar relaciones de "padre-hijo" en estructuras jerárquicas. Además, los árboles binarios pueden ser de búsqueda o de expresión, dependiendo de su aplicación específica.

Según Cormen, Leiserson, Rivest y Stein (2009), "un árbol binario es una estructura de datos jerárquica en la que cada nodo tiene como máximo dos hijos, conocidos como hijo izquierdo y hijo derecho".

2.2.2. Funcionamiento y Operaciones

Los árboles binarios admiten una variedad de operaciones fundamentales, incluyendo la inserción, eliminación, búsqueda y recorrido de nodos.

Inserción: Para insertar un nuevo nodo en un árbol binario, se comparan los valores del nuevo nodo con los nodos existentes y se decide si debe colocarse a la izquierda o a la derecha del nodo actual. Este proceso continúa hasta que se encuentra un nodo vacío.

Eliminación: La eliminación de un nodo en un árbol binario puede ser un poco más complicada que la inserción. Se deben considerar diferentes casos, como si el nodo a eliminar tiene cero, uno o dos hijos. Después de eliminar el nodo, el árbol debe reorganizarse para mantener su propiedad de árbol binario.

Búsqueda: La búsqueda en un árbol binario sigue un proceso similar a la inserción. Se compara el valor

buscado con el valor de los nodos actuales y se decide en qué dirección continuar la búsqueda, ya sea hacia la izquierda o hacia la derecha, hasta que se encuentre el nodo deseado o se llegue a un nodo vacío.

Recorrido: Los árboles binarios admiten varios métodos de recorrido para visitar todos los nodos del árbol en un orden específico, como el recorrido en orden, el recorrido en preorden y el recorrido en postorden.

2.2.3. Aplicaciones

Los árboles binarios tienen una amplia variedad de aplicaciones en informática, incluyendo:

Árboles de Búsqueda Binaria (BST): Los árboles binarios de búsqueda son utilizados en aplicaciones donde se requiere una búsqueda eficiente de elementos. Por ejemplo, en bases de datos, compiladores y algoritmos de búsqueda.

Árboles de Expresión: En compiladores y evaluadores de expresiones matemáticas, los árboles binarios se utilizan para representar la estructura de las expresiones aritméticas de manera jerárquica.

Árboles AVL y Árboles Rojo-Negro: Estas son variantes de árboles binarios que mantienen un equilibrio específico entre las alturas de los subárboles para garantizar un rendimiento óptimo en operaciones como inserción, eliminación y búsqueda.

Según Horowitz, Sahni y Anderson-Freed (2013), "los árboles binarios son utilizados en una amplia variedad de aplicaciones informáticas, desde la búsqueda eficiente de elementos hasta la representación de expresiones matemáticas".

Aplicación 1

Clase NodoArbol

```
package ed_tda13_arboles_binarios;

/**
 *
 * @author Marcelo
 */
public class NodoArbol {
   int dato;
   String nombre;
   NodoArbol hijoIzquierdo, hijoDerecho;
   //Constructor
   public NodoArbol(int d, String nom){
      this.dato=d;
      this.nombre=nom;
      this.hijoDerecho=null;
      this.hijoIzquierdo=null;
   }
}
```

Clase NodoArbol

```
package ed_tda13_arboles_binarios;


/**
 *
 * @author Marcelo
 */
public class ArbolBinario {
   NodoArbol raiz;
   //constructor del Arbol BInanrio
```



```
  public ArbolBinario(){
     raiz=null;
  }
  //IMplementar el método de inserción de elementos del Arbol Binario
   public void agregarNodo(int d, String nom){
     NodoArbol nuevo=new NodoArbol(d, nom);
     if (raiz==null){
        raiz=nuevo;
     }else{
        NodoArbol aux=raiz;
        NodoArbol padre;
        while (true){
           padre=aux;
           if (d<aux.dato){ ///Si VA A LA IZQUIERDA O SI VA A LA DERECHA
aux=aux.hijoIzquierdo;
              if(aux==null){//encontré el lugar para insertarlo; SUBARBOL IZQUIER-
DO
                 padre.hijoIzquierdo=nuevo;
                 return;
              }
           }else{
              aux=aux.hijoDerecho;
              if(aux==null){//encontré el lugar para insertarlo; SUBARBOL DERECHA
padre.hijoDerecho=nuevo;
                 return;
              }

           }
        }
     }
  }
//Método para cuando el árbol no tiene elementos estaVacio()
   public boolean estaVacio(){
     return raiz==null;
   }
```

2.3. Recorridos de Árboles (inorden, preorden, postorden)

Los recorridos inorden, preorden y postorden son técnicas fundamentales para visitar los nodos de un árbol en una secuencia específica. En este artículo, exploraremos en detalle cada uno de estos recorridos, cómo se implementan y sus aplicaciones en el contexto de las estructuras de datos de árboles.

2.3.1. Recorrido Inorden

El recorrido inorden, también conocido como recorrido simétrico, visita los nodos de un árbol de manera que primero visita el subárbol izquierdo, luego el nodo raíz y finalmente el subárbol derecho.

Este recorrido es útil para procesar los nodos en orden ascendente en un árbol de búsqueda binaria, ya que garantiza que los nodos se visiten en orden creciente.

Según Goodrich, Tamassia y Goldwasser (2011), "el recorrido inorden es un método de recorrido de árboles que visita primero el subárbol izquierdo, luego el nodo raíz y finalmente el subárbol derecho".

2.3.2. Recorrido Preorden

El recorrido preorden, también conocido como recorrido en profundidad primero (DFS), visita los nodos de un árbol de manera que primero visita el nodo raíz, luego el subárbol izquierdo y finalmente el subárbol derecho.

Este recorrido es útil para la creación de copias de árboles y para imprimir la expresión de un árbol de manera legible.

De acuerdo con Cormen, Leiserson, Rivest y Stein (2009), "el recorrido preorden es un método de recorrido de árboles que visita primero el nodo raíz, luego el subárbol izquierdo y finalmente el subárbol derecho".

2.3.3. Recorrido Postorden

El recorrido postorden, también conocido como recorrido en profundidad último (DFS), visita los nodos de un árbol de manera que primero visita los subárboles izquierdo y derecho y finalmente el nodo raíz.

Este recorrido es útil para liberar la memoria asignada a un árbol y para evaluar expresiones aritméticas representadas como árboles de expresión.

Según Horowitz, Sahni y Anderson-Freed (2013), "el recorrido postorden es un método de recorrido de árboles que visita primero los subárboles izquierdo y derecho, y finalmente el nodo raíz".

2.3.4. Aplicaciones

Los recorridos inorden, preorden y postorden tienen una variedad de aplicaciones en informática, como:

• Árboles de Búsqueda Binaria: Estos recorridos se utilizan para buscar, insertar o eliminar elementos en un árbol de búsqueda binaria de manera eficiente.

• Compiladores: Los recorridos preorden y postorden se utilizan para generar código intermedio o para optimizar expresiones en compiladores.

• Evaluación de Expresiones: Los recorridos preorden y postorden se utilizan para evaluar expresiones aritméticas representadas como árboles de expresión.

Aplicación 1

Se debe adherir el siguiente código al código presentado en la aplicación del árbol binario, en la clase "ArbolBinario"

Clase ArbolBinario

```
//Método recorrer si tenemos elementos IN-ORDEN
  //RECURSIVIDAD
  //Izquierda, Raíz, Derecha
  public void inOrden(NodoArbol r){
     if(r!=null){
        inOrden(r.hijoIzquierdo);
        System.out.println(r.dato);
        inOrden(r.hijoDerecho);
     }
  }
 //Método para recorrer el árbol binario PRE-ORDEN
  //raíz, izquierda, derecha
  public void preOrden(NodoArbol r){
     if (r!=null){
        System.out.println(r.dato);
        preOrden(r.hijoIzquierdo);
        preOrden(r.hijoDerecho);
```



```
        preOrden(r.hijoDerecho);
     }
  }

  //Método para recorrer el árbil binario POST-ORDEN
   //izquierda, derecha, raíz
  public void postOrden(NodoArbol r){
     if (r!=null){
        postOrden(r.hijoIzquierdo);
        postOrden(r.hijoDerecho);
        System.out.println(r.dato);
     }
  }

}
```

Clase TDA_Aroles_Binarios

```
package ed_tda13_arboles_binarios;

import javax.swing.JOptionPane;

/**
 *
 * @author Marcelo
 */
public class ED_TDA13_Arboles_Binarios {
/**
    * @param args the command line arguments
    */
   public static void main(String[] args) {
     // TODO code application logic here
      int opcion = 0, elemento;
     String nombre;
```

```
ArbolBinario arbol = new ArbolBinario();
do {
  try {
    opcion = Integer.parseInt(JOptionPane.showInputDialog(null,
        "1. Agregar un Nodo al Arbol Binario\n"
        + "2. Recorrer el Arbol In-Orden\n"
        + "3. Recorrer el Arbol Pre-Orden\n"
        + "4. Recorrer el Arbol Post-Orden\n"
        + "5. Salir\n"
        + "Elige una opción.....", "Menú de Arboles Binarios",
        JOptionPane.QUESTION_MESSAGE));
    switch (opcion) {
      case 1:
        elemento = Integer.parseInt(JOptionPane.showInputDialog(null,
            "Ingrese el Id del Nodo....", "Agregando el Nodo",
            JOptionPane.QUESTION_MESSAGE));
        /*nombre=JOptionPane.showInputDialog(null, "Ingresa el nombre del
Nodo....",
            "Agregando el Nodo",JOptionPane.QUESTION_MESSAGE);*/
        arbol.agregarNodo(elemento, "");
        break;
      case 2:
        if (!arbol.estaVacio()) {
          arbol.inOrden(arbol.raiz);
        } else {
          JOptionPane.showMessageDialog(null, "El Árbol está vacío...",
              "Revíselo por favor", JOptionPane.INFORMATION_MESSAGE);
        }
        break;
      case 3:
        if (!arbol.estaVacio()) {
          arbol.preOrden(arbol.raiz);
        } else {
```



```
          JOptionPane.showMessageDialog(null, "El Árbol está vacío...",
              "Revíselo por favor", JOptionPane.INFORMATION_MESSAGE);
        }
        break;
      case 4:
        if (!arbol.estaVacio()) {
          arbol.postOrden(arbol.raiz);
        } else {
          JOptionPane.showMessageDialog(null, "El Árbol está vacío...",
              "Revíselo por favor", JOptionPane.INFORMATION_MESSAGE);
        }
        break;
      case 5:
        JOptionPane.showMessageDialog(null, "Aplicación Finalizada",
            "FIN", JOptionPane.INFORMATION_MESSAGE);
        break;
      default:
        JOptionPane.showMessageDialog(null, "Opción Incorrecta",
            "Verifique por favor....", JOptionPane.INFORMATION_MESSAGE);
    }

  } catch (NumberFormatException n) {
    JOptionPane.showMessageDialog(null, "Error" + n.getMessage());
  }

} while (opcion != 5);
  }
}
```

CAPÍTULO III

ARBOLES BINARIOS DE BÚSQUEDA

Los árboles binarios de búsqueda son una estructura de datos fundamental en informática utilizada para organizar y almacenar datos de manera eficiente. En este artículo, exploraremos en profundidad qué son los árboles binarios de búsqueda, cómo funcionan, sus características principales y sus aplicaciones en diversos campos de la informática.

Un árbol binario de búsqueda (BST por sus siglas en inglés, Binary Search Tree) es una estructura de datos jerárquica en la que cada nodo tiene hasta dos hijos, denominados hijo izquierdo y hijo derecho. La propiedad clave de un BST es que para cada nodo, los valores en el subárbol izquierdo son menores que el valor del nodo, mientras que los valores en el subárbol derecho son mayores.

Según Sedgewick y Wayne (2011), "un árbol binario de búsqueda es una estructura de datos jerárquica en la que cada nodo tiene un valor asociado y dos hijos, uno a la izquierda y otro a la derecha. La propiedad de los BST es que, para cada nodo, todos los valores en el subárbol izquierdo son menores que el valor del nodo y todos los valores en el subárbol derecho son mayores".

Los árboles binarios de búsqueda admiten una variedad de operaciones fundamentales, incluyendo inserción, eliminación, búsqueda y recorrido de nodos.

- Inserción: Para insertar un nuevo nodo en un árbol binario de búsqueda, se compara el valor del nuevo

nodo con el valor del nodo actual. Dependiendo de la comparación, se decide si el nuevo nodo debe colocarse en el subárbol izquierdo o derecho del nodo actual. Este proceso continúa hasta que se encuentra un lugar vacío para insertar el nuevo nodo.

• Eliminación: La eliminación de un nodo en un BST puede ser más complicada que la inserción. Se deben considerar diferentes casos, como si el nodo a eliminar es una hoja, tiene un solo hijo o tiene dos hijos. Después de eliminar el nodo, el árbol debe reorganizarse para mantener su propiedad de árbol binario de búsqueda.

• Búsqueda: La búsqueda en un árbol binario de búsqueda sigue un proceso similar a la inserción. Se compara el valor buscado con el valor de los nodos actuales y se decide en qué dirección continuar la búsqueda, ya sea hacia el subárbol izquierdo o derecho, hasta que se encuentre el nodo deseado o se llegue a un nodo vacío.

• Recorrido: Los árboles binarios de búsqueda admiten varios métodos de recorrido para visitar todos los nodos del árbol en un orden específico, como el recorrido en orden, el recorrido en preorden y el recorrido en postorden.

Los árboles binarios de búsqueda tienen una amplia variedad de aplicaciones en informática, incluyendo:

• Búsqueda Eficiente: Debido a la estructura ordenada

de un BST, la búsqueda de elementos se puede realizar de manera eficiente utilizando la propiedad de búsqueda binaria.

• Almacenamiento y Organización de Datos: Los BST se utilizan en bases de datos y sistemas de archivos para almacenar y organizar datos de manera eficiente, especialmente cuando se requiere acceso rápido a los datos ordenados.

• Implementación de Conjuntos y Diccionarios: Los BST se pueden utilizar para implementar estructuras de conjuntos y diccionarios, donde la inserción, eliminación y búsqueda de elementos se pueden realizar de manera eficiente.

Según Goodrich, Tamassia y Goldwasser (2011), "los árboles binarios de búsqueda son ampliamente utilizados en informática debido a su eficiencia en operaciones de búsqueda y su capacidad para almacenar y organizar datos de manera ordenada".

3.1. Conceptos Básicos

Los árboles binarios de búsqueda son una estructura de datos fundamental en ciencias de la computación. Según Cormen, Leiserson, Rivest y Stein (2009), un árbol binario de búsqueda es un tipo de árbol en el cual cada nodo tiene a lo sumo dos hijos, conocidos como nodo izquierdo y nodo derecho, y en el cual se cumple la siguiente condición: para cualquier nodo, todos los nodos del subárbol izquierdo tienen valores inferiores al

valor del nodo, y todos los nodos del subárbol derecho tienen valores mayores que el valor del nodo.

La característica principal de un árbol binario de búsqueda es su capacidad para realizar búsquedas eficientes. Según Goodrich y Tamassia (2013), la estructura jerárquica de un árbol binario de búsqueda permite realizar búsquedas en tiempo logarítmico en el peor de los casos, lo que lo convierte en una estructura de datos eficiente para la búsqueda de elementos en una colección ordenada.

Además de la eficiencia en las búsquedas, los árboles binarios de búsqueda también permiten la inserción y eliminación de elementos de manera eficiente. De acuerdo con Horowitz, Sahni y Mehta (2011), la inserción y eliminación en un árbol binario de búsqueda se pueden realizar en tiempo logarítmico, siempre y cuando el árbol se encuentre balanceado.

Uno de los conceptos clave en los árboles binarios de búsqueda es el balanceamiento. Como mencionan Cormen et al. (2009), un árbol binario de búsqueda está balanceado cuando la altura de sus subárboles izquierdo y derecho difiere en, a lo sumo, una unidad. Mantener el balance de un árbol binario de búsqueda es crucial para garantizar su eficiencia en operaciones de inserción, eliminación y búsqueda.

Además de las operaciones básicas como búsqueda, inserción y eliminación, los árboles binarios de búsqueda

también permiten realizar recorridos en profundidad, como el preorden, inorden y postorden. Según Goodrich y Tamassia (2013), los recorridos en profundidad son útiles para procesar todos los nodos de un árbol binario de búsqueda de manera sistemática.

3.2. Operaciones (inserción, búsqueda, eliminación)

Los árboles binarios de búsqueda (BST) son una estructura de datos fundamental en informática que permite organizar y almacenar datos de manera eficiente. Las operaciones principales en un BST incluyen la inserción de nuevos nodos, la búsqueda de nodos existentes y la eliminación de nodos. En este artículo, exploraremos en detalle estas operaciones y su importancia en la manipulación de árboles binarios de búsqueda.

3.2.1. Inserción

La inserción es una operación fundamental en los árboles binarios de búsqueda que consiste en agregar un nuevo nodo al árbol de manera que se mantenga la propiedad de orden del BST. Esta propiedad establece que para cada nodo, todos los valores en el subárbol izquierdo son menores que el valor del nodo, y todos los valores en el subárbol derecho son mayores.

Para insertar un nuevo nodo en un BST, se sigue un proceso recursivo o iterativo. En cada paso, se compara

el valor del nuevo nodo con el valor del nodo actual. Si el valor es menor, se desciende hacia el subárbol izquierdo; si es mayor, se desciende hacia el subárbol derecho. Este proceso continúa hasta llegar a un nodo vacío, donde se inserta el nuevo nodo.

Según Cormen, Leiserson, Rivest y Stein (2009), "la inserción en un árbol binario de búsqueda se realiza comparando el valor del nuevo nodo con el valor del nodo actual y descendiéndose hacia el subárbol izquierdo o derecho según sea necesario hasta encontrar un lugar vacío para insertar el nuevo nodo".

3.2.2. Búsqueda

La búsqueda es otra operación importante en los árboles binarios de búsqueda que consiste en encontrar un nodo con un valor específico en el árbol. La búsqueda se realiza comparando el valor buscado con el valor del nodo actual y descendiendo hacia el subárbol izquierdo o derecho según corresponda. Este proceso se repite recursivamente hasta encontrar el nodo deseado o hasta llegar a un nodo vacío, lo que indica que el valor no está presente en el árbol.

La búsqueda en un BST es eficiente debido a la propiedad de orden del árbol, que permite descartar rápidamente subárboles enteros basándose en la comparación del valor buscado con el valor del nodo actual.

Según Sedgewick y Wayne (2011), "la búsqueda en un

árbol binario de búsqueda se realiza comparando el valor buscado con el valor del nodo actual y descendiendo hacia el subárbol izquierdo o derecho según corresponda hasta encontrar el nodo deseado o llegar a un nodo vacío".

3.2.3. Eliminación

La eliminación es la operación que permite eliminar un nodo específico de un BST manteniendo la propiedad de orden del árbol. La eliminación en un BST puede ser más compleja que la inserción y la búsqueda, ya que se deben considerar diferentes casos dependiendo de la estructura del árbol y la ubicación del nodo a eliminar.

Los casos principales a considerar durante la eliminación son:

- El nodo a eliminar es una hoja (no tiene hijos).
- El nodo a eliminar tiene un solo hijo.
- El nodo a eliminar tiene dos hijos.

En cada caso, se deben realizar diferentes acciones para mantener la propiedad de orden del árbol después de la eliminación, como la reorganización de los nodos y la redistribución de los hijos.

Según Goodrich, Tamassia y Goldwasser (2011), "la eliminación en un árbol binario de búsqueda implica considerar diferentes casos, como si el nodo a eliminar es una hoja, tiene un solo hijo o tiene dos hijos, y realizar

acciones apropiadas para mantener la propiedad de orden del árbol".

3.3. Balanceo de árboles

En informática, el balanceo de árboles binarios de búsqueda (BST) es un proceso crucial para mantener la eficiencia de las operaciones de inserción, búsqueda y eliminación. Un árbol binario de búsqueda está balanceado cuando la altura de su subárbol izquierdo y derecho no difiere en más de una unidad. En este artículo, exploraremos en detalle la importancia del balanceo de los árboles binarios de búsqueda, así como diferentes técnicas para lograr un balanceo óptimo.

3.3.1. Importancia del Balanceo

Un árbol binario de búsqueda desbalanceado puede llevar a un peor rendimiento en las operaciones de búsqueda, inserción y eliminación. Cuando un árbol binario de búsqueda está desbalanceado, las operaciones pueden requerir un tiempo lineal en lugar de logarítmico, lo que reduce la eficiencia del algoritmo y aumenta los tiempos de ejecución. Por lo tanto, mantener un árbol binario de búsqueda balanceado es crucial para garantizar un rendimiento óptimo en aplicaciones que dependen de estas estructuras de datos.

Según Sedgewick y Wayne (2011), "el balanceo de árboles binarios de búsqueda es esencial para mantener el rendimiento de las operaciones de búsqueda, inserción

y eliminación en tiempo logarítmico. Un árbol binario de búsqueda desbalanceado puede llevar a tiempos de ejecución lineales en lugar de logarítmicos, lo que afecta negativamente el rendimiento de los algoritmos que dependen de estas estructuras de datos".

3.3.2. Técnicas de Balanceo

Existen varias técnicas para balancear árboles binarios de búsqueda, cada una con sus propias ventajas y desventajas. Algunas de las técnicas más comunes incluyen:

Rotaciones simples y dobles: Las rotaciones son operaciones que reorganizan los nodos de un árbol para mejorar su balance. Las rotaciones simples y dobles se utilizan para corregir diferentes tipos de desequilibrios, como el desequilibrio izquierdo-izquierdo, derecho-derecho, izquierdo-derecho y derecho-izquierdo.

Árboles AVL: Los árboles AVL son una variante de los árboles binarios de búsqueda que mantienen un equilibrio óptimo automáticamente. Se aseguran de que la diferencia de altura entre los subárboles izquierdo y derecho de cada nodo sea como máximo uno. Esto se logra mediante la realización de rotaciones durante la inserción y eliminación para mantener el balance.

Árboles Rojo-Negro: Los árboles rojo-negro son otra variante de los árboles binarios de búsqueda que garantizan un balance óptimo. Cada nodo en un árbol

rojo-negro está marcado con un color (rojo o negro), y se aplican reglas específicas durante la inserción y eliminación para mantener el balance y garantizar que ningún camino desde la raíz hasta una hoja tenga más de dos veces la longitud de cualquier otro camino.

Árboles Splay: Los árboles splay son una estructura de datos dinámica que reorganiza los nodos en respuesta a operaciones de búsqueda. Cuando un nodo es accedido, se mueve hacia la raíz del árbol mediante una serie de rotaciones y reestructuraciones, lo que puede mejorar el balance del árbol con el tiempo.

3.3.3. Aplicaciones

El balanceo de árboles binarios de búsqueda se aplica en una amplia variedad de aplicaciones informáticas donde se requiere un acceso eficiente a los datos. Algunas de estas aplicaciones incluyen:

- Bases de datos: Para indexar y buscar registros de manera eficiente.
- Compiladores: Para almacenar y recuperar información sobre símbolos y variables.
- Sistemas operativos: Para gestionar procesos y recursos de manera eficiente.
- Redes: Para enrutar paquetes de datos de manera eficiente a través de la red.

El balanceo de árboles binarios de búsqueda es esencial

en estas aplicaciones para garantizar un rendimiento óptimo y tiempos de respuesta rápidos.

Aplicación 1

Clase NodoArbol

```
/**
 *
 * @author Marcelo
 */

public class NodoArbol {
   int valor;
   NodoArbol izquierdo;
   NodoArbol derecho;

   public NodoArbol(int valor) {
      this.valor = valor;
      izquierdo = null;
      derecho = null;
   }
}
```

Clase ArbolBinarioBusqueda

```
/**
 *
 * @author Marcelo
 */
public class ArbolBinarioBusqueda {
   NodoArbol raiz;

   public ArbolBinarioBusqueda() {
      raiz = null;
```

```
    }

    public void insertar(int valor) {
        raiz = insertarRecursivo(raiz, valor);
    }

    private NodoArbol insertarRecursivo(NodoArbol nodo, int valor) {
        if (nodo == null) {
            return new NodoArbol(valor);
        }

        if (valor < nodo.valor) {
            nodo.izquierdo = insertarRecursivo(nodo.izquierdo, valor);
        } else if (valor > nodo.valor) {
            nodo.derecho = insertarRecursivo(nodo.derecho, valor);
        }

        return nodo;
    }

    // Aquí puedes agregar métodos de balanceo como rotaciones, Árboles AVL,
Árboles Rojo-Negro, etc.
}
```

Clase TDA_Arbol_Busqueda

```
/**
 *
 * @author Marcelo
 */
public class Main {
    public static void main(String[] args) {
        ArbolBinarioBusqueda arbol = new ArbolBinarioBusqueda();
```



```
        // Insertar nodos de ejemplo
        arbol.insertar(50);
        arbol.insertar(30);
        arbol.insertar(70);
        arbol.insertar(20);
        arbol.insertar(40);
        arbol.insertar(60);
        arbol.insertar(80);

        // Aquí puedes llamar a los métodos de balanceo o realizar otras operaciones
en el árbol
    }
}
```

CAPÍTULO IV

GRAFOS

Los grafos son estructuras de datos fundamentales en ciencias de la computación que permiten modelar y representar relaciones entre objetos. En este artículo, exploraremos en detalle qué son los grafos, cómo se representan y algunas de sus aplicaciones en programación.

4.1. Definición y Componentes

Un grafo es una colección de nodos (vértices) conectados entre sí mediante arcos (aristas). Estos nodos pueden representar entidades como personas, ciudades o elementos, mientras que los arcos representan relaciones entre estas entidades. Los grafos se utilizan para modelar una variedad de problemas del mundo real, desde redes sociales hasta rutas de vuelo.

Según Cormen, Leiserson, Rivest y Stein (2009), "un grafo consiste en un conjunto de vértices y un conjunto de aristas que conectan estos vértices. Los grafos son una de las estructuras de datos fundamentales en ciencias de la computación y se utilizan para modelar relaciones entre objetos".

4.1.1. Componentes de Grafos

Los grafos constan de varios componentes fundamentales que describen su estructura y propiedades. Algunos de los componentes más importantes son:

• Vértices (Nodos): Son los elementos individuales del grafo que representan entidades. Cada vértice puede

tener un nombre o una etiqueta única que lo identifica.

• Aristas (Arcos): Son las conexiones entre los nodos del grafo que representan relaciones entre las entidades. Las aristas pueden ser dirigidas o no dirigidas, dependiendo de si la relación es unidireccional o bidireccional, respectivamente.

• Peso: En algunos grafos, las aristas pueden tener un peso o costo asociado que indica la distancia, el tiempo o cualquier otra medida entre los nodos que conectan. Los pesos se utilizan en problemas como el cálculo de la ruta más corta.

• Grafo Dirigido: Es un grafo en el que las aristas tienen una dirección asociada. Esto significa que la relación entre dos nodos es unidireccional, y se representa mediante una flecha que va desde el nodo de origen al nodo de destino.

• Grafo No Dirigido: Es un grafo en el que las aristas no tienen una dirección asociada. Esto significa que la relación entre dos nodos es bidireccional, y se representa mediante una línea que conecta los nodos sin una dirección específica.

4.2. Grafos Dirigidos vs. No Dirigidos

Los grafos son estructuras de datos fundamentales en ciencias de la computación que representan relaciones entre objetos. Una de las distinciones principales en los grafos es entre los grafos dirigidos y los grafos no dirigidos.

En este artículo, exploraremos en detalle las diferencias entre estos dos tipos de grafos, sus características, y sus aplicaciones en programación.

4.2.1. Definición y Características

Grafos Dirigidos: En un grafo dirigido, también conocido como digrafo, cada arista tiene una dirección asociada. Esto significa que la relación entre dos nodos tiene una dirección específica, indicando que un nodo es el origen y otro es el destino. Por ejemplo, en una red de transporte, las carreteras de una ciudad a otra pueden ser modeladas como un grafo dirigido, donde una carretera solo se puede recorrer en una dirección.

Grafos No Dirigidos: En un grafo no dirigido, las aristas no tienen una dirección asociada. Esto significa que la relación entre dos nodos es simétrica, lo que implica que la conexión es bidireccional. Por ejemplo, en una red social, las amistades entre usuarios pueden modelarse como un grafo no dirigido, donde la conexión entre dos usuarios es recíproca.

4.2.2. Representación

Grafos Dirigidos: En la representación de grafos dirigidos, cada arista se representa como un par ordenado de nodos, donde el primer nodo es el origen y el segundo nodo es el destino. Por ejemplo, si hay una arista que va desde el nodo A al nodo B en un grafo dirigido, se representaría como (A, B).

Grafos No Dirigidos: En la representación de grafos no dirigidos, cada arista se representa como un conjunto no ordenado de nodos, lo que implica que la conexión entre los nodos es bidireccional. Por ejemplo, si hay una arista que conecta los nodos A y B en un grafo no dirigido, se representaría como {A, B} o {B, A}, ya que la conexión es simétrica.

4.2.3. Aplicaciones

Grafos Dirigidos: Los grafos dirigidos se utilizan en una variedad de aplicaciones donde la dirección de la relación entre objetos es importante. Algunas de estas aplicaciones incluyen redes de transporte, redes de flujo, representación de dependencias, y modelado de sistemas de información.

Grafos No Dirigidos: Los grafos no dirigidos se utilizan en aplicaciones donde la simetría de la relación entre objetos es fundamental. Algunas de estas aplicaciones incluyen redes sociales, modelado de relaciones entre elementos, y problemas de conectividad en grafos.

4.2.4. Algoritmos y Operaciones

Grafos Dirigidos: Para grafos dirigidos, existen algoritmos específicos que tienen en cuenta la dirección de las aristas. Algunos de estos algoritmos incluyen la búsqueda en profundidad (DFS) y la búsqueda en amplitud (BFS), así como algoritmos para encontrar caminos más cortos como el algoritmo de Dijkstra.

Grafos No Dirigidos: Para grafos no dirigidos, los algoritmos y operaciones suelen ser más simples, ya que no se tiene en cuenta la dirección de las aristas. Sin embargo, todavía se pueden aplicar algoritmos como DFS y BFS para recorrer el grafo y encontrar componentes conectados.

4.3. Representación de Grafos (listas de adyacencia, matrices de adyacencia)

Existen varias formas de representar grafos en estructuras de programación, cada una con sus propias ventajas y desventajas:

Matriz de Adyacencia: En esta representación, se utiliza una matriz bidimensional para almacenar información sobre las conexiones entre los nodos. La posición (i, j) de la matriz indica si existe una arista entre los nodos i y j.

Lista de Adyacencia: En esta representación, se utiliza una lista de listas para almacenar los nodos vecinos de cada nodo. Cada elemento de la lista principal corresponde a un nodo, y contiene una lista de los nodos vecinos de ese nodo.

Lista de Aristas: En esta representación, se utiliza una lista para almacenar todas las aristas del grafo. Cada elemento de la lista representa una arista y contiene información sobre los nodos que conecta.

4.3.1. Aplicaciones en Programación

Los grafos tienen una amplia variedad de aplicaciones

en programación, algunas de las cuales incluyen:

• Redes Sociales: Modelado de amistades y conexiones entre usuarios en redes sociales como Facebook.

• Algoritmos de Rutas: Determinación de la ruta más corta entre dos puntos en un mapa utilizando algoritmos como Dijkstra o Bellman-Ford.

• Compiladores: Representación de estructuras de datos como árboles de análisis sintáctico y árboles de dependencia de expresiones.

• Planificación de Proyectos: Modelado de dependencias entre tareas en un proyecto y determinación del orden óptimo de ejecución.

Aplicación 1

A continuación se presenta la aplicación para hallar los caminos mínimos entre los componentes de un grafo no dirigido en el lenguaje de programación Java

Clase ejecutable en Java

Clase CaminoMinimo

```
package ed_tda14_grafos;


/**
 *
 * @author Marcelo
 */
public class CaminoMinimo {
```



```
  public String algoritmo1(long[][] mAdya) {
     int vertices = mAdya.length;
     long matrizAdyacencia[][] = mAdya;
     String caminos[][] = new String[vertices][vertices];
     String caminosAuxiliares[][] = new String[vertices][vertices];
     String caminoRecorrido = "", cadena = "", caminos2 = "";
     int i, j, k;
     float temp1, temp2, temp3, temp4, minimo;

     for (i = 0; i < vertices; i++) {
        for (j = 0; j < vertices; j++) {
           caminos[i][j] = "";
           caminosAuxiliares[i][j] = "";
        }
     }

     for (k = 0; k < vertices; k++) {
        for (i = 0; i < vertices; i++) {
           //System.out.print("["+matrizAdyacencia[k][i]+"]");
           for (j = 0; j < vertices; j++) {
              temp1=matrizAdyacencia[i][j];
              temp2=matrizAdyacencia[i][k];
              temp3=matrizAdyacencia[k][j];

              temp4=temp2+temp3;
              minimo=Math.min(temp1, temp4);
              if(temp1!=temp4){
                 if(minimo == temp4){
                    caminoRecorrido="";
                    caminosAuxiliares[i][j]=k+"";
                    caminos[i][j]=caminosR(i,k, caminosAuxiliares, caminoRecorri-
do)+(k+1);
                 }
              }
```

```
                matrizAdyacencia[i][j]=(long)minimo;
            }
        }
        // System.out.print("\n");
    }

    for(i=0; i<vertices; i++){
      for(j=0; j<vertices; j++){
          cadena=cadena+ "[" + matrizAdyacencia[i][j] + "]";
      }
      cadena=cadena+"\n";
    }

    for (i=0; i<vertices; i++){
       for(j=0; j<vertices; j++){
          if(matrizAdyacencia[i][j] != 1000000000){
             if(i!=j){
                if(caminos[i][j].equals("")){
                   caminos2+= "De ("+ (i+1) + "--->" + (j+1) + ") Irse por .... (" + (i+1) + ","
+ (j+1) + ")\n";
                 }else{
                   caminos2+= "De ("+ (i+1) + "--->" + (j+1) + ") Irse por .... (" + (i+1) + ","
+ caminos[i][j] + "," + (j+1) + ")\n";
                }
             }
          }
       }
    }
    return "La matriz de camino Mas Corto entre los diferentes vértices es: \n"+ca-
dena
        + "\n Los diferentes caminos más cortos entre vértices son: \n"+caminos2;

  }
```



```
    public String caminosR(int i, int k, String[][] caminosAux, String caminoRecorrido)
{
       if (caminosAux[i][k].equals("")){
          return "";
       }else{
          caminoRecorrido+=caminosR(i, Integer.parseInt(caminosAux[i][k].toString()),
caminosAux, caminoRecorrido)+(Integer.parseInt(caminosAux[i][k].toString())+1)+",";
          return caminoRecorrido;
       }
    }

}
```

Clase ejecutable en Java

Clase TDA_Grafos

```
package ed_tda14_grafos;

/**
 *
 * @author Marcelo
 */
public class ED_TDA14_Grafos {

   /**
    * @param args the command line arguments
    */
   public static void main(String[] args) {
      // TODO code application logic here
      long
matrizA[][]={{0,3,4,999999999,8,999999999},{999999999,0,999999999,999999999,5,9
99999999},{999999999,999999999,0,999999999,3,999999999},{9999 99999,999999999
,999999999,0,999999999,999999999},{999999999,999999999, 999999999,7,0,3},{9999
99999,999999999,999999999,2,999999999,0}};
```

```
        CaminoMinimo ruta=new CaminoMinimo();
        System.out.println(ruta.algoritmo1(matrizA));
    }

}
```

Aplicación 2

Clase Nodo

```
/**
 *
 * @author Marcelo
 */
public class Nodo {
    private int id;

    public Nodo(int id) {
        this.id = id;
    }

    public int getId() {
        return id;
    }

    @Override
    public String toString() {
        return "Nodo{" +
                "id=" + id +
                '}';
    }
}
```


Clase Arista

```
/**
 *
 * @author Marcelo
 */
public class Arista {
    private Nodo origen;
    private Nodo destino;
    private int peso;

    public Arista(Nodo origen, Nodo destino, int peso) {
        this.origen = origen;
        this.destino = destino;
        this.peso = peso;
    }

    public Nodo getOrigen() {
        return origen;
    }

    public Nodo getDestino() {
        return destino;
    }

    public int getPeso() {
        return peso;
    }

   @Override
    public String toString() {
      return "Arista{" +
              "origen=" + origen +
             ", destino=" + destino +
```

```
            ", peso=" + peso +
            '}';
    }
}
```

Clase GrafoDirigido

```
/**
 *
 * @author Marcelo
 */
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class GrafoDirigido {
    private Map<Nodo, List<Arista>> mapaAdyacencia;

    public GrafoDirigido() {
        this.mapaAdyacencia = new HashMap<>();
    }

    public void agregarNodo(Nodo nodo) {
        mapaAdyacencia.putIfAbsent(nodo, new ArrayList<>());
    }

    public void agregarArista(Nodo origen, Nodo destino, int peso) {
        if (!mapaAdyacencia.containsKey(origen)) {
            agregarNodo(origen);
        }
        if (!mapaAdyacencia.containsKey(destino)) {
            agregarNodo(destino);
        }
```



```
        mapaAdyacencia.get(origen).add(new Arista(origen, destino, peso));
    }

    public List<Arista> obtenerAristasSalientes(Nodo nodo) {
        return mapaAdyacencia.getOrDefault(nodo, new ArrayList<>());
    }

    public List<Nodo> obtenerNodos() {
        return new ArrayList<>(mapaAdyacencia.keySet());
    }

    @Override
    public String toString() {
        return "GrafoDirigido{" +
                "mapaAdyacencia=" + mapaAdyacencia +
                '}';
    }
}
```

CAPÍTULO V

ALGORITMOS VORACES

5.1. Introducción

Los algoritmos voraces, también conocidos como algoritmos ávidos o greedy, son un enfoque de resolución de problemas en informática que consiste en tomar decisiones locales óptimas en cada paso con la esperanza de encontrar una solución global óptima. Este enfoque es simple y eficiente en muchos casos, lo que lo convierte en una herramienta poderosa para una amplia gama de problemas. En este artículo, exploraremos los fundamentos de los algoritmos voraces, sus características principales y algunas aplicaciones comunes.

5.2. Definición y Características

Los algoritmos voraces son algoritmos que toman decisiones en cada paso para avanzar hacia una solución óptima, sin revisar decisiones previas. A diferencia de otros enfoques más complejos, como la programación dinámica, los algoritmos voraces no revisan las decisiones tomadas anteriormente y no se preocupan por las consecuencias a largo plazo. En cambio, se centran en elegir la mejor opción en cada momento, esperando que estas decisiones locales conduzcan a una solución global óptima.

Según Cormen, Leiserson, Rivest y Stein (2009), "los algoritmos voraces son un enfoque de resolución de problemas que implica tomar la mejor decisión local en cada paso, con la esperanza de encontrar una solución

global óptima. Estos algoritmos son simples, eficientes y se aplican en una variedad de problemas".

5.3. Principios Básicos

Los algoritmos voraces siguen varios principios básicos que guían su funcionamiento:

• Elección Greedy: En cada paso, el algoritmo selecciona la opción óptima localmente, sin considerar las consecuencias futuras. Esta elección se basa en algún criterio de selección, como la máxima ganancia, el menor costo o el mínimo tiempo.

• Solución Factible: En cada paso, el algoritmo verifica si la opción seleccionada satisface los requisitos del problema y no viola ninguna restricción.

• Subestructuras Óptimas: Los algoritmos voraces a menudo tienen una estructura recursiva o iterativa que permite dividir el problema en subproblemas más pequeños, cada uno de los cuales se resuelve de manera óptima.

5.4. Aplicaciones de Algoritmos Voraces

Los algoritmos voraces se aplican en una variedad de problemas en informática y ciencias de la computación. Algunas de las aplicaciones más comunes incluyen:

• Algoritmo de Kruskal: Utilizado para encontrar el árbol de expansión mínimo de un grafo ponderado no dirigido.

• Algoritmo de Prim: Similar al algoritmo de Kruskal, se utiliza para encontrar el árbol de expansión mínimo de un grafo ponderado no dirigido.

• Algoritmo de Selección de Actividades: Utilizado para programar un conjunto de actividades en un intervalo de tiempo limitado, maximizando la cantidad de actividades que se pueden realizar sin solapamiento

Aplicación 1

Clase AlgoritmosVoraces

```
package ed_tda15_algoritmosvoraces;

import java.util.Scanner;

/**
 *
 * @author Marcelo
*/
public class ED_TDA15_AlgoritmosVoraces {

   /**
    * @param args the command line arguments
    */
   public static void main(String[] args) {
       // TODO code application logic here
       int[] billete={100,50,20,10,5,1};
      int saldo_total;
       int vuelto;
      int precio;
       Scanner leer=new Scanner(System.in);
      System.out.println("Ingrese el valor entregado por el cliente: ");
```

```
        saldo_total=leer.nextInt();
        leer.nextLine();
        System.out.println("Cuál es el costo del artículo: ");
        precio=leer.nextInt();
        leer.nextLine();
        vuelto=saldo_total-precio;
        int[] cambio=calcular(vuelto, billete);
        System.out.println("El número de cada uno de los billetes a entregar como vuelto
son: " +vuelto);
        for (int i=0; i<cambio.length; i++){
            System.out.println("$ "+billete[i] + " = "+ cambio[i]+ " unidades");
        }
    }
    public static int[] calcular(int vuelto, int[]valor){
        int[] billete=new int[valor.length];
        for (int i=0; i<valor.length; i++){
            while (valor[i] <= vuelto){
                billete[i]++;
                vuelto=vuelto-valor[i];
            }
        }
        return  billete;
    }

}
```

Clase Arista

```
/**
 *
 * @author Marcelo
*/
```



```
public class Arista implements Comparable<Arista> {
    private final int nodoInicio;
    private final int nodoFin;
    private final int peso;

    public Arista(int nodoInicio, int nodoFin, int peso) {
        this.nodoInicio = nodoInicio;
        this.nodoFin = nodoFin;
        this.peso = peso;
    }

    public int getNodoInicio() {
        return nodoInicio;
    }

    public int getNodoFin() {
        return nodoFin;
    }

    public int getPeso() {
        return peso;
    }

    @Override
    public int compareTo(Arista otraArista) {
        return Integer.compare(this.peso, otraArista.peso);
    }

    @Override
    public String toString() {
        return "(" + nodoInicio + ", " + nodoFin + ") peso: " + peso;
    }
}
```

Aplicación 2

Clase ConjuntoDisjunto

```
/**
 *
 * @author Marcelo
 */
public class ConjuntoDisjunto {
    private int[] padre;
    private int[] rango;
    private int n;

    public ConjuntoDisjunto(int n) {
        this.n = n;
        padre = new int[n];
        rango = new int[n];
        for (int i = 0; i < n; i++) {
            padre[i] = i;
            rango[i] = 0;
        }
    }

    public int encontrar(int x) {
        if (padre[x] != x) {
            padre[x] = encontrar(padre[x]);
        }
        return padre[x];
    }

    public void unir(int x, int y) {
        int raizX = encontrar(x);
        int raizY = encontrar(y);
        if (raizX != raizY) {
            if (rango[raizX] < rango[raizY]) {
                padre[raizX] = raizY;
```



```
            } else {
                padre[raizY] = raizX;
                if (rango[raizX] == rango[raizY]) {
                    rango[raizX]++;
                }
            }
        }
    }
}
```

Clase GrafoNoDirigido

```
/**
 *
 * @author Marcelo
 */
import java.util.ArrayList;
import java.util.List;

public class GrafoNoDirigido {
    private final int V;
    private final List<Arista> aristas;

    public GrafoNoDirigido(int V) {
        this.V = V;
        this.aristas = new ArrayList<>();
    }

    public void agregarArista(int nodoInicio, int nodoFin, int peso) {
        Arista arista = new Arista(nodoInicio, nodoFin, peso);
        aristas.add(arista);
    }
```

```
    public List<Arista> getAristas() {
        return aristas;
    }

    public int getV() {
        return V;
    }
}
```

Clase Kruskal

```
/**
 *
 * @author Marcelo
 */
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class Kruskal {
    public static List<Arista> algoritmoKruskal(GrafoNoDirigido grafo) {
        List<Arista> arbolExpansionMinima = new ArrayList<>();
        List<Arista> aristas = grafo.getAristas();
        Collections.sort(aristas);
       ConjuntoDisjunto conjuntoDisjunto = new ConjuntoDisjunto(grafo.getV());

        for (Arista arista : aristas) {
            int nodoInicio = arista.getNodoInicio();
           int nodoFin = arista.getNodoFin();

            int raizInicio = conjuntoDisjunto.encontrar(nodoInicio);
           int raizFin = conjuntoDisjunto.encontrar(nodoFin);
```



```
mport java.util.ArrayList;
import java.util.Collections;
import java.util.List;
            if (raizInicio != raizFin) {
                arbolExpansionMinima.add(arista);
                conjuntoDisjunto.unir(raizInicio, raizFin);
            }
        }

        return arbolExpansionMinima;
    }
}
```

Aplicación 3

Clase Arista

```
/**
 *
 * @author Marcelo
 */
public class Arista implements Comparable<Arista> {
    private final int nodoInicio;
    private final int nodoFin;
    private final int peso;

    public Arista(int nodoInicio, int nodoFin, int peso) {
        this.nodoInicio = nodoInicio;
        this.nodoFin = nodoFin;
        this.peso = peso;
    }

    public int getNodoInicio() {
        return nodoInicio;
```

```
    }

    public int getNodoFin() {
        return nodoFin;
    }

    public int getPeso() {
        return peso;
    }

    @Override
    public int compareTo(Arista otraArista) {
        return Integer.compare(this.peso, otraArista.peso);
    }

    @Override
    public String toString() {
        return "(" + nodoInicio + ", " + nodoFin + ") peso: " + peso;
    }
}
```

Clase GrafoNoDirigido

```
/**
 *
 * @author Marcelo
 */
import java.util.ArrayList;
import java.util.List;

public class GrafoNoDirigido {
    private final int V;
```



```
    private final List<List<Arista>> listaAdyacencia;

    public GrafoNoDirigido(int V) {
        this.V = V;
        this.listaAdyacencia = new ArrayList<>(V);
        for (int i = 0; i < V; i++) {
            listaAdyacencia.add(new ArrayList<>());
        }
    }

    public void agregarArista(int nodoInicio, int nodoFin, int peso) {
        listaAdyacencia.get(nodoInicio).add(new Arista(nodoInicio, nodoFin, peso));
        listaAdyacencia.get(nodoFin).add(new Arista(nodoFin, nodoInicio, peso));
    }

    public List<Arista> obtenerAristas(int nodo) {
        return listaAdyacencia.get(nodo);
    }

    public int getV() {
        return V;
    }
}
```

Clase Prim

```
/** *
 * @author Marcelo
 */
import java.util.ArrayList;
import java.util.List;
import java.util.PriorityQueue;
```

```
public class Prim {
    public static List<Arista> algoritmoPrim(GrafoNoDirigido grafo) {
        List<Arista> arbolExpansionMinima = new ArrayList<>();
        boolean[] visitado = new boolean[grafo.getV()];

        PriorityQueue<Arista> colaPrioridad = new PriorityQueue<>();
        visitado[0] = true;
        colaPrioridad.addAll(grafo.obtenerAristas(0));

        while (!colaPrioridad.isEmpty()) {
            Arista arista = colaPrioridad.poll();
            int nodoInicio = arista.getNodoInicio();
            int nodoFin = arista.getNodoFin();

            if (visitado[nodoInicio] && visitado[nodoFin]) {
                continue;
            }

            arbolExpansionMinima.add(arista);
            int nuevoNodo = visitado[nodoInicio] ? nodoFin : nodoInicio;
            visitado[nuevoNodo] = true;

            for (Arista vecino : grafo.obtenerAristas(nuevoNodo)) {
                if (!visitado[vecino.getNodoFin()]) {
                    colaPrioridad.add(vecino);
                }
            }
        }

        return arbolExpansionMinima;
    }
}
```


Aplicación 4

A continuación la implementación en el lenguaje Java del algoritmo voraz “Selección de Actividades”

Clase Actividad

```
/**
 *
 * @author Marcelo
 */
public class Actividad implements Comparable<Actividad> {
    private int inicio;
    private int fin;

    public Actividad(int inicio, int fin) {
        this.inicio = inicio;
        this.fin = fin;
    }

    public int getInicio() {
        return inicio;
    }

    public int getFin() {
        return fin;
    }

    @Override
    public int compareTo(Actividad otraActividad) {
        return Integer.compare(this.fin, otraActividad.fin);
    }

    @Override
```

```
    public String toString() {
        return "Actividad{" +
                "inicio=" + inicio +
                ", fin=" + fin +
                '}';
    }
}
```

Clase SeleccionActividades

```
/**
 *
 * @author Marcelo
 */
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class AlgoritmoSeleccionActividades {
    public static List<Actividad> seleccionActividades(List<Actividad> actividades) {
        List<Actividad> seleccionadas = new ArrayList<>();
        if (actividades.isEmpty()) {
            return seleccionadas;
        }

        // Ordenar actividades por tiempo de finalización
        Collections.sort(actividades);

        // La primera actividad siempre se selecciona
        seleccionadas.add(actividades.get(0));

        // Seleccionar actividades compatibles con la última actividad seleccionada
        int n = actividades.size();
```



```
        int n = actividades.size();
        Actividad ultimaSeleccionada = actividades.get(0);
        for (int i = 1; i < n; i++) {
            Actividad actividadActual = actividades.get(i);
            // Si la actividad actual comienza después de la finalización de la última
actividad seleccionada, se selecciona
            if (actividadActual.getInicio() >= ultimaSeleccionada.getFin()) {
                seleccionadas.add(actividadActual);
                ultimaSeleccionada = actividadActual;
            }
        }

        return seleccionadas;
    }
}
```

Clase EjecutableSeleccionActividades

```
/**
 *
 * @author Marcelo
 */
import java.util.ArrayList;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        List<Actividad> actividades = new ArrayList<>();
        actividades.add(new Actividad(1, 4));
        actividades.add(new Actividad(3, 5));
        actividades.add(new Actividad(0, 6));
        actividades.add(new Actividad(5, 7));
        actividades.add(new Actividad(3, 8));
```

```
        actividades.add(new Actividad(5, 9));
        actividades.add(new Actividad(6, 10));
        actividades.add(new Actividad(8, 11));
        actividades.add(new Actividad(8, 12));
        actividades.add(new Actividad(2, 13));
        actividades.add(new Actividad(12, 14));

        List<Actividad> seleccionadas = AlgoritmoSeleccionActividades.seleccionActividades(actividades);

        System.out.println("Actividades seleccionadas:");
        for (Actividad actividad : seleccionadas) {
            System.out.println(actividad);
        }
    }
}
```

del algoritmo voraz "Selección de Actividades"

CAPÍTULO VI

HEAPS

6.1. Introducción

Los heaps, o montículos, son estructuras de datos fundamentales en el ámbito de la informática y la ciencia de la computación. Son utilizados principalmente para implementar colas de prioridad y para la clasificación eficiente de elementos. En este artículo, exploraremos en detalle qué son los heaps, cómo funcionan, sus diferentes tipos y algunas de sus aplicaciones en el mundo real.

6.2. Definición y Características

Un heap es una estructura de datos basada en un árbol binario completo que cumple con la propiedad del heap: para cada nodo ii en el árbol, el valor almacenado en ii es mayor o menor que los valores almacenados en sus hijos, dependiendo de si es un max-heap o un min-heap, respectivamente.

Los heaps se dividen en dos categorías principales: max-heaps y min-heaps. En un max-heap, el valor almacenado en cada nodo es mayor o igual que los valores almacenados en sus hijos, mientras que en un min-heap, el valor en cada nodo es menor o igual que los valores en sus hijos.

Las operaciones principales en un heap son la inserción y la eliminación de elementos. Estas operaciones mantienen la propiedad del heap y garantizan que el árbol permanezca completo en todo momento.

6.3. Operaciones

Las operaciones principales en un heap incluyen:

• Inserción: Agregar un nuevo elemento al heap mientras se mantiene la propiedad del heap.

• Extracción: Eliminar y devolver el elemento con la máxima o mínima prioridad (en max-heaps y min-heaps, respectivamente).

• Disminución de la Clave: Reducir el valor de un elemento en el heap y reorganizar la estructura según sea necesario.

• Eliminar: Eliminar un elemento específico del heap.

• Heapify: Convertir una estructura de datos arbitraria en un heap válido.

Estas operaciones son fundamentales para la utilización efectiva de los heaps en algoritmos y aplicaciones.

6.4. Heaps Binarios

Los heaps binarios son una estructura de datos fundamental en el ámbito de la informática, especialmente en algoritmos de ordenamiento, estructuras de datos y algoritmos de grafos. Representan una forma eficiente de organizar datos, permitiendo operaciones rápidas de inserción, eliminación y recuperación del máximo o mínimo elemento, dependiendo del tipo de heap. En este artículo, exploraremos en detalle qué son los heaps

binarios, cómo funcionan, sus tipos, aplicaciones y su importancia en la ciencia computacional.

Un heap binario es un tipo particular de árbol binario completo, donde cada nodo tiene un valor mayor (o menor) que los valores de sus hijos, dependiendo del tipo de heap. Es decir, en un heap máximo, el valor de cada nodo es mayor o igual al valor de sus hijos, mientras que en un heap mínimo, el valor de cada nodo es menor o igual al valor de sus hijos. Esto se conoce como la propiedad del heap.

La estructura de un heap binario se puede representar de dos maneras principales: a través de un arreglo o mediante punteros. En la representación mediante arreglo, los nodos se almacenan secuencialmente en un arreglo, y se puede acceder a los hijos y al padre de un nodo en posiciones específicas del arreglo. En la representación mediante punteros, cada nodo tiene referencias a sus hijos y su padre.

6.4.1. Tipos de Heaps Binarios

Existen dos tipos principales de heaps binarios:

• Heap Máximo: En un heap máximo, el valor de cada nodo es mayor o igual al valor de sus hijos. Por lo tanto, el nodo raíz contiene el elemento máximo en el heap.

• Heap Mínimo: En un heap mínimo, el valor de cada nodo es menor o igual al valor de sus hijos. De esta manera, el nodo raíz contiene el elemento mínimo en el

heap.

La elección entre un heap máximo o mínimo depende de las necesidades del problema que se esté abordando.

6.4.2. Operaciones en Heaps Binarios

Los heaps binarios admiten varias operaciones fundamentales:

- Inserción: Agregar un nuevo elemento al heap manteniendo la propiedad del heap.
- Eliminación: Eliminar el máximo o mínimo elemento del heap y restaurar la propiedad del heap.
- Recuperación del Máximo/Mínimo: Obtener el máximo o mínimo elemento del heap sin eliminarlo.

Estas operaciones se realizan de manera eficiente en tiempo logarítmico en relación con la altura del árbol, lo que hace que los heaps binarios sean especialmente útiles en aplicaciones que requieren un rendimiento óptimo.

6.4.3. Aplicaciones de los Heaps Binarios

Los heaps binarios tienen una amplia gama de aplicaciones en informática, algunas de las cuales incluyen:

- Ordenamiento Eficiente: Los heaps binarios se utilizan en algoritmos de ordenamiento como heapsort, que ofrece una complejidad de tiempo de O(n log n) en

el peor de los casos.

- Implementación de Colas de Prioridad: Debido a su capacidad para recuperar rápidamente el máximo o mínimo elemento, los heaps binarios se utilizan en la implementación eficiente de colas de prioridad.
- Algoritmos de Grafos: En algoritmos como el algoritmo de Dijkstra para encontrar el camino más corto en un grafo ponderado, los heaps binarios se utilizan para mantener los nodos no explorados ordenados por su distancia desde el nodo de inicio.
- Programación Competitiva: Los heaps binarios son una herramienta esencial en programación competitiva, donde se utilizan para resolver problemas de optimización y búsqueda de manera eficiente.

Aplicación 1

Clase HeapBinario

```
/**
 *
 * @author Marcelo
 */
import java.util.ArrayList;
import java.util.List;

public class HeapBinario {
    private List<Integer> heap;

   public HeapBinario() {
      heap = new ArrayList<>();
```

```
    }

// Insertar un elemento en el heap
    public void insertar(int elemento) {
        heap.add(elemento);
        flotar(heap.size() - 1);
    }

    // Eliminar el elemento máximo (para un heap máximo)
    public int eliminarMaximo() {
        if (heap.isEmpty()) {
            throw new IllegalStateException("El heap está vacío");
        }

        int maximo = heap.get(0);
        int ultimo = heap.remove(heap.size() - 1);
        if (!heap.isEmpty()) {
            heap.set(0, ultimo);
            hundir(0);
        }
        return maximo;
    }

    // Método auxiliar para mantener la propiedad del heap al insertar un elemento
    private void flotar(int indice) {
        int padre = (indice - 1) / 2;
        while (indice > 0 && heap.get(indice) > heap.get(padre)) {
            intercambiar(indice, padre);
            indice = padre;
            padre = (indice - 1) / 2;
        }
    }
```



```
    // Método auxiliar para mantener la propiedad del heap al eliminar el máximo elemento
    private void hundir(int indice) {
        int hijoIzquierdo = 2 * indice + 1;
        int hijoDerecho = 2 * indice + 2;
        int maximo = indice;

        if (hijoIzquierdo < heap.size() && heap.get(hijoIzquierdo) > heap.get(maximo)) {
            maximo = hijoIzquierdo;
        }
        if (hijoDerecho < heap.size() && heap.get(hijoDerecho) > heap.get(maximo)) {
            maximo = hijoDerecho;
        }

        if (maximo != indice) {
            intercambiar(indice, maximo);
            hundir(maximo);
        }
    }

    // Método auxiliar para intercambiar dos elementos en el heap
    private void intercambiar(int i, int j) {
        int temp = heap.get(i);
        heap.set(i, heap.get(j));
        heap.set(j, temp);
    }
}
```

Clase Ejecutable

```
/**
 *
 * @author Marcelo
 */
public class Main {
    public static void main(String[] args) {
        HeapBinario heap = new HeapBinario();
        heap.insertar(10);
        heap.insertar(30);
        heap.insertar(20);
        heap.insertar(50);
        heap.insertar(40);

        System.out.println("Elemento máximo extraído: " + heap.eliminarMaximo());
        System.out.println("Elemento máximo extraído: " + heap.eliminarMaximo());
    }
}
```

Este código implementa un heap binario que admite la inserción de elementos y la extracción del máximo elemento. Se utilizan operaciones de flotación y hundimiento para mantener la propiedad del heap después de insertar y eliminar elementos. Puedes probar este código ejecutando la clase Ejecutable, que inserta algunos elementos en el heap y luego extrae el máximo elemento varias veces para demostrar su funcionamiento.

6.5. Heaps Binomiales

Los heaps binomiales son una estructura de datos en árbol utilizada para implementar colas de prioridad eficientes. Se distinguen por su estructura jerárquica y la

propiedad de ser autoorganizativos, lo que los hace ideales para operaciones de inserción, eliminación y búsqueda en una amplia gama de aplicaciones. Este artículo explora en profundidad los conceptos fundamentales de los heaps binomiales, sus propiedades, operaciones principales y aplicaciones en diversos campos.

6.5.1. Fundamentos de los Heaps Binomiales

Los heaps binomiales son una generalización de los árboles binarios completos y se caracterizan por seguir una estructura jerárquica basada en la combinación de árboles binomiales. Un árbol binomial de orden k se define recursivamente como un árbol binario formado por dos árboles binomiales de orden k-1, además de un nodo adicional que actúa como el nodo raíz del árbol binomial de orden k. Esta estructura permite una representación compacta de múltiples niveles de prioridad dentro de la cola de prioridad.

Los heaps binomiales se construyen a partir de árboles binomiales, cada uno representando un grado específico. Los grados de los árboles binomiales en un heap binomial actúan como coeficientes binomiales, lo que garantiza que cada grado esté presente en la estructura exactamente una vez. Esto facilita la combinación de dos heaps binomiales de la misma estructura, manteniendo así la propiedad de heap binomial.

6.5.2. Propiedades Clave

Los heaps binomiales poseen varias propiedades clave que los hacen altamente eficientes y útiles en aplicaciones prácticas:

•Estructura Jerárquica: Los heaps binomiales mantienen una estructura jerárquica clara, donde cada árbol binomial representa un grado específico de prioridad. Esto permite un acceso rápido a los elementos de acuerdo con su prioridad.

• Propiedad de Ordenación: Dentro de cada árbol binomial, los nodos están ordenados en función de su valor de clave, con el nodo raíz teniendo la prioridad más alta. Esto simplifica las operaciones de inserción, eliminación y búsqueda.

•Operaciones Eficientes: Las operaciones fundamentales en heaps binomiales, como inserción, eliminación y fusión, tienen complejidad temporal O(log n), donde n es el número de elementos en el heap binomial. Esto se debe a la estructura autoorganizativa y la naturaleza equilibrada de los árboles binomiales.

• Mantenimiento Automático: La propiedad de autoorganización de los heaps binomiales garantiza que se mantenga la estructura correcta del heap binomial después de cada operación. Esto elimina la necesidad de realizar ajustes manuales y mejora la eficiencia de la estructura.

6.5.3. Operaciones Principales

Los heaps binomiales admiten diversas operaciones que permiten la manipulación eficiente de la cola de prioridad:

• Inserción: Agrega un nuevo elemento al heap binomial, manteniendo la propiedad de heap binomial.

• Eliminación Mínima: Elimina y devuelve el elemento de mayor prioridad del heap binomial.

• Unión (o Fusión): Combina dos heaps binomiales en uno solo, manteniendo la propiedad de heap binomial.

• Decremento Clave: Reduce la prioridad de un elemento específico en el heap binomial, ajustando su posición según sea necesario.

• Búsqueda: Encuentra un elemento específico en el heap binomial, dada su clave.

6.5.4. Aplicaciones en Ciencias Computacionales

Los heaps binomiales tienen una amplia gama de aplicaciones en diversas áreas de la informática y la ingeniería. Algunas de estas aplicaciones incluyen:

• Colas de Prioridad: Los heaps binomiales son una estructura de datos fundamental para implementar colas de prioridad eficientes, utilizadas en algoritmos de búsqueda, grafos, algoritmos de enrutamiento y planificación de tareas.

- Algoritmos de Grafos: Se utilizan en algoritmos de búsqueda en grafos, como el algoritmo de Dijkstra y el algoritmo de Kruskal para encontrar caminos más cortos y árboles de expansión mínimos, respectivamente.

- Planificación de Procesos: Los heaps binomiales se utilizan en sistemas operativos para administrar la planificación de procesos y la asignación de recursos de manera eficiente.

- Optimización de Búsqueda: Son útiles en algoritmos de búsqueda en inteligencia artificial y aprendizaje automático, donde se requiere una selección eficiente de nodos o soluciones.

Aplicación Heap Binomial

Clase Ejecutable Binomial

```
/**
 *
 * @author Marcelo
 */
import java.util.ArrayList;
import java.util.List;

// Clase que representa un nodo en un árbol binomial
class BinomialNode {
    int valor;
    int grado;
   BinomialNode padre;
    List<BinomialNode> hijos;
   public BinomialNode(int valor) {
       this.valor = valor;
```



```
        this.grado = 0;
        this.padre = null;
        this.hijos = new ArrayList<>();
    }
}

// Clase que representa un heap binomial
public class BinomialHeap {
    private List<BinomialNode> raices;

    public BinomialHeap() {
        raices = new ArrayList<>();
    }

    // Insertar un valor en el heap binomial
    public void insertar(int valor) {
        BinomialNode nuevoNodo = new BinomialNode(valor);
        raices.add(nuevoNodo);
        consolidar();
    }

    // Combinar dos árboles binomiales del mismo grado
    private void unirArboles(BinomialNode nodo1, BinomialNode nodo2) {
        nodo1.padre = nodo2;
       nodo1.hijos.add(nodo2);
        nodo1.grado++;
    }

    // Consolidar el heap binomial después de insertar un nuevo nodo
    private void consolidar() {
      List<BinomialNode> nuevosRaices = new ArrayList<>();
        for (BinomialNode nodo : raices) {
            int grado = nodo.grado;
            while (grado >= nuevosRaices.size()) {
```

```
            nuevosRaices.add(null);
        }
        while (nuevosRaices.get(grado) != null) {
            BinomialNode otroNodo = nuevosRaices.get(grado);
            if (nodo.valor > otroNodo.valor) {
                BinomialNode temp = nodo;
                nodo = otroNodo;
                otroNodo = temp;
            }
            unirArboles(nodo, otroNodo);
            nuevosRaices.set(grado, null);
            grado++;
        }
        nuevosRaices.set(grado, nodo);
    }
    raices.clear();
    for (BinomialNode nodo : nuevosRaices) {
        if (nodo != null) {
            raices.add(nodo);
        }
    }
}

// Obtener el mínimo valor del heap binomial
public int obtenerMinimo() {
    if (raices.isEmpty()) {
        return Integer.MIN_VALUE;
    }
    int minimoValor = raices.get(0).valor;
    for (BinomialNode nodo : raices) {
        minimoValor = Math.min(minimoValor, nodo.valor);
    }
    return minimoValor;
}
```



```
    // Eliminar y devolver el mínimo valor del heap binomial
    public int eliminarMinimo() {
        if (raices.isEmpty()) {
            return Integer.MIN_VALUE;
        }
        int minimoValor = raices.get(0).valor;
        BinomialNode minimoNodo = raices.get(0);
        for (BinomialNode nodo : raices) {
            if (nodo.valor < minimoValor) {
                minimoValor = nodo.valor;
                minimoNodo = nodo;
            }
        }
        raices.remove(minimoNodo);
        BinomialHeap nuevoHeap = new BinomialHeap();
        for (BinomialNode hijo : minimoNodo.hijos) {
            hijo.padre = null;
            nuevoHeap.raices.add(hijo);
        }
        consolidar();
        return minimoValor;
    }

    // Combinar dos heaps binomiales
    public void combinar(BinomialHeap otroHeap) {
        raices.addAll(otroHeap.raices);
        consolidar();
    }

    // Verificar si el heap binomial está vacío
    public boolean estaVacio() {
        return raices.isEmpty();
    }
}
```

Esta implementación proporciona las operaciones básicas de un Heap Binomial, como inserción, eliminación del mínimo, y combinación de heaps binomiales. Puedes utilizar esta implementación como base y agregar otras funcionalidades según sea necesario para tu aplicación específica.

6.6. Heaps de Fibonacci

Los Heaps de Fibonacci son una estructura de datos avanzada que permite una implementación eficiente de diversas operaciones de montículos (heaps), como la inserción, la extracción del mínimo, la unión de montículos y la disminución de la clave. Se basan en la estructura de árboles binomiales y ofrecen tiempos de ejecución amortizados muy eficientes para estas operaciones, lo que los hace valiosos en aplicaciones donde se requiere un rendimiento óptimo, como algoritmos de grafos y algoritmos de programación dinámica. En este artículo, exploraremos en detalle los conceptos básicos, la estructura interna, las operaciones y las aplicaciones de los Heaps de Fibonacci.

6.6.1. Conceptos Básicos

Los Heaps de Fibonacci son una variante de los montículos (heaps) que se basan en la estructura de los árboles binomiales. La característica distintiva de los Heaps de Fibonacci es que permiten la unión rápida de dos montículos en tiempo constante amortizado y la disminución de la clave en tiempo constante amortizado.

Esto se logra mediante el uso de árboles binomiales que permiten unirse y dividirse eficientemente.

6.6.2. Estructura Interna

En un Heap de Fibonacci, los elementos se organizan en árboles binomiales, que son árboles binarios ordenados de manera especial. Cada árbol binomial de orden k tiene exactamente 2^k nodos. Además, los árboles binomiales en un Heap de Fibonacci se organizan en una lista enlazada circular doble, donde cada árbol apunta al siguiente árbol de mayor orden. El montículo mantiene un puntero al árbol con el valor mínimo (nodo raíz más pequeño).

6.6.3. Operaciones

Los Heaps de Fibonacci admiten varias operaciones principales:

- Inserción: Se inserta un nuevo elemento en el montículo creando un nuevo árbol binomial de orden cero y fusionándolo con la lista de árboles existente.

- Extracción del mínimo: Se elimina y devuelve el elemento mínimo del montículo. Para hacer esto, se fusionan todos los árboles binomiales en la lista principal y se actualiza el puntero al mínimo.

- Unión: Se fusionan dos montículos en un nuevo montículo en tiempo constante amortizado. Esto se logra simplemente concatenando las listas de árboles

binomiales de ambos montículos y ajustando el puntero al mínimo según sea necesario.

• Disminución de la clave: Se disminuye el valor de la clave de un elemento específico en el montículo. Esta operación se realiza alterando el valor del elemento y, si es necesario, cortando y moviendo el nodo hacia arriba en la estructura del árbol binomial para mantener la propiedad del montículo.

6.6.4. Aplicaciones

Los Heaps de Fibonacci tienen varias aplicaciones en algoritmos y estructuras de datos avanzados:

Algoritmos de grafos: Se utilizan en algoritmos como Dijkstra y el algoritmo de Prim para encontrar caminos mínimos y árboles de expansión mínimos en grafos ponderados.

Colas de prioridad: Se utilizan en implementaciones eficientes de colas de prioridad en algoritmos como el algoritmo de búsqueda A* y el algoritmo de Dijkstra.

Programación dinámica: Se pueden utilizar en algoritmos de programación dinámica para optimizar el uso de la memoria y mejorar el rendimiento en problemas como la optimización de rutas y la asignación de recursos.

Aplicación 1

Clase Ejecutabl Fibonacci

```
/**
 *
 * @author Marcelo
 */
import java.util.*;

public class FibonacciHeap<T extends Comparable<T>> {
    private Node<T> minNode;
    private int size;

    public FibonacciHeap() {
        this.minNode = null;
        this.size = 0;
    }

    public boolean isEmpty() {
        return minNode == null;
    }

    public void insert(T value) {
        Node<T> newNode = new Node<>(value);
       if (minNode != null) {
            newNode.next = minNode;
           newNode.prev = minNode.prev;
            minNode.prev = newNode;
            newNode.prev.next = newNode;
            if (value.compareTo(minNode.value) < 0) {
                minNode = newNode;
            }
       } else {
           minNode = newNode;
```

```
    }
    size++;
  }

  public T findMin() {
    if (isEmpty()) throw new NoSuchElementException("Heap is empty");
    return minNode.value;
  }

  public T deleteMin() {
    if (isEmpty()) throw new NoSuchElementException("Heap is empty");
    Node<T> oldMin = minNode;
    if (minNode.next == minNode) {
      minNode = null;
    } else {
      minNode.prev.next = minNode.next;
      minNode.next.prev = minNode.prev;
      minNode = minNode.next;
    }
    if (oldMin.child != null) {
      Node<T> current = oldMin.child;
      do {
        current.parent = null;
        current = current.next;
      } while (current != oldMin.child);
      minNode = mergeLists(minNode, oldMin.child);
    }
    if (minNode == null) return oldMin.value;
    List<Node<T>> treeTable = new ArrayList<>();
    List<Node<T>> toVisit = new ArrayList<>();
    for (Node<T> start = minNode; toVisit.isEmpty() || toVisit.get(0) != start; start =
start.next) {
      toVisit.add(start);
    }
```



```
    for (Node<T> current : toVisit) {
      while (true) {
        while (current.degree >= treeTable.size()) {
          treeTable.add(null);
        }
        if (treeTable.get(current.degree) == null) {
          treeTable.set(current.degree, current);
          break;
        }
        Node<T> other = treeTable.get(current.degree);
        treeTable.set(current.degree, null);
        Node<T> min = (current.value.compareTo(other.value) < 0) ? current : other;
        Node<T> max = (current.value.compareTo(other.value) < 0) ? other : current;
        max.next.prev = max.prev;
        max.prev.next = max.next;
        max.next = max.prev = max;
        min.child = mergeLists(min.child, max);
        max.parent = min;
        max.childCut = false;
        min.degree++;
        current = min;
      }
      if (current.value.compareTo(minNode.value) <= 0) minNode = current;
    }
    size--;
    return oldMin.value;
  }

  public void decreaseKey(Node<T> node, T newValue) {
    if (newValue.compareTo(node.value) > 0) {
      throw new IllegalArgumentException("New value is greater than current
value");
    }
    node.value = newValue;
```

```
        Node<T> parent = node.parent;
        if (parent != null && node.value.compareTo(parent.value) < 0) {
            cut(node, parent);
            cascadingCut(parent);
        }
        if (node.value.compareTo(minNode.value) < 0) {
            minNode = node;
        }
    }

    private void cut(Node<T> node, Node<T> parent) {
        node.childCut = false;
        if (node.next == node) {
            parent.child = null;
        } else {
            node.next.prev = node.prev;
            node.prev.next = node.next;
            parent.child = node.next;
        }
        parent.degree--;
        node.prev = node.next = node;
        minNode = mergeLists(minNode, node);
        node.parent = null;
    }

    private void cascadingCut(Node<T> node) {
        Node<T> parent = node.parent;
        if (parent != null) {
            if (!node.childCut) {
                node.childCut = true;
            } else {
                cut(node, parent);
                cascadingCut(parent);
            }
```



```
        }
    }

    private Node<T> mergeLists(Node<T> a, Node<T> b) {
        if (a == null && b == null) return null;
        if (a != null && b == null) return a;
        if (a == null) return b;
        Node<T> aNext = a.next;
        a.next = b.next;
        a.next.prev = a;
        b.next = aNext;
        b.next.prev = b;
        return a.value.compareTo(b.value) < 0 ? a : b;
    }

    private static class Node<T> {
        T value;
        Node<T> child;
        Node<T> parent;
        Node<T> next;
        Node<T> prev;
        boolean childCut;
        int degree;

        Node(T value) {
            this.value = value;
            next = prev = this;
        }
    }
}
```

Esta implementación proporciona las operaciones básicas necesarias para un heap de Fibonacci, como

inserción, eliminación del mínimo y disminución de clave. La estructura de datos se basa en nodos doblemente enlazados para facilitar las operaciones de fusión y eliminación. Puedes utilizar esta implementación como punto de partida para construir aplicaciones que requieran las propiedades específicas de un heap de Fibonacci.

6.7. Aplicaciones en el Mundo Real

Los heaps tienen una amplia gama de aplicaciones en la informática y más allá. Algunos ejemplos incluyen:

- Colas de Prioridad: Los heaps se utilizan para implementar colas de prioridad, donde los elementos con mayor prioridad (o menor, según el tipo de heap) se procesan primero.

- Algoritmos de Grafos: En algoritmos como Dijkstra y Prim, los heaps se utilizan para encontrar el camino más corto o el árbol de expansión mínimo, respectivamente.

Planificación de Procesos: Los heaps se utilizan en la planificación de procesos para asignar recursos de manera eficiente según la prioridad de los procesos.

CAPÍTULO VII

INTRODUCCIÓN A ALGORITMOS DE INTELIGENCIA ARTIFICIAL

7.1. Introducción

La inteligencia artificial (IA) es un campo de la informática que se centra en el desarrollo de sistemas capaces de realizar tareas que normalmente requieren inteligencia humana. A lo largo de las décadas, la IA ha experimentado un crecimiento impresionante y ha encontrado aplicaciones en una amplia gama de áreas, desde el procesamiento de lenguaje natural hasta la conducción autónoma de vehículos. En este artículo, exploraremos los fundamentos de la inteligencia artificial, sus aplicaciones, desafíos y el impacto que tiene en la sociedad.

7.2. Definición y Características

La inteligencia artificial se define como la capacidad de las máquinas para imitar el comportamiento humano y realizar tareas que normalmente requieren inteligencia humana, como el aprendizaje, la percepción, el razonamiento y la resolución de problemas. Uno de los objetivos principales de la IA es desarrollar sistemas que puedan tomar decisiones de manera autónoma, aprendiendo de la experiencia y mejorando con el tiempo.

Russell y Norvig (2016) definen la inteligencia artificial como “el estudio de cómo hacer que las computadoras realicen tareas que, hasta ahora, requieren inteligencia humana”. Esta definición destaca el enfoque de la IA en la automatización de tareas cognitivas y el aprendizaje automático.

7.3. Ramas de la Inteligencia Artificial

La inteligencia artificial abarca varias ramas y enfoques, algunos de los cuales incluyen:

• Aprendizaje Automático: Se centra en el desarrollo de algoritmos que permiten a las máquinas aprender de los datos y mejorar su rendimiento con la experiencia. El aprendizaje automático incluye técnicas como redes neuronales, árboles de decisión y algoritmos de clustering.

• Visión por Computadora: Se enfoca en desarrollar sistemas que pueden interpretar y analizar imágenes y videos. Esto incluye la detección de objetos, el reconocimiento facial y la segmentación de imágenes.

• Procesamiento de Lenguaje Natural: Se centra en el desarrollo de sistemas que pueden comprender y generar lenguaje humano de manera natural. Esto incluye la traducción automática, la generación de texto y el análisis de sentimientos.

• Robótica: Se refiere al diseño y construcción de robots inteligentes capaces de realizar tareas físicas y cognitivas. Esto incluye la navegación autónoma, la manipulación de objetos y la interacción social.

7.4. Aplicaciones de la Inteligencia Artificial

La inteligencia artificial tiene una amplia gama de aplicaciones en diversas industrias y sectores, algunas

de las cuales incluyen:

• Asistentes Virtuales: Los asistentes virtuales como Siri, Alexa y Google Assistant utilizan IA para comprender y responder a las consultas de los usuarios.

• Automatización Industrial: En la industria, la IA se utiliza para optimizar procesos de fabricación, predecir fallas en equipos y mejorar la eficiencia operativa.

• Medicina: En medicina, la IA se utiliza para el diagnóstico médico, la personalización de tratamientos y la investigación de nuevos fármacos.

• Finanzas: En el sector financiero, la IA se utiliza para el análisis de riesgos, la detección de fraudes y la toma de decisiones de inversión.

• Transporte: En el transporte, la IA se utiliza en sistemas de navegación autónoma, gestión de flotas y optimización de rutas.

7.5. Desafíos y Consideraciones Éticas

A pesar de los avances en inteligencia artificial, aún existen varios desafíos y consideraciones éticas que deben abordarse. Algunos de estos desafíos incluyen:

• Transparencia y Explicabilidad: Los modelos de IA a menudo son difíciles de interpretar, lo que plantea preocupaciones sobre la transparencia y la explicabilidad de las decisiones tomadas por estos sistemas.

• Equidad y Sesgo: Los algoritmos de IA pueden perpetuar sesgos existentes en los datos utilizados para entrenarlos, lo que puede llevar a decisiones injustas o discriminatorias.

• Privacidad y Seguridad: El uso de datos personales para entrenar modelos de IA plantea preocupaciones sobre la privacidad y la seguridad de la información.

7.6. Algoritmos de Inteligencia Artificial

El algoritmo de Dijkstra, desarrollado por el científico Edsger W. Dijkstra en 1956, es una herramienta fundamental en el campo de la teoría de grafos y la optimización de rutas. Este algoritmo se utiliza para encontrar el camino más corto desde un nodo de inicio a todos los demás nodos en un grafo ponderado y dirigido con pesos no negativos. En este artículo, exploraremos en detalle el funcionamiento del algoritmo de Dijkstra, sus aplicaciones y su impacto en diversos campos de la informática y la ingeniería.

7.6.1. Estructura

El algoritmo de Dijkstra se basa en el principio de búsqueda voraz, donde se selecciona el nodo más cercano al nodo de inicio en cada iteración y se actualizan las distancias mínimas conocidas a los demás nodos. A medida que avanza el algoritmo, se garantiza que las distancias mínimas calculadas hasta el momento sean las más cortas posibles.

Según Dijkstra (1959), el algoritmo se puede describir en los siguientes pasos:

• Inicializar todas las distancias desde el nodo de inicio como infinito, excepto la distancia del nodo de inicio a sí mismo, que es cero.

• Seleccionar el nodo con la distancia mínima no procesada y marcarlo como visitado.

• Para cada nodo adyacente al nodo seleccionado, actualizar la distancia mínima si la suma de la distancia desde el nodo seleccionado más el peso de la arista es menor que la distancia mínima conocida hasta ahora.

• Repetir los pasos 2 y 3 hasta que todos los nodos hayan sido visitados.

Al final del algoritmo, se obtiene un conjunto de distancias mínimas desde el nodo de inicio a todos los demás nodos en el grafo.

7.6.2. Aplicaciones

El algoritmo de Dijkstra tiene una amplia gama de aplicaciones en diversos campos:

• Redes de Transporte: Se utiliza para encontrar la ruta más corta entre dos ubicaciones en un sistema de transporte, como carreteras, ferrocarriles o redes de vuelo.

• Redes de Computadoras: Se utiliza para calcular

las rutas más eficientes en redes de computadoras, minimizando el tiempo de transmisión de datos.

• Logística: En la gestión de cadenas de suministro y la logística, se utiliza para optimizar las rutas de entrega y minimizar los costos de transporte.

• Sistemas de Navegación: Se utiliza en sistemas de navegación GPS para calcular las rutas más cortas y rápidas entre dos ubicaciones.

• Telecomunicaciones: En el enrutamiento de llamadas y la transmisión de datos, se utiliza para encontrar las rutas más eficientes a través de una red de telecomunicaciones.

A continuación se encuentra implementado las clases para el algoritmo de la ruta más corta o Dijkstra

Clase Nodo

```
/**
 *
 * @author Marcelo
*/
public class Nodo {
    private final String nombre;

    public Nodo(String nombre) {
        this.nombre = nombre;
    }

    public String getNombre() {
        return nombre;
```



```
    }

    @Override
    public String toString() {
        return nombre;
    }
}
```

Clase Arista

```
/**
 *
 * @author Marcelo
 */
public class Arista {
    private final Nodo origen;
    private final Nodo destino;
    private final int peso;

    public Arista(Nodo origen, Nodo destino, int peso) {
        this.origen = origen;
        this.destino = destino;
        this.peso = peso;
    }

    public Nodo getOrigen() {
        return origen;
    }

    public Nodo getDestino() {
        return destino;
    }
```

```
    public int getPeso() {
        return peso;
    }

    @Override
    public String toString() {
        return origen + " -> " + destino + ", peso: " + peso;
    }
}
```

Clase GrafoDirigido

```
/**
 *
 * @author Marcelo
 */
import java.util.*;

public class GrafoDirigido {
    private final Map<Nodo, List<Arista>> mapaAdyacencia;

    public GrafoDirigido() {
        mapaAdyacencia = new HashMap<>();
    }

    public void agregarNodo(Nodo nodo) {
        mapaAdyacencia.putIfAbsent(nodo, new ArrayList<>());
    }

    public void agregarArista(Nodo origen, Nodo destino, int peso) {
        mapaAdyacencia.get(origen).add(new Arista(origen, destino, peso));
    }
```



```
    public Map<Nodo, List<Arista>> getMapaAdyacencia() {
        return mapaAdyacencia;
    }

    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
        for (Map.Entry<Nodo, List<Arista>> entry : mapaAdyacencia.entrySet()) {
            sb.append(entry.getKey()).append(" -> ").append(entry.getValue()).appen-
d("\n");
        }
        return sb.toString();
    }
}
```

Clase Dijkstra

```
/**
 *
 * @author Marcelo
 */
import java.util.*;

public class Dijkstra {
    public static Map<Nodo, Integer> encontrarCaminoMasCorto(GrafoDirigido grafo,
Nodo origen) {
        Map<Nodo, Integer> distancias = new HashMap<>();
        Set<Nodo> visitados = new HashSet<>();
        PriorityQueue<Nodo> colaPrioridad = new PriorityQueue<>(Comparator.compa-
ringInt(distancias::get));
        for (Nodo nodo : grafo.getMapaAdyacencia().keySet()) {
            if (nodo.equals(origen)) {
                distancias.put(nodo, 0);
```

```
            } else {
                distancias.put(nodo, Integer.MAX_VALUE);
            }
            colaPrioridad.offer(nodo);
        }

        while (!colaPrioridad.isEmpty()) {
            Nodo nodoActual = colaPrioridad.poll();
            if (visitados.contains(nodoActual)) {
                continue;
            }
            for (Arista arista : grafo.getMapaAdyacencia().getOrDefault(nodoActual,
Collections.emptyList())) {
                int nuevaDistancia = distancias.get(nodoActual) + arista.getPeso();
                if (nuevaDistancia < distancias.get(arista.getDestino())) {
                    distancias.put(arista.getDestino(), nuevaDistancia);
                    colaPrioridad.offer(arista.getDestino());
                }
            }
            visitados.add(nodoActual);
        }

        return distancias;
    }
}
```


CAPÍTULO VIII

ALGORITMOS GENÉTICOS

Los algoritmos genéticos son una técnica de optimización y búsqueda inspirada en la teoría de la evolución y la genética. Fueron desarrollados por primera vez por John Holland en la década de 1960 y desde entonces se han convertido en una herramienta poderosa en el campo de la inteligencia artificial y la computación evolutiva. En esta sección, exploraremos en detalle los fundamentos de los algoritmos genéticos, su funcionamiento, aplicaciones y el impacto que han tenido en diversos campos de la ciencia y la ingeniería.

8.1. Conceptos Básicos

Los algoritmos genéticos se basan en la idea de la evolución biológica, donde las soluciones candidatas a un problema se representan como "individuos" en una "población" y se someten a operadores genéticos como selección, cruce y mutación para producir nuevas generaciones de individuos. Estos individuos compiten entre sí en función de su aptitud (o "fitness"), que es una medida de qué tan bueno es un individuo en términos de la solución al problema.

Según Holland (1975), los algoritmos genéticos se basan en tres principios fundamentales:

- Selección: Los individuos con una mayor aptitud tienen más probabilidades de ser seleccionados para reproducirse y producir descendencia.
- Cruce: Los individuos seleccionados se cruzan

entre sí para producir nuevos individuos, combinando características de ambos padres.

• Mutación: Ocasionalmente, se introducen cambios aleatorios en los individuos para explorar nuevas áreas del espacio de búsqueda.

8.2. Funcionamiento

El funcionamiento de un algoritmo genético se puede describir en los siguientes pasos:

• Inicialización: Se crea una población inicial de individuos de manera aleatoria o mediante un proceso heurístico.

• Evaluación de la Aptitud: Cada individuo en la población se evalúa en función de su aptitud para resolver el problema.

• Selección: Se seleccionan individuos de la población actual para reproducirse, con una probabilidad proporcional a su aptitud.

• Cruce: Los individuos seleccionados se cruzan entre sí, intercambiando información genética para producir nuevos individuos.

• Mutación: Ocasionalmente, se aplican cambios aleatorios a los individuos para mantener la diversidad genética.

• Reemplazo: Los nuevos individuos reemplazan a los

menos aptos en la población actual.

• Convergencia: El proceso se repite durante un número fijo de generaciones o hasta que se cumpla un criterio de convergencia.

8.3. Áreas Congitivas

Los algoritmos genéticos tienen una amplia gama de aplicaciones en diversas áreas, algunas de las cuales incluyen:

• Optimización: Se utilizan para encontrar soluciones óptimas en problemas de optimización combinatoria, como la planificación de rutas, el diseño de circuitos y la optimización de funciones matemáticas.

• Diseño de Sistemas: Se utilizan para diseñar sistemas complejos y encontrar configuraciones óptimas en ingeniería, diseño de productos y planificación de proyectos.

• Aprendizaje Automático: Se utilizan en el diseño de algoritmos de aprendizaje automático y de redes neuronales para encontrar modelos óptimos para conjuntos de datos.

• Biología Computacional: Se utilizan para estudiar y simular la evolución biológica y los procesos genéticos, como la evolución de poblaciones y la selección natural.

8.4. Impacto y Futuro

Los algoritmos genéticos han tenido un impacto significativo en la resolución de problemas complejos en una variedad de campos, gracias a su capacidad para encontrar soluciones óptimas en espacios de búsqueda enormes y complejos. A medida que continúan los avances en hardware y algoritmos, es probable que los algoritmos genéticos jueguen un papel aún más importante en la resolución de problemas del mundo real y en la búsqueda de soluciones innovadoras.

8.5. Aplicaciones

8.5.1. **Implementación del Algoritmo Genético con Optimización**

A continuación se presenta la implementación en el lenguaje de programación Java del algoritmo genético utilizando optimización

Clase Inidividuo

```
/**
 *
 * @author Marcelo
 */
import java.util.Random;

public class Individuo {
    private int[] genes;
    private int aptitud;
```



```
    public Individuo(int longitudGenes) {
        genes = new int[longitudGenes];
        Random rand = new Random();
        for (int i = 0; i < longitudGenes; i++) {
            genes[i] = rand.nextInt(2); // Genes binarios (0 o 1)
        }
        calcularAptitud();
    }

    public int[] getGenes() {
        return genes;
    }

    public int getAptitud() {
        return aptitud;
    }

    public void calcularAptitud() {
        aptitud = 0;
        for (int gen : genes) {
            aptitud += gen;
        }
    }
}
```

Clase Población

```
/**
 *
 * @author Marcelo
 */
import java.util.ArrayList;
import java.util.List;
```

```
public class Poblacion {
    private List<Individuo> individuos;

    public Poblacion(int tamanoPoblacion, int longitudGenes) {
        individuos = new ArrayList<>();
        for (int i = 0; i < tamanoPoblacion; i++) {
            individuos.add(new Individuo(longitudGenes));
        }
    }

    public List<Individuo> getIndividuos() {
        return individuos;
    }

    public void agregarIndividuo(Individuo individuo) {
        individuos.add(individuo);
    }
}
```

Clase AlgoritmoGenetico

```
/**
 *
 * @author Marcelo
 */
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class AlgoritmoGenetico {
    private double tasaMutacion;
    private int tamanoTorneo;
    private int longitudGenes;
```



```
    public AlgoritmoGenetico(double tasaMutacion, int tamanoTorneo, int longitud-
Genes) {
        this.tasaMutacion = tasaMutacion;
        this.tamanoTorneo = tamanoTorneo;
        this.longitudGenes = longitudGenes;
    }

    public Poblacion evolucionarPoblacion(Poblacion poblacion) {
        Poblacion nuevaPoblacion = new Poblacion(0, longitudGenes);

        // Mantener el mejor individuo
        Collections.sort(poblacion.getIndividuos(), (ind1, ind2) -> Integer.compare(ind2.
getAptitud(), ind1.getAptitud()));
        nuevaPoblacion.agregarIndividuo(poblacion.getIndividuos().get(0));

        // Reproducción
        while (nuevaPoblacion.getIndividuos().size() < poblacion.getIndividuos().size()) {
            Individuo padre1 = seleccionTorneo(poblacion);
            Individuo padre2 = seleccionTorneo(poblacion);
            Individuo hijo = cruzar(padre1, padre2);
            nuevaPoblacion.agregarIndividuo(hijo);
        }

        // Mutación
        for (Individuo individuo : nuevaPoblacion.getIndividuos()) {
            mutar(individuo);
        }

        return nuevaPoblacion;
    }
    private Individuo seleccionTorneo(Poblacion poblacion) {
        Poblacion torneo = new Poblacion(tamanoTorneo, longitudGenes);
```

```
        for (int i = 0; i < tamanoTorneo; i++) {
            int indiceAleatorio = (int) (Math.random() * poblacion.getIndividuos().size());
            torneo.agregarIndividuo(poblacion.getIndividuos().get(indiceAleatorio));
        }
        Collections.sort(torneo.getIndividuos(), (ind1, ind2) -> Integer.compare(ind2.
getAptitud(), ind1.getAptitud()));
        return torneo.getIndividuos().get(0);
    }

    private Individuo cruzar(Individuo padre1, Individuo padre2) {
        Individuo hijo = new Individuo(longitudGenes);
        int puntoCruce = (int) (Math.random() * longitudGenes);
        for (int i = 0; i < longitudGenes; i++) {
            if (i < puntoCruce) {
                hijo.getGenes()[i] = padre1.getGenes()[i];
            } else {
                hijo.getGenes()[i] = padre2.getGenes()[i];
            }
        }
        hijo.calcularAptitud();
        return hijo;
    }

    private void mutar(Individuo individuo) {
        for (int i = 0; i < longitudGenes; i++) {
            if (Math.random() < tasaMutacion) {
                int nuevoGen = (int) Math.round(Math.random());
                individuo.getGenes()[i] = nuevoGen;
            }
        }
        individuo.calcularAptitud();
    }
}
```


8.5.2. Implementación del Algoritmo Genético con Diseño de Sistemas

A continuación se presenta la implementación en el lenguaje de programación Java del algoritmo genético utilizando Diseño de Sistemas

Clase Sistema

```
/**
 *
 * @author Marcelo
 */
public class Sistema {
    private int[] configuracion;

    public Sistema(int[] configuracion) {
        this.configuracion = configuracion;
    }

    public int[] getConfiguracion() {
        return configuracion;
    }

    public void setConfiguracion(int[] configuracion) {
        this.configuracion = configuracion;
    }

    public int calcularAptitud() {
        // Simplemente como ejemplo, supongamos que la aptitud es la suma de los
elementos de la configuración
        int aptitud = 0;
        for (int valor : configuracion) {
            aptitud += valor;
```

```
        }
        return aptitud;
    }
}
```

Clase Individuo

```
/**
 *
 * @author Marcelo
 */
import java.util.Random;

public class Individuo {
    private int[] genes;
    private int aptitud;

    public Individuo(int longitudGenes) {
        genes = new int[longitudGenes];
        Random rand = new Random();
        for (int i = 0; i < longitudGenes; i++) {
            genes[i] = rand.nextInt(2); // Genes binarios (0 o 1)
        }
    }
    public int[] getGenes() {
        return genes;
    }

    public int getAptitud() {
        return aptitud;
    }

    public void calcularAptitud(Sistema sistema) {
        sistema.setConfiguracion(genes);
```



```
        aptitud = sistema.calcularAptitud();
    }
}
```

Clase Población

```
/**
 *
 * @author Marcelo
 */
import java.util.ArrayList;
import java.util.List;

public class Poblacion {
    private List<Individuo> individuos;

    public Poblacion(int tamanoPoblacion, int longitudGenes) {
        individuos = new ArrayList<>();
        for (int i = 0; i < tamanoPoblacion; i++) {
            individuos.add(new Individuo(longitudGenes));
        }
    }

    public List<Individuo> getIndividuos() {
        return individuos;
    }

    public void agregarIndividuo(Individuo individuo) {
        individuos.add(individuo);
    }
}
```

Clase AlgoritmoGenetico

```
/**
 *
 * @author Marcelo
 */
import java.util.Collections;
import java.util.List;

public class AlgoritmoGenetico {
    private double tasaMutacion;
    private int tamanoTorneo;
    private int longitudGenes;

    public AlgoritmoGenetico(double tasaMutacion, int tamanoTorneo, int longitud-
Genes) {
        this.tasaMutacion = tasaMutacion;
        this.tamanoTorneo = tamanoTorneo;
        this.longitudGenes = longitudGenes;
    }

    public Poblacion evolucionarPoblacion(Poblacion poblacion, Sistema sistema) {
        Poblacion nuevaPoblacion = new Poblacion(0, longitudGenes);

        // Reproducción
        while (nuevaPoblacion.getIndividuos().size() < poblacion.getIndividuos().size()) {
            Individuo padre1 = seleccionTorneo(poblacion, sistema);
            Individuo padre2 = seleccionTorneo(poblacion, sistema);
            Individuo hijo = cruzar(padre1, padre2);
            nuevaPoblacion.agregarIndividuo(hijo);
        }

        // Mutación
        for (Individuo individuo : nuevaPoblacion.getIndividuos()) {
```



```
            mutar(individuo);
        }

        return nuevaPoblacion;
    }

    private Individuo seleccionTorneo(Poblacion poblacion, Sistema sistema) {
        Collections.shuffle(poblacion.getIndividuos());
        Poblacion torneo = new Poblacion(tamanoTorneo, longitudGenes);
        torneo.getIndividuos().addAll(poblacion.getIndividuos().subList(0, tamanoTor-
neo));
        torneo.getIndividuos().forEach(individuo -> individuo.calcularAptitud(sistema));
        return Collections.max(torneo.getIndividuos(), (ind1, ind2) -> Integer.compa-
re(ind1.getAptitud(), ind2.getAptitud()));
    }

    private Individuo cruzar(Individuo padre1, Individuo padre2) {
        Individuo hijo = new Individuo(longitudGenes);
        for (int i = 0; i < longitudGenes; i++) {
            if (Math.random() < 0.5) {
                hijo.getGenes()[i] = padre1.getGenes()[i];
            } else {
                hijo.getGenes()[i] = padre2.getGenes()[i];
            }
        }
        return hijo;
    }

    private void mutar(Individuo individuo) {
        for (int i = 0; i < longitudGenes; i++) {
            if (Math.random() < tasaMutacion) {
                individuo.getGenes()[i] ^= 1; // Mutación de bits
            }
        }
```

```
    }
}
```

Clase NodoArbol

```
/**
 *
 * @author Marcelo
 */
public class Main {
    public static void main(String[] args) {
        int tamanoPoblacion = 10;
        int longitudGenes = 8;
        double tasaMutacion = 0.01;
        int tamanoTorneo = 3;
        int numGeneraciones = 100;

        Sistema sistema = new Sistema(new int[longitudGenes]); // Configuración inicial
del sistema
        AlgoritmoGenetico algoritmoGenetico = new AlgoritmoGenetico(tasaMutacion,
tamanoTorneo, longitudGenes);
        Poblacion poblacion = new Poblacion(tamanoPoblacion, longitudGenes);

        for (int i = 0; i < numGeneraciones; i++) {
            poblacion = algoritmoGenetico.evolucionarP
```

8.5.3. Implementación del Algoritmo Genético con Aprendizaje Automático

A continuación se presenta la implementación en el lenguaje de programación Java del algoritmo genético utilizando Aprendizaje Automático. En este ejemplo, se utiliza un conjunto de datos de entrenamiento datosX y

las etiquetas correspondientes etiquetasY. El algoritmo genético evoluciona una población de conjuntos de pesos para un modelo de regresión logística. La función de aptitud se define como el inverso del error cuadrático medio entre las predicciones y las etiquetas reales. A medida que evoluciona la población, el algoritmo genético busca maximizar esta función de aptitud, lo que conduce a conjuntos de pesos que minimizan el error de predicción en el conjunto de datos de entrenamiento.

Clase Ejecutable

```
/**
 *
 * @author Marcelo
 */
import java.util.Arrays;
import java.util.Random;

public class AlgoritmoGeneticoML {
    private static final int TAMANO_POBLACION = 50;
    private static final int NUM_GENERACIONES = 100;
    private static final double TASA_CRUCE = 0.7;
    private static final double TASA_MUTACION = 0.01;

    // Datos de entrenamiento
    private static final double[][] datosX = { {2.78, 2.55}, {1.47,
2.36}, {2.36, 1.82}, {3.04, 2.31}, {7.63, 2.75}, {5.33, 2.08}, {6.92, 1.77}, {8.67, 3.29},
{9.8, 2.99}, {7.68, 3.12} };
    private static final int[] etiquetasY = {0, 0, 0, 0, 1, 1, 1, 1, 1, 1};

    public static void main(String[] args) {
        // Inicializar población
```

```
        double[][] poblacion = inicializarPoblacion();

        // Evolucionar la población
        for (int generacion = 0; generacion < NUM_GENERACIONES; generacion++) {
            poblacion = evolucionarPoblacion(poblacion);
        }

        // Mostrar el mejor individuo (conjunto de pesos) encontrado
        double[] mejorIndividuo = obtenerMejorIndividuo(poblacion);
        System.out.println("Mejor conjunto de pesos encontrado: " + Arrays.toString(me-
jorIndividuo));
    }

    // Inicialización de la población con valores aleatorios
    private static double[][] inicializarPoblacion() {
        Random random = new Random();
        double[][] poblacion = new double[TAMANO_POBLACION][datosX[0].length +
1]; // +1 para el sesgo
        for (int i = 0; i < TAMANO_POBLACION; i++) {
            for (int j = 0; j < poblacion[i].length; j++) {
                poblacion[i][j] = random.nextDouble(); // Valores entre 0 y 1
            }
        }
        return poblacion;
    }

    // Evolución de la población
    private static double[][] evolucionarPoblacion(double[][] poblacion) {
        double[][] nuevaPoblacion = new double[TAMANO_POBLACION][poblacion[0].
length];
        for (int i = 0; i < TAMANO_POBLACION; i++) {
            double[] padre1 = seleccionTorneo(poblacion);
            double[] padre2 = seleccionTorneo(poblacion);
            double[] hijo = cruzar(padre1, padre2);
```



```
            mutar(hijo);
            nuevaPoblacion[i] = hijo;
        }
        return nuevaPoblacion;
    }

    // Selección por torneo
    private static double[] seleccionTorneo(double[][] poblacion) {
        Random random = new Random();
        int indicePadre1 = random.nextInt(TAMANO_POBLACION);
        int indicePadre2 = random.nextInt(TAMANO_POBLACION);
        return evaluarAptitud(poblacion[indicePadre1])
evaluarAptitud(poblacion[indicePadre2]) ? poblacion[indicePadre1] : poblacion[indi-
cePadre2];
    }

    // Cruce de dos individuos (padres) para generar un hijo
    private static double[] cruzar(double[] padre1, double[] padre2) {
        double[] hijo = new double[padre1.length];
        Random random = new Random();
        for (int i = 0; i < padre1.length; i++) {
            hijo[i] = random.nextDouble() < TASA_CRUCE ? padre1[i] : padre2[i];
        }
        return hijo;
    }

    // Mutación de un individuo (hijo)
    private static void mutar(double[] individuo) {
        Random random = new Random();
        for (int i = 0; i < individuo.length; i++) {
            if (random.nextDouble() < TASA_MUTACION) {
                individuo[i] = random.nextDouble();
            }
```

```
        }
    }

    // Función de aptitud (regresión logística)
    private static double evaluarAptitud(double[] individuo) {
        double aptitud = 0;
        for (int i = 0; i < datosX.length; i++) {
            double[] datos = datosX[i];
            double sumaPonderada = individuo[0]; // Sesgo
            for (int j = 0; j < datos.length; j++) {
                sumaPonderada += datos[j] * individuo[j + 1];
            }
            double prediccion = 1.0 / (1.0 + Math.exp(-sumaPonderada)); // Función
sigmoide
            aptitud += Math.pow(prediccion - etiquetasY[i], 2); // Error cuadrático
        }
        return 1 / (1 + aptitud); // Mayor aptitud para menor error
    }

    // Obtener el mejor individuo (conjunto de pesos) de la población
    private static double[] obtenerMejorIndividuo(double[][] poblacion) {
        double mejorAptitud = Double.MIN_VALUE;
        double[] mejorIndividuo = null;
        for (double[] individuo : poblacion) {
            double aptitud = evaluarAptitud(individuo);
            if (aptitud > mejorAptitud) {
                mejorAptitud = aptitud;
                mejorIndividuo = individuo;
            }
        }
          double prediccion = 1.0 / (1.0 + Math.exp(-sumaPonderada)); // Función
sigmoide
```



```
            aptitud += Math.pow(prediccion - etiquetasY[i], 2); // Error cuadrático
        }
        return 1 / (1 + aptitud); // Mayor aptitud para menor error
    }

    // Obtener el mejor individuo (conjunto de pesos) de la población
    private static double[] obtenerMejorIndividuo(double[][] poblacion) {
        double mejorAptitud = Double.MIN_VALUE;
        double[] mejorIndividuo = null;
        for (double[] individuo : poblacion) {
            double aptitud = evaluarAptitud(individuo);
            if (aptitud > mejorAptitud) {
                mejorAptitud = aptitud;
                mejorIndividuo = individuo;
            }
        }
        return mejorIndividuo;
    }
}
```

8.5.4. Implementación del Algoritmo Genético con Biología Computacional

A continuación se presenta la implementación en el lenguaje de programación Java del algoritmo genético utilizando Biología Computacional. Con estas clases básicas, puedes implementar un algoritmo genético para resolver problemas en biología computacional, como la optimización de secuencias de ADN para que coincidan con un patrón deseado. Puedes ajustar y expandir este código según sea necesario para adaptarlo a tu problema específico.

Clase Individuo

```
/**
 *
 * @author Marcelo
 */
import java.util.Random;

public class Individuo {
    private char[] genes;
    private String patronObjetivo;

    public Individuo(int longitud) {
        genes = new char[longitud];
        Random random = new Random();
        for (int i = 0; i < longitud; i++) {
            genes[i] = generarCaracterAleatorio();
        }
    }

    public Individuo(char[] genes) {
        this.genes = genes;
    }

    public char[] getGenes() {
        return genes;
    }

    public String obtenerADN() {
        return new String(genes);
    }

    public void calcularAptitud(String patronObjetivo) {
        int coincidencias = 0;
```



```
        for (int i = 0; i < genes.length; i++) {
            if (genes[i] == patronObjetivo.charAt(i)) {
                coincidencias++;
            }
        }
        this.patronObjetivo = patronObjetivo;
    }

    public int obtenerAptitud() {
        int coincidencias = 0;
        for (int i = 0; i < genes.length; i++) {
            if (genes[i] == patronObjetivo.charAt(i)) {
                coincidencias++;
            }
        }
        return coincidencias;
    }

    private char generarCaracterAleatorio() {
        Random random = new Random();
        int codigoAscii = random.nextInt(91 - 65) + 65; // Letras mayúsculas en ASCII
        return (char) codigoAscii;
    }

    public Individuo cruzar(Individuo otroIndividuo) {
        Random random = new Random();
        int puntoCruce = random.nextInt(genes.length);
        char[] genesHijo = new char[genes.length];
        for (int i = 0; i < genes.length; i++) {
            genesHijo[i] = i < puntoCruce ? genes[i] : otroIndividuo.getGenes()[i];
        }
        return new Individuo(genesHijo);
    }
```

```
    public void mutar(double tasaMutacion) {
        Random random = new Random();
        for (int i = 0; i < genes.length; i++) {
            if (random.nextDouble() < tasaMutacion) {
                genes[i] = generarCaracterAleatorio();
            }
        }
    }
}
```

Clase Poblacion

```
/**
 *
 * @author Marcelo
 */
import java.util.ArrayList;
import java.util.List;

public class Poblacion {
    private List<Individuo> individuos;

    public Poblacion(int tamano, int longitud) {
        individuos = new ArrayList<>();
        for (int i = 0; i < tamano; i++) {
            individuos.add(new Individuo(longitud));
        }
    }

    public List<Individuo> getIndividuos() {
        return individuos;
    }
```



```
    public void agregarIndividuo(Individuo individuo) {
        individuos.add(individuo);
    }
}
```

Clase AlgoritmoGenetico

```
/**
 *
 * @author Marcelo
 */
public class AlgoritmoGenetico {
    private double tasaMutacion;

    public AlgoritmoGenetico(double tasaMutacion) {
        this.tasaMutacion = tasaMutacion;
    }

    public Poblacion evolucionarPoblacion(Poblacion poblacion, String patronObjetivo)
{
        Poblacion nuevaPoblacion = new Poblacion(0, poblacion.getIndividuos().get(0).
getGenes().length);

        // Calcular la aptitud de cada individuo
        for (Individuo individuo : poblacion.getIndividuos()) {
            individuo.calcularAptitud(patronObjetivo);
        }

        // Reproducción y mutación
        while (nuevaPoblacion.getIndividuos().size()
poblacion.getIndividuos().size()) {
```

```
            Individuo padre1 = seleccionTorneo(poblacion);
            Individuo padre2 = seleccionTorneo(poblacion);
            Individuo hijo = padre1.cruzar(padre2);
            hijo.mutar(tasaMutacion);
            nuevaPoblacion.agregarIndividuo(hijo);
        }

        return nuevaPoblacion;
    }

    private Individuo seleccionTorneo(Poblacion poblacion) {
        Poblacion torneo = new Poblacion(3, poblacion.getIndividuos().get(0).getGenes().
length);
        for (int i = 0; i < 3; i++) {
            int indiceAleatorio = (int) (Math.random() * poblacion.getIndividuos().size());
            torneo.agregarIndividuo(poblacion.getIndividuos().get(indiceAleatorio));
        }
        Individuo mejorIndividuo = torneo.getIndividuos().get(0);
        for (Individuo individuo : torneo.getIndividuos()) {
            if (individuo.obtenerAptitud() > mejorIndividuo.obtenerAptitud()) {
                mejorIndividuo = individuo;
            }
        }
        return mejorIndividuo;
    }
}
```


EPILOGO

En el transcurso de este libro, se ha explorado las intrincadas profundidades de las estructuras de datos no lineales, desentrañando su complejidad y revelando su poder en la manipulación y gestión de datos en el ámbito computacional. Desde los árboles binarios hasta los grafos y heaps avanzados, navegando por un vasto océano de conceptos, algoritmos y aplicaciones prácticas.

En este epílogo, no solo se reflexiona sobre nuestro viaje a través de las estructuras de datos no lineales, sino que también proporciona un mirada hacia el futuro y exploraremos la intersección entre este fascinante campo y la evolución en constante cambio de la inteligencia artificial.

Un Viaje de Descubrimiento

A lo largo de este libro, se ha profundizado en un mundo de estructuras de datos complejas y potentes. Desde los conceptos fundamentales hasta las implementaciones prácticas en Java, se ha abordado una variedad de temas, desde árboles binarios hasta grafos y heaps de Fibonacci. Cada página ha sido una aventura de descubrimiento, proporcionando una comprensión más profunda de cómo estas estructuras pueden dar forma a la forma en que se interactúa con los datos en el mundo digital.

Un Vistazo al Futuro de la Inteligencia Artificial

Mientras se sumerge en el mundo de las estructuras de datos no lineales, no puede pasar por alto la creciente

influencia de la inteligencia artificial en nuestra sociedad. La inteligencia artificial, con sus capacidades para el aprendizaje automático, el procesamiento del lenguaje natural, la visión por computadora y más, está transformando rápidamente la forma en que se interactúa con la tecnología y entre nosotros.

La Convergencia de Estructuras de Datos e Inteligencia Artificial

La inteligencia artificial depende en gran medida de las estructuras de datos eficientes para almacenar, organizar y procesar datos a gran escala. Desde la representación de redes neuronales hasta la optimización de algoritmos de búsqueda, las estructuras de datos no lineales desempeñan un papel crucial en la infraestructura subyacente de la inteligencia artificial.

Desafíos y Oportunidades

A medida que avanzamos hacia el futuro, enfrentamos una serie de desafíos y oportunidades emocionantes en el ámbito de la inteligencia artificial y las estructuras de datos no lineales. Desde la optimización de algoritmos hasta la exploración de nuevas técnicas de aprendizaje automático, hay un vasto territorio de investigación y descubrimiento por delante.

Un Llamado a la Acción

Como expertos en ciencias computacionales, se tiene la responsabilidad y el privilegio de liderar este viaje

hacia el futuro. Al continuar explorando, investigando y colaborando en estos campos interdisciplinarios, podemos allanar el camino para nuevas innovaciones y avances en la ciencia y la tecnología.

Referencias

Adelson-Velsky, G. M., & Landis, E. M. (1962). An algorithm for the organization of information.

Aho, A. V., Ullman, J. D., & Hopcroft, J. E. (1983). Data Structures and Algorithms. Addison-Wesley.

Bang-Jensen, J., & Gutin, G. (2009). Digraphs: Theory, Algorithms and Applications. Springer Science & Business Media.

Bayer, R. (1972). Symmetric binary B-Trees: Data structure and maintenance algorithms.

Breiman, L. (2001). Random forests. Machine learning, 45(1), 5-32.

Comer, D. (1979). Ubiquitous B-Tree. ACM Computing Surveys (CSUR), 11(2), 121-137.

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

Deitel, P., & Deitel, H. (2017). Java: How to program (Early objects) (11th ed.). Pearson.

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. Numerische Mathematik, 1, 269-271.

Goodrich, M. T., & Tamassia, R. (2014). Data Structures and Algorithms in Java (6.a ed.). Wiley.

Horowitz, E., Sahni, S., & Anderson-Freed, S. (2013).

Fundamentals of Data Structures in C++. Silicon Press.

Jurafsky, D., & Martin, J. H. (2009). Speech and language processing (2nd ed.). Pearson Education.

Knuth, D. E. (1998). The Art of Computer Programming, Volume 3: Sorting and Searching (2.a ed.). Addison-Wesley.

Leiserson, C. E., Rivest, R. L., Cormen, T. H., & Stein, C. (2020). Introduction to Algorithms (4.a ed.). MIT Press.

Newman, M. E. J. (2010). Networks: an introduction. Oxford University Press.

Sedgewick, R., & Wayne, K. (2011). Algorithms (4.a ed.). Addison-Wesley.

Silberschatz, A., Korth, H. F., & Sudarshan, S. (2020). Database System Concepts (7.a ed.). McGraw-Hill Education.

ISBN: 978-9942-621-74-0