

**FACULTA DE CIENCIAS E INGENIERÍAS FÍSICAS Y FORMALES.**

**Escuela Profesional de Ingeniería de Sistemas.**

**Curso:**

**ALGORITMIA Y ESTRUCTURA DE DATOS.**

**TÉCNICAS DE DISEÑO DE ALGORITMOS**

**ACTIVIDADES – EJERCICIOS – TAREA.**

**Nuñez Arista Piero Bruker**

**Docente**

**MBA Mg. Ing. Rene Alonso Nieto Valencia.**

**Propuesta Trabajo**

**Versión 1.0**

**01 de marzo del 2025**

**Historial de Revisiones**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Fecha de Elaboración** | **Versión** | **Elaborado por:** | **Descripción** | **Revisado por:** | **Fecha de Revisión** |
| 15/05/2024 | 1.0 |  | Versión preliminar como propuesta. | Rene Alonso Nieto Valencia | 15/05/2024 |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

# **ÍNDICE**

[**ÍNDICE** 3](#_Toc201351786)

[**1.** **ACTIVIDAD 01.** 5](#_Toc201351787)

[**1.1.** **ENUNCIADO.** 5](#_Toc201351788)

[**1.2.** **ENLACE GITHUB/GITLAB.** 5](#_Toc201351789)

[**1.3.** **RESULTADOS.** 5](#_Toc201351790)

[**2.** **EJERCICIO 01.** 8](#_Toc201351791)

[**2.1.** **ENUNCIADO.** 8](#_Toc201351792)

[**2.2.** **ENLACE GITHUB/GITLAB.** 8](#_Toc201351793)

[**2.3.** **RESULTADOS.** 8](#_Toc201351794)

[**2.4.** **CÓDIGO FUENTE DOCUMENTADO.** 9](#_Toc201351795)

[**2.5.** **EXPLICACIÓN.** 13](#_Toc201351796)

[**3.** **EJERCICIO 02.** 13](#_Toc201351797)

[**3.1.** **ENUNCIADO.** 13](#_Toc201351798)

[**3.2.** **ENLACE GITHUB/GITLAB.** 13](#_Toc201351799)

[**3.3.** **RESULTADOS.** 13](#_Toc201351800)

[**3.4.** **CÓDIGO FUENTE DOCUMENTADO.** 13](#_Toc201351801)

[**4.** **EJERCICIO 03.** 17](#_Toc201351802)

[**4.1.** **ENUNCIADO.** 17](#_Toc201351803)

[**4.2.** **ENLACE GITHUB/GITLAB.** 17](#_Toc201351804)

[**4.3.** **RESULTADOS.** 18](#_Toc201351805)

[**4.4.** **CÓDIGO FUENTE DOCUMENTADO.** 18](#_Toc201351806)

[**4.5.** **EXPLICACIÓN.** 22](#_Toc201351807)

[**5.** **EJERCICIO 04.** 22](#_Toc201351808)

[**5.1.** **ENUNCIADO.** 22](#_Toc201351809)

[**5.2.** **ENLACE GITHUB/GITLAB.** 22](#_Toc201351810)

[**5.3.** **RESULTADOS.** 22](#_Toc201351811)

[**5.4.** **CÓDIGO FUENTE DOCUMENTADO.** 23](#_Toc201351812)

[**5.5.** **EXPLICACIÓN.** 25](#_Toc201351813)

[**REFERENCIAS Y BIBLIOGRAFÍA.** 26](#_Toc201351814)

[**ANEXOS.** 27](#_Toc201351815)

# **ACTIVIDAD 01.**

## **ENUNCIADO.**

En una hoja realice los siguientes ejercicios: 1.1.Inserte lassiguientes claves en un árbol B de orden 5 inicialmente vacío en el siguiente orden: 100, 50, 20, 70, 10, 30, 80, 90, 200, 25, 15, 5, 65, 35, 60, 18, 93,94..

## **ENLACE GITHUB/GITLAB.**

null

## **RESULTADOS.**

1. Complete la siguiente tabla conforme usted vaya realizando el proceso de inserción y cada vez ocurra una división complete la siguiente tabla:

* La clave que al ser insertada provoca la división (Inserción K)
* La clave mediana que debe ser promovida al padre (Key Mediana)
* Claves de nodo que está siendo divido (Keys Nodo)
* Claves del nuevo nodo creado (Keys Nuevo)
* ¿La división se propaga? (¿Se propagala división?) Si/No. Si la respuesta es Sí, complete el siguiente reglón desde la primera columna.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Inserción K** | **Key Mediana** | **Keys Nodo** | **Keys Nuevos** | **¿Se propaga la división?** |
| 100 |  |  |  | No |
| 50 |  |  |  | No |
| 20 |  |  |  | No |
| 70 |  |  |  | No |
| 10 | 50 | 20, 50, 70, 100 | (10, 20), (70, 100) | Si |
| 30 |  |  |  | No |
| 80 |  |  |  | No |
| 90 |  |  |  | No |
| 200 | 90 | 70, 80, 90, 100 | 100, 200 | Si |
| 25 |  |  |  | No |
| 15 | 20 | 10, 20, 25, 30 | 25, 30 | Si |
| 5 |  |  |  | No |
| 65 |  |  |  | No |
| 35 |  |  |  | No |
| 60 |  |  |  | No |
| 18 |  |  |  | No |
| 93 |  |  |  | No |
| 94 |  |  |  | No |

# **ACTIVIDAD 2**

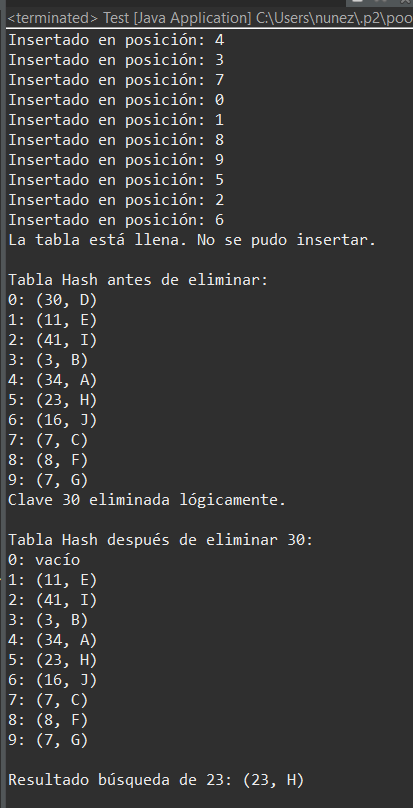
# **ENUNCIADO.**

Implementa un sistema de almacenamiento de registros mediante una tabla hash de tamaño fijo, utilizando la técnica de hash cerrado con sondeo lineal. La estructura almacenará objetos de tipo `Register`, que contienen un `int key` y un `String name`. 2.1 Objetivos 1. Crear una clase Register para representar los elementos almacenados. 2. Crear una clase HashC que implemente una tabla hash cerrada. 3. Implementar inserción, búsqueda y eliminación lógica. 4. Mostrar el estado actual de la tabla. 5. Probar la implementación con valores repetidos y colisiones.

Clase de prueba para insertar los siguientes valores: 34, 3, 7, 30, 11, 8, 7, 23, 41, 16, 34 (con diferentes nombres). Debe mostrar la tabla hash antes y después de eliminar la clave 30 y buscar la clave 23.

# **ENLACE GITHUB.**

# **RESULTADOS.**



# **CODIGO FUENTE.**

Texto

El contenido generado por IA puede ser incorrecto.

Texto

El contenido generado por IA puede ser incorrecto.

Texto

El contenido generado por IA puede ser incorrecto.

# **EXPLICACION.**

# **ACTIVIDAD 3**

# **ENUNCIADO.**

Implementando Hash abierto (encadenamiento) Implementa una tabla hash con encadenamiento (hash abierto), donde cada celda contiene una lista enlazada de elementos que colisionan en el mismo índice. La estructura almacenará objetos de tipo Register, compuestos por una clave entera (key) y un nombre (name).

1. Crear una clase Register que almacene la clave y el nombre.

3. Probar el comportamiento insertando elementos con colisiones.

4. Mostrar el contenido de cada lista de la tabla hash.

# **ENLACE GITHUB.**

# **RESULTADOS.**

Texto

El contenido generado por IA puede ser incorrecto.

# **CODIGO FUENTE.**

Texto

El contenido generado por IA puede ser incorrecto.

Texto

El contenido generado por IA puede ser incorrecto.

# **EXPLICACION.**

# **EJERCICIO 01.**

## **ENUNCIADO.**

Insertar sin colisiones Crea una tabla hash de tamaño 7 para almacenar números enteros. Usa la función hash h(x) = x % 7. 1. Inserta los siguientes valores: 3, 10, 17, 24. 2. Muestra la tabla hash final..

## **ENLACE GITHUB/GITLAB.**

<https://github.com/pierobruker/ALGORITMOS-Y-ESTRUCTURA-DE-DATOS/tree/main/Sesion10/src>

## **RESULTADOS.**

## **CÓDIGO FUENTE DOCUMENTADO.**

Lo mismo de la Actividad, pero se implementaron más métodos. **Clase BTree.java**

* **search(E key)**  
  Método público que inicia la búsqueda de una clave en el árbol B.  
  Si encuentra la clave, imprime el idNode y la posición de la clave en ese nodo.  
  Devuelve true si se encuentra, false en otro caso.
* **searchNode(BNode<E> current, E key)**  
  Método privado recursivo.  
  Busca la clave en el nodo actual. Si no la encuentra, desciende al hijo correspondiente.  
  Devuelve true si encuentra la clave, false si no.

## **EXPLICACIÓN.**

El método search permite localizar una clave en el árbol B de manera eficiente.  
La búsqueda compara la clave con las del nodo actual y decide el hijo al que debe descender.  
Gracias a la estructura balanceada del B-Tree, esta operación es muy rápida, incluso con grandes conjuntos de datos.  
El método también facilita depuración al mostrar el nodo y posición donde se encuentra la clave.

# **EJERCICIO 02.**

## **ENUNCIADO.**

Resolver colisiones con sondeo lineal Implementa una tabla hash con tamaño 6 usando hash cerrado (sondeo lineal). Usa h(x) = x % 6. 1. Inserta los valores: 12, 18, 24, 30. 2. Muestra la tabla paso a paso. 3. Explica qué pasa cuando hay una colisión.

## **ENLACE GITHUB/GITLAB.**

<https://github.com/pierobruker/ALGORITMOS-Y-ESTRUCTURA-DE-DATOS/tree/main/Sesion10/src>

## **RESULTADOS.**

## **CÓDIGO FUENTE DOCUMENTADO.**

**Lo mismo de la Actividad, pero se implementaron más métodos.**

**Clase BTree.java.**

* **remove(E key)**  
  Método público que inicia el proceso de eliminación de una clave.  
  Si el árbol está vacío, no realiza ninguna acción.
* **removeNode(BNode<E> current, E key)**  
  Busca la clave y la elimina.  
  Si es necesario, reemplaza la clave por su sucesor (cuando la clave está en un nodo interno).  
  Si el nodo queda con menos del mínimo de claves, llama a fixNodeAfterDeletion.
* f**ixNodeAfterDeletion(BNode<E> parent, int childPos)**  
  Determina si se debe redistribuir o fusionar el nodo que quedó con pocas claves.
* **redistributeWithLeftSibling / redistributeWithRightSibling**  
  Realizan préstamo de una clave de un hermano para restaurar el número mínimo de claves.
* **mergeWithLeftSibling / mergeWithRightSibling**  
  Fusionan el nodo con su hermano y ajustan las claves del padre.

**Clases de prueba TestEje3.java**

* **Realiza eliminaciones de claves del árbol y muestra el árbol resultante después de cada operación**.

**Explicación:**

El método remove asegura que al eliminar una clave el árbol mantenga sus propiedades:  
Si un nodo queda con menos del mínimo de claves, intenta redistribuir claves con sus hermanos.  
Si no es posible redistribuir, fusiona nodos y ajusta el padre.  
Esto mantiene el árbol balanceado y eficiente en sus operaciones posteriores.

# **EJERCICIO 03.**

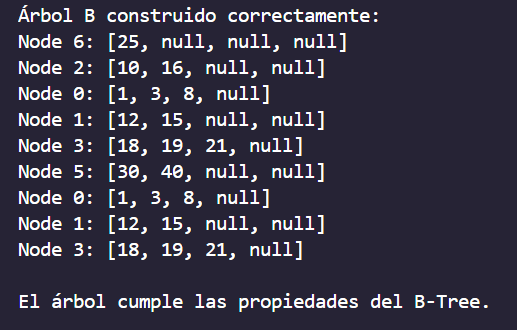
## **ENUNCIADO.**

A partir de los datos que se encuentra en el archivo arbolB.txt en el que se almacenalainformación de un BTree, en la clase BTree agregue un método de clase building\_Btree() apartir del cual se construya el correspondiente árbol y se verifique que cumpla todas laspropiedades de un BTree. Este método debe devolver el árbol BTree correspondiente en caso si cumpla contodaslas propiedades, en caso contrario deberá levantar una excepción del tipo ItemNoFound. El archivo arbolB.txt debe tener la siguiente estructura: Linea 1: orden del árbol Line 2: Nivel del nodo, idNodo, Conjunto de claves (separado por comas) Line 3: Nivel del nodo, idNodo, Conjunto de claves (separado por comas).

## **ENLACE GITHUB/GITLAB.**

<https://github.com/pierobruker/ALGORITMOS-Y-ESTRUCTURA-DE-DATOS/tree/main/Sesion10/src>

## **RESULTADOS.**



## **CÓDIGO FUENTE DOCUMENTADO.**

Texto

El contenido generado por IA puede ser incorrecto.

* **Lo mismo de la Actividad, pero se implementaron más métodos.**
* **Clase BTree.java.**
* **building\_Btree(String filename)**  
  Método estático que construye un árbol B a partir de un archivo.  
  Lee el orden y los nodos del árbol, los enlaza y valida la estructura.  
  Lanza ItemNoFound si el archivo tiene errores o el árbol no es válido.
* **validateBTree()**  
  Verifica que el árbol cumpla las propiedades del B-Tree:

Número correcto de claves e hijos por nodo.

Claves ordenadas.

Nodos hoja en el mismo nivel.

* **validateNode(BNode<E> node, int currentLevel, int[] leafLevel)**  
  Método privado recursivo que implementa la validación en cada nodo.
* **Clases de prueba TestEje3.java**
* Construye el árbol desde un archivo arbolB.txt.
* Muestra el árbol y el resultado de la validación.

## **EXPLICACIÓN.**

Este método permite construir un B-Tree a partir de datos almacenados en un archivo.  
La validación garantiza que el árbol generado sea correcto y funcional.  
Este mecanismo es útil para inicializar el árbol desde archivos de configuración o bases de datos externas.  
El sistema evita el uso de estructuras corruptas al validar el árbol después de construirlo.

# **EJERCICIO 04.**

## **ENUNCIADO.**

Se desea implementar un sistema de búsqueda eficiente que permita consultar estudiantesregistrados en la universidad a través de su código universitario. Se usará un árbol Bdeorden 4 que almacene objetos de tipo RegistroEstudiante, el cual contiene.

## **ENLACE GITHUB/GITLAB.**

<https://github.com/pierobruker/ALGORITMOS-Y-ESTRUCTURA-DE-DATOS/tree/main/Sesion10/src>

## **RESULTADOS.**

## **CÓDIGO FUENTE DOCUMENTADO.**

**Lo mismo de la Actividad, pero se implementaron más métodos.**

**Clase BTree.java:**

**buscarNombre(int codigo)**  
Busca un estudiante por código y devuelve el nombre si existe o "No encontrado" si no.

**buscarRegistro(BNode<E> node, E target)**  
Método de soporte recursivo que localiza el nodo y clave buscados.

**Clase RegistroEstudiante.java**

* **compareTo(RegistroEstudiante otro)**  
  Compara dos estudiantes por su código.
* **toString()**  
  Muestra el código y nombre del estudiante.
* **getCodigo() / getNombre()**  
  Acceden al código y nombre respectivamente.

**Clases de prueba TestEje4.java**

* Inserta estudiantes con distintos códigos y nombres.
* Realiza búsquedas por código.
* Elimina un estudiante.
* Inserta un nuevo estudiante y verifica su inserción.

## **EXPLICACIÓN.**

Este ejercicio muestra cómo el B-Tree puede aplicarse a un caso práctico:  
Almacenar registros de estudiantes ordenados por su código.  
Permite búsquedas eficientes de nombres por código gracias al balance del árbol.  
Mantiene el árbol ordenado y balanceado a pesar de inserciones y eliminaciones.  
Demuestra cómo una estructura genérica puede adaptarse a un dominio concreto como la gestión de registros..

# **REFERENCIAS Y BIBLIOGRAFÍA.**

1. Goodrich, M. T., & Tamassia, R. (2014). *Data structures and algorithms in Java* (6th ed.). Wiley.
2. Lafore, R. (2002). *Data structures and algorithms in Java* (2nd ed.). Sams Publishing.
3. Drozdek, A. (2012). *Data structures and algorithms in C++* (4th ed.). Cengage Learning.

# **ANEXOS.**