**Trabajo Práctico Integrador**

09 de junio de 2025

Programación I // Comisión 14

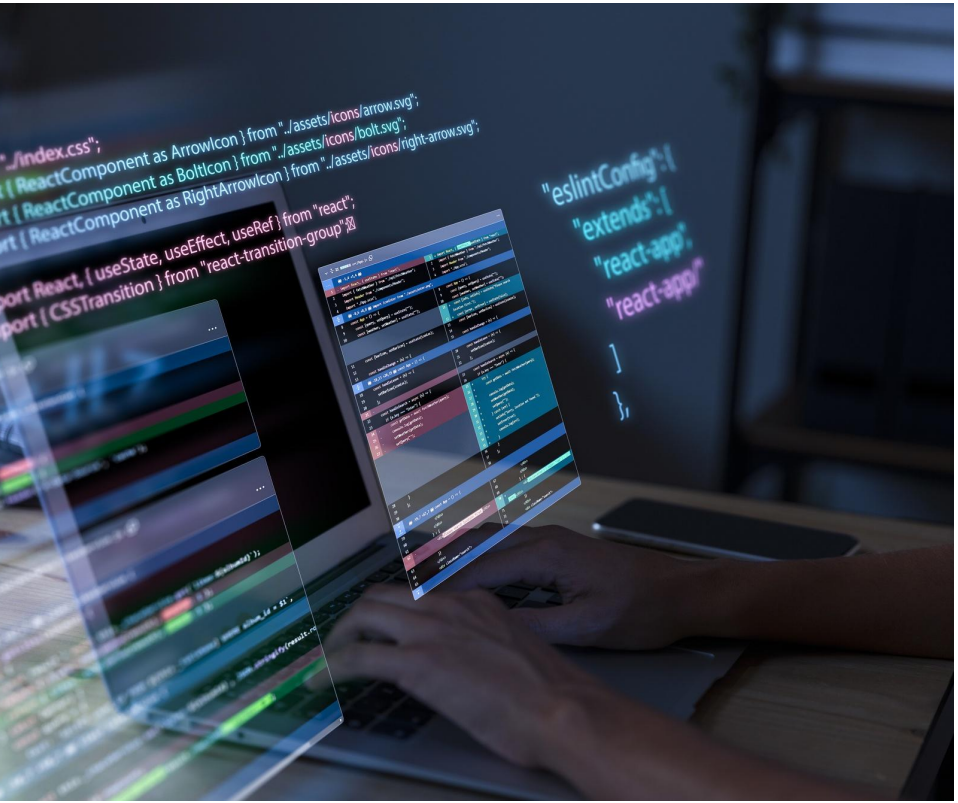
# Estructuras de Datos Avanzadas: Árboles

**Federico Garcia**

**Federico Garcia Bengolea**

**Profesor:**
**Ariel Enferrel**
**Tutor:**
**Ramiro Hualpa**

# Contenido:




Estructuras de Datos Avanzadas: Árboles

# Introducción

Las estructuras de datos son modelos organizativos fundamentales que **permiten almacenar y manipular información de manera eficiente**. Se utilizan en diversas áreas, desde la programación cotidiana hasta la optimización en bases de datos y sistemas de archivos. Se eligió este tema ya que impacta directamente en el rendimiento de los sistemas, favoreciendo la velocidad de procesamiento, consumo de memoria y escalabilidad.

El objetivo de este trabajo es demostrar, mediante una implementación en Python basada únicamente en listas, las diferencias en rendimiento entre estructura BST y Arbol-B, evaluando su comportamiento en inserción, búsqueda y eficiencia computacional.

Estructuras de Datos Avanzadas: Árboles

# Marco Teórico

Las estructuras de datos representan la forma en la que se organizan y manipulan estos mismos en la informática. Su importancia radica en la eficiencia y por este motivo son un pilar en el desarrollo de algoritmos.

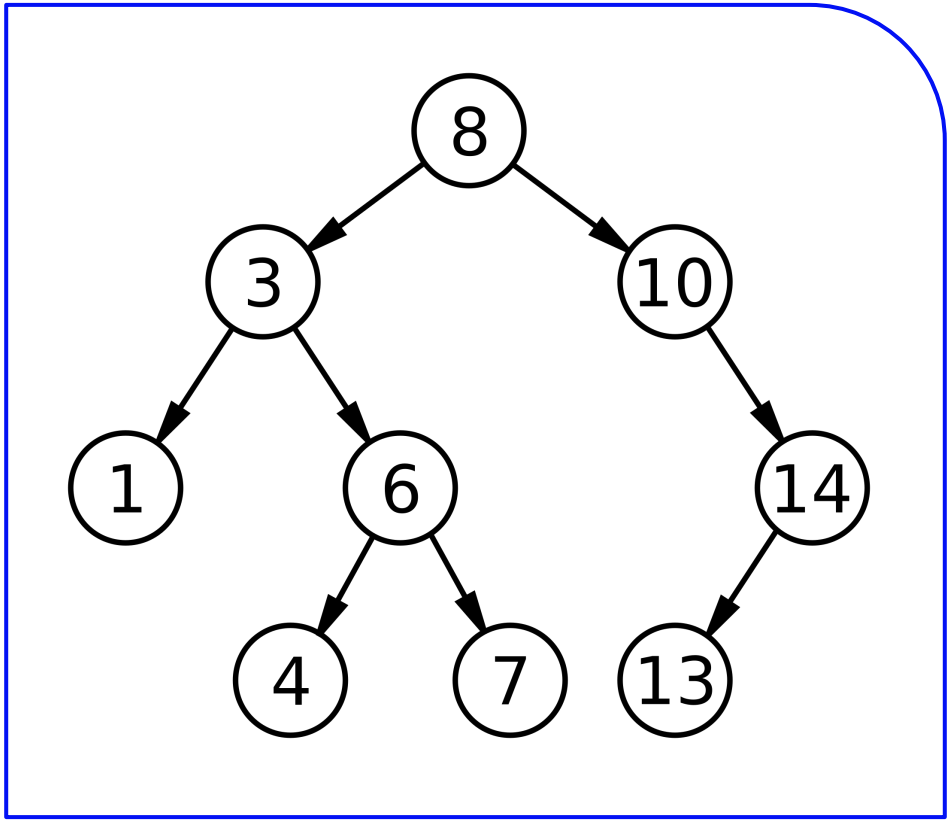
En bases de datos y almacenamiento de información, la elección de una estructura adecuada afecta aspectos clave como tiempo de acceso, consumo de memoria y escalabilidad. Los árboles son una de las opciones más utilizadas debido a su capacidad para organizar datos jerárquicamente y permitir operaciones optimizadas.

## Estructuras lineales

organizan los elementos de manera secuencial (listas, pilas, colas).

## Estructuras jerárquicas o no lineales

permiten relaciones complejas entre datos, como árboles y grafos.




# Árbol Binario de Búsqueda (BST)

Un BST es una estructura jerárquica donde cada nodo puede tener como máximo dos hijos (árboles de orden 2) y estos se ordenan específicamente siguiendo el siguiente patrón: si el hijo es de menor valor que el padre se sitúa en la rama izquierda, caso contrario se sitúa en la rama derecha. Lo mismo con los nodos hijos de cualquiera de estos hijos y así sucesivamente.

**Este diseño permite realizar búsquedas eficientes, ya que cada consulta reduce el espacio de búsqueda a la mitad. El rendimiento de esta estructura se ve afectada cuando el árbol no se encuentra balanceado.**

Este tipo de árbol realiza operaciones principales como la búsqueda, inserción y eliminación. Es importante recordar que la inserción debe garantizar el orden del BTS y que la eliminación puede llevar a la reestructuración de gran parte del árbol si este tenía múltiples hijos.

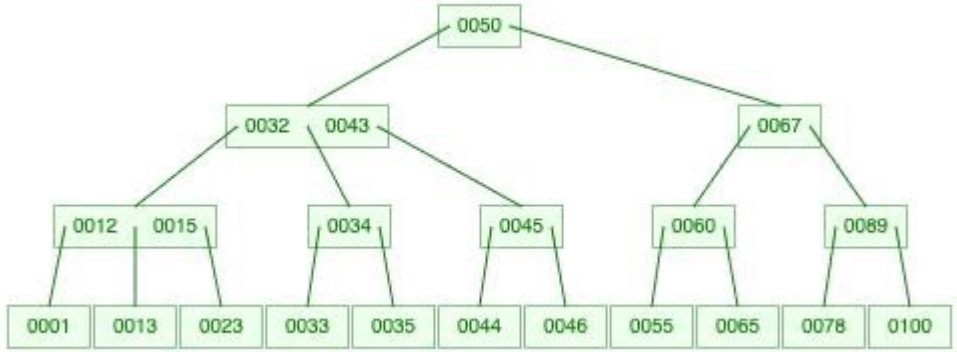
## Operaciones Principales:

**Búsqueda*:*** Se compara el valor buscado con el nodo actual y se avanza según la regla del árbol.

**Inserción:** Se coloca el nuevo nodo respetando la organización del BST.

**Eliminación:** Puede implicar reestructuración si el nodo tiene múltiples hijos.

**Árbol B de grado 3**




# Árbol B

El Árbol B es una estructura balanceada a diferencia del BST, diseñada para almacenamiento masivo y acceso eficiente a la memoria. Cada nodo puede contener múltiples claves y tener más de dos hijos, lo que lo hace ideal para bases de datos y sistemas de archivos.

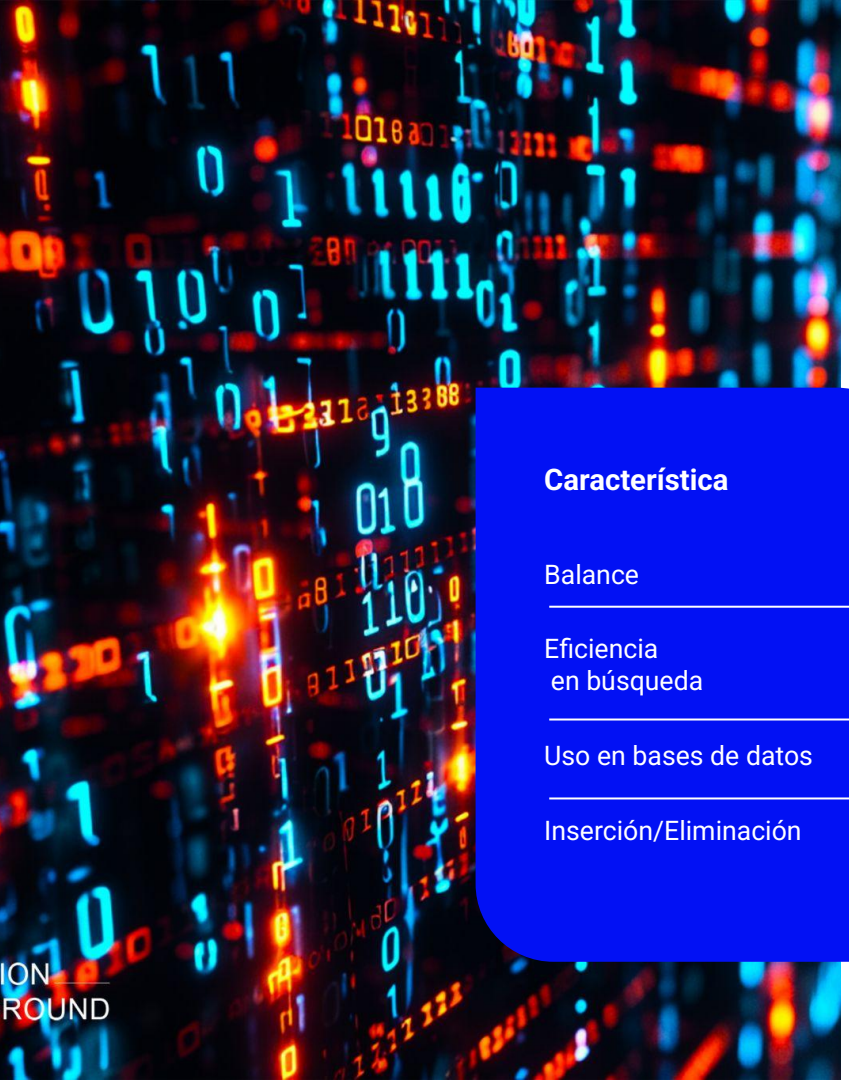
**Las operaciones que puede realizar este árbol son de búsqueda, inserción y eliminación, similar al árbol BST pero con un enfoque diferente, ya que utiliza las claves de los nodos para mantener el balance, re organizar y dividir dependiendo el escenario requerido.**

## Operaciones Principales:

**Búsqueda*:*** Se analiza el nodo actual antes de descender en la estructura.

**Inserción:** Si un nodo excede su capacidad, se divide en dos y se reorganiza el árbol.

**Eliminación:** Se redistribuyen claves entre nodos para mantener el balance.




Estructuras de Datos Avanzadas: Árboles

# Resumiendo

| Característica | BST | Árbol B |
|---|---|---|
| Balance | No garantizado | Siempre balanceado |
| Eficiencia en búsqueda | Depende del balance | Optimizado para bases de datos |
| Uso en bases de datos | Poco eficiente | Excelente rendimiento |
| Inserción/Eliminación | Puede ser costosa si se desbalancea | Optimizado en almacenamiento masivo |

Estructuras de Datos Avanzadas: Árboles

# Caso Práctico

**El objetivo fue evaluar la diferencia de rendimiento entre un BST y un Árbol B, implementados en Python con listas. Se busca analizar el comportamiento de cada estructura en operaciones de inserción y búsqueda, considerando cómo afecta el balanceo en la eficiencia computacional.**

Estructuras de Datos Avanzadas: Árboles

# BST en Python con Listas

**Inserta un valor en el BST respetando la estructura de listas.**

```
def insert_node_bts(tree, value):
if not tree:
return [value,[],[]]
if value == tree[0]:
return tree
if value < tree[0]:
tree[1] = insert_node_bts(tree[1], value)
else:
tree[2] = insert_node_bts(tree[2], value)
return tree
```

Estructuras de Datos Avanzadas: Árboles

# BST en Python con Listas

**Realiza una búsqueda en el BST.**

```
if not tree:
return False
if value == tree[0]:
return True
if value < tree[0]:
return search_node_bst(tree[1], value)
else:
return search_node_bst(tree[2], value)
```

Estructuras de Datos Avanzadas: Árboles

# Árbol B en Python con Listas

**Inserta un valor en el Árbol B, manteniendo el orden.**

```
def insert_node_b(tree, value, grade=3):
if not tree:
return [value, []]

new_tree = insert_in_node(tree, value, grade)
if isinstance(new_tree, tuple):
prom_key, (left, right) = new_tree
return [prom_key, [left, right]]

return new_tree

def insert_in_node(node, value, grade):
keys = node[:-1].copy()
children = node[-1].copy()

if not children:
keys.append(value)
keys.sort()

if len(keys) == grade:
mid = grade // 2
prom_key = keys[mid]

left_keys = keys[:mid]
right_keys = keys[mid + 1:]

left_node = left_keys + [[]]
right_node = right_keys + [[]]

return prom_key, (left_node, right_node)
else:
```

```
return keys + [[]]

else:
i = 0
while i < len(keys) and value > keys[i]:
i += 1

resp = insert_in_node(children[i], value, grade)

if isinstance(resp, tuple):
prom_key, (left, right) = resp
keys.insert(i, prom_key)
children[i] = left
children.insert(i + 1, right)

if len(keys) == grade:
mid = grade // 2
prom_key = keys[mid]

left_keys = keys[:mid]
left_children = children[:mid + 1]
left_node = left_keys + [left_children]

right_keys = keys[mid + 1:]
right_children = children[mid + 1:]
right_node = right_keys + [right_children]

return prom_key, (left_node, right_node)
else:
return keys + [children]
else:
children[i] = resp
```

Estructuras de Datos Avanzadas: Árboles

# Árbol B en Python con Listas

**Búsqueda en Árbol B.**

```
def search_node_b(node, value):
if not node:
return False

keys = node[:-1]
children = node[-1]

i = 0
while i < len(keys) and value >
keys[i]:
i += 1

if i < len(keys) and value == keys[i]:
return True

if not children:
return False

return search_node_b(children[i],
value)
```

Para la implementación de estos árboles en Python, **consideraremos en el caso del BST** que cada nodo se representará como una lista “[valor, izquierda, derecha]”, donde el “valor” es el dato almacenado, “izquierda” será otra lista representando el subárbol izquierdo y “derecha” será otra lista representando el subárbol derecho.

Para el caso del **Árbol B** cada nodo se hizo una lista con múltiples claves y referencias a hijos, indicada como “[clave1, clave2, ..., hijos]”, donde hijos es una lista de subárboles. Cada nodo tiene un número máximo de claves, según el grado definido.

# Metodología Utilizada

**01**

Se realizó una búsqueda de información (videos tutoriales, artículos publicados en internet, documentación oficial, consultas en IA).

**02**

Se realizo la selección del tema, dada su relevancia en bases de datos y optimización del almacenamiento.

**03**

Se establecieron los requisitos del código, asegurando que la implementación se realizara con estructuras de listas en Python. Luego se diseñaron funciones de inserción y búsqueda, considerando la estructura lógica de cada tipo de árbol.

**04**

Se registraron los resultados de cada ejecución, evaluando el rendimiento de los árboles en términos de inserción, búsqueda y eficiencia computacional.

**05**

Se implemento el método pair-programming remoto en donde pudimos realizar todo lo previo mencionado.

Estructuras de Datos Avanzadas: Árboles

# Resultados Obtenidos

**BST mostró tiempos de inserción más rápidos en estructuras pequeñas,** pero se volvió ineficiente cuando el árbol perdió balance, simulando el comportamiento de una lista enlazada. Esto afectó negativamente la búsqueda de elementos en árboles grandes.

**El Árbol B mantuvo estabilidad en el tiempo de búsqueda,** incluso con grandes volúmenes de datos. La organización con múltiples claves en cada nodo permitió minimizar la profundidad del árbol y mejorar el acceso a la información.

**Las mediciones de rendimiento confirmaron que el BST funciona bien en datos pequeños y organizados,** mientras que el Árbol B es más adecuado para bases de datos y almacenamiento masivo, optimizando la gestión de información a largo plazo.

Estructuras de Datos Avanzadas: Árboles

# Conclusiones

El trabajo práctico permitió aprender dos enfoques en estructuras de datos diferentes con aplicaciones en bases de datos. Mientras que el **BST ofrece rapidez en entornos pequeños**, su falta de autobalance pudo demostrar que afecta la eficiencia en datos desorganizados. Por otro lado, el **Árbol B garantiza tiempos de búsqueda estables**, siendo una opción clave en bases de datos de gran volumen.

Los resultados obtenidos reflejan que el **BST es más adecuado para el procesamiento en memoria**, mientras que el **Árbol B se ajusta mejor en el almacenamiento masivo**, optimizando accesos y minimizando costos computacionales.

# Conclusiones

## Análisis de Datos

| Datos | |
|---|---|
| Cantidad de elementos en los árboles | 500000 |
| Grado del árbol tipo B | 3 |
| Elemento existente buscado en ámbos árboles | 400335 |
| Elemento NO existente buscado en ámbos árboles | 1305636 |
| Elemento medio existente buscado en ámbos árboles | 250000 |

| Tiempos | |
|---|---|
| Tiempo en árbol BTS para elemento existente | 0.000009060 |
| Tiempo en árbol B para elemento existente | 0.000005960 |
| Tiempo en árbol BTS para elemento NO existente | 0.000002623 |
| Tiempo en árbol B para elemento NO existente | 0.000000954 |
| Tiempo en árbol BTS para elemento medio | 0.000006199 |
| Tiempo en árbol B para elemento medio | 0.000000000 |

**Promedio de búsquedas en árbol BTS: 0.000005960**

**Promedio de búsquedas en árbol B: 0.000002305**

# Conclusiones

## Análisis de Datos

| Datos | |
|---|---|
| Cantidad de elementos en los árboles | 1000000 |
| Grado del árbol tipo B | 3 |
| Elemento existente buscado en ámbos árboles | 967982 |
| Elemento NO existente buscado en ámbos árboles | 2702798 |
| Elemento medio existente buscado en ámbos árboles | 500000 |

| Tiempos | |
|---|---|
| Tiempo en árbol BTS para elemento existente | 0.000012159 |
| Tiempo en árbol B para elemento existente | 0.000005960 |
| Tiempo en árbol BTS para elemento NO existente | 0.000004053 |
| Tiempo en árbol B para elemento NO existente | 0.000000954 |
| Tiempo en árbol BTS para elemento medio | 0.000005007 |
| Tiempo en árbol B para elemento medio | 0.000000000 |

**Promedio de búsquedas en árbol BTS: 0.000007073**

**Promedio de búsquedas en árbol B: 0.000002305**

# Conclusiones

## Análisis de Datos

| Datos | |
|---|---|
| Cantidad de elementos en los árboles | 1500000 |
| Grado del árbol tipo B | 3 |
| Elemento existente buscado en ámbos árboles | 1418887 |
| Elemento NO existente buscado en ámbos árboles | 3052759 |
| Elemento medio existente buscado en ámbos árboles | 750000 |

| Tiempos | |
|---|---|
| Tiempo en árbol BTS para elemento existente | 0.000012159 |
| Tiempo en árbol B para elemento existente | 0.000005960 |
| Tiempo en árbol BTS para elemento NO existente | 0.000006199 |
| Tiempo en árbol B para elemento NO existente | 0.000000954 |
| Tiempo en árbol BTS para elemento medio | 0.000005960 |
| Tiempo en árbol B para elemento medio | 0.000000000 |

**Promedio de búsquedas en árbol BTS: 0.000008106**

**Promedio de búsquedas en árbol B: 0.000002305**

# Conclusiones

## Análisis de Datos

| Datos | |
|---|---|
| Cantidad de elementos en los árboles | 2000000 |
| Grado del árbol tipo B | 3 |
| Elemento existente buscado en ámbos árboles | 1624076 |
| Elemento NO existente buscado en ámbos árboles | 4922332 |
| Elemento medio existente buscado en ámbos árboles | 1000000 |

| Tiempos | |
|---|---|
| Tiempo en árbol BTS para elemento existente | 0.000007868 |
| Tiempo en árbol B para elemento existente | 0.000005960 |
| Tiempo en árbol BTS para elemento NO existente | 0.000003815 |
| Tiempo en árbol B para elemento NO existente | 0.000000000 |
| Tiempo en árbol BTS para elemento medio | 0.000008106 |
| Tiempo en árbol B para elemento medio | 0.000000000 |

**Promedio de búsquedas en árbol BTS: 0.000006596**

**Promedio de búsquedas en árbol B: 0.000001987**

**Trabajo Práctico Integrador**

09 de junio de 2025

Programación I // Comisión 14

# ¡Gracias!

**Federico Garcia**
garcia.federico@outlook.com

**Federico Garcia Bengolea**
feddericogarciaa@gmail.com

**Profesor:**
**Diego Lobos**
**Tutor:**
**Nicolas Carcaño**