PUCE Virtual

CLASE

4

ESTRUCTURA DE DATOS (PROGRAMACIÓN III)

Árboles

QS WORLD UNIVERSITY RANKINGS

Educación de calidad

INTRODUCCIÓN DE LA CLASE

GRADO / POSGRADO / TECNOLOGÍAS

Estudia a tu tiempo
son interrupciones

EDUCACIÓN EN LÍNEA DE CALIDAD

La estructura de datos conocida como árbol constituye uno de los pilares fundamentales en el diseño de sistemas informáticos eficientes y escalables. A diferencia de las estructuras lineales como arreglos o listas, los árboles permiten representar información de forma jerárquica, modelando relaciones de dependencia entre elementos. Esta característica resulta especialmente útil en contextos donde los datos no siguen una secuencia simple, sino que se organizan en niveles, categorías o ramas interrelacionadas. En términos computacionales, los árboles facilitan operaciones de búsqueda, inserción y eliminación con mayor eficiencia, especialmente cuando se emplean variantes optimizadas como los árboles binarios de búsqueda o los árboles balanceados.

En el ámbito de la ciberseguridad y los sistemas de alto rendimiento, los árboles desempeñan un papel estratégico. Los centros de operaciones de seguridad (SOC) deben gestionar grandes volúmenes de eventos provenientes de múltiples fuentes, organizarlos dinámicamente y permitir consultas rápidas para detectar amenazas en tiempo real. En estos escenarios, la estructura de árbol permite mantener datos ordenados y accesibles con complejidad logarítmica, reduciendo significativamente los tiempos de respuesta frente a incidentes críticos. Asimismo, los árboles son fundamentales en la implementación de índices en bases de datos, en la gestión de sistemas de archivos y en la construcción de modelos de decisión para detección de anomalías.

Comprender el funcionamiento conceptual de los árboles, sus propiedades, sus variantes y los mecanismos de balanceo es esencial para desarrollar soluciones robustas, eficientes y escalables en entornos tecnológicos modernos.

Semana 4:
Desarrollar algoritmos y estructuras de datos avanzadas para la resolución de problemas de complejidad media-alta, considerando criterios de eficiencia computacional en escenarios de ciberseguridad.

4. Árboles

4.1. Introducción a árboles

Un árbol constituye uno de los pilares fundamentales en la informática avanzada, especialmente en contextos donde la organización eficiente de grandes volúmenes de información es crítica, como en la ciberseguridad. En los centros de operaciones de seguridad (SOC), se reciben millones de eventos diarios provenientes de firewalls, sistemas IDS/IPS, servidores y dispositivos de red. Estos datos deben clasificarse, indexarse y consultarse rápidamente para detectar amenazas en tiempo real. Los árboles permiten representar relaciones jerárquicas —como categorías de incidentes, niveles de severidad o estructuras de directorios en análisis forense— de forma natural y eficiente.

Los árboles binarios de búsqueda (BST) facilitan operaciones de inserción y consulta en O(log n) en promedio, optimizando la detección de patrones sospechosos o búsqueda de direcciones IP maliciosas. Sin embargo, si el árbol se desbalancea, el rendimiento puede degradarse. Por ello, estructuras balanceadas como los árboles AVL garantizan estabilidad y escalabilidad, elementos esenciales en infraestructuras de defensa digital modernas.

4.1.1. Concepto de árbol y terminología

Un árbol es una estructura de datos no lineal compuesta por nodos conectados mediante enlaces, diseñada para representar información organizada de forma jerárquica. A diferencia de las estructuras lineales como arreglos o listas, en un árbol los datos se distribuyen en niveles, permitiendo modelar relaciones de dependencia y clasificación.
Existe un único nodo inicial llamado raíz, que actúa como punto de partida. Cada nodo puede tener cero o más nodos hijos, formando ramas que se extienden hacia distintos niveles. No existen ciclos, lo que significa que no se puede regresar al mismo nodo siguiendo los enlaces. Además, todo nodo, excepto la raíz, tiene exactamente un padre (Cormen et al., 2009).

Terminología básica

- **Nodo**: Elemento que almacena un valor o información.
- **Raíz**: Nodo principal del árbol.
- **Nodo hoja**: Nodo que no tiene hijos.
- **Padre / Hijo**: Relación jerárquica entre nodos conectados.
- **Nivel**: Distancia desde la raíz hasta un nodo específico.
- **Altura**: Longitud del camino más largo desde la raíz hasta una hoja.
- **Subárbol**: Árbol formado por un nodo y todos sus descendientes.

La Figura 1 muestra un ejemplo de un árbol binario con la terminología que se usa.

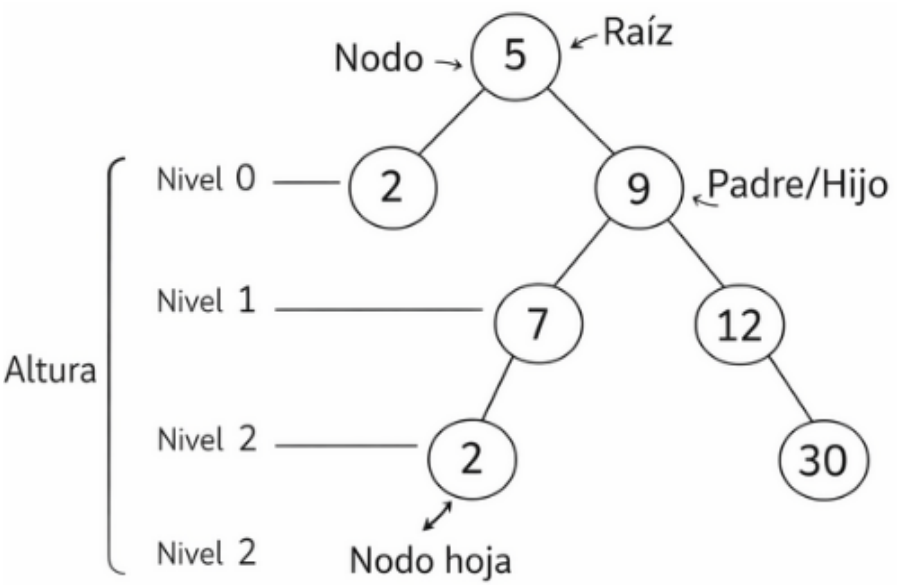


Figura 1. Terminología en árboles binarios. Creación de autor Patricio Coba

Los árboles permiten implementar búsquedas eficientes y organizar datos de forma dinámica, especialmente cuando se utilizan variantes como los árboles binarios de búsqueda y árboles balanceados (Weiss, 2013).

En ciberseguridad, los árboles permiten organizar información compleja de manera estructurada y eficiente. Se utilizan para representar jerarquías organizacionales en la gestión de accesos, modelar estructuras de directorios durante análisis forense digital y construir árboles de decisión para la detección automatizada de amenazas. También facilitan la clasificación jerárquica de incidentes según tipo, severidad y origen, permitiendo un análisis más rápido y preciso en entornos donde la eficiencia y el tiempo de respuesta son críticos.

4.1.2. Árboles binarios

Un árbol binario es un tipo especial de árbol en el que cada nodo tiene como máximo dos hijos: un hijo izquierdo y un hijo derecho. Esta restricción estructural permite organizar la información de manera ordenada y facilita la implementación de algoritmos eficientes para búsqueda, inserción y recorrido. Los árboles binarios constituyen la base de estructuras más avanzadas, como los árboles binarios de búsqueda (BST), árboles AVL y árboles Red-Black, ampliamente utilizados en sistemas que requieren acceso rápido a la información (Cormen et al., 2009).

La Figura 2 muestra un ejemplo de un árbol binario.

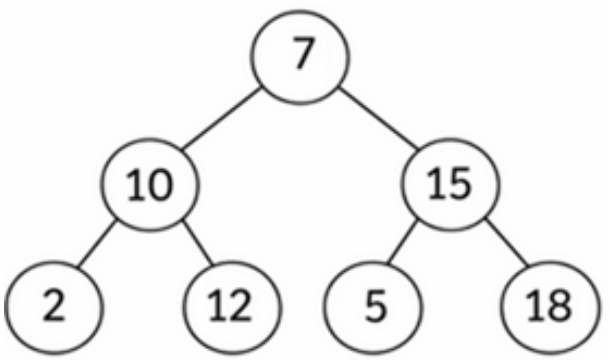


Figura 2. Ejemplo de árbol binario. Creación de autor Patricio Coba

Características:
Puede estar completo, cuando todos los niveles están llenos excepto posiblemente el último, estar lleno, cuando cada nodo tiene exactamente cero o dos hijos, ser degenerado, cuando cada nodo tiene un solo hijo, comportándose como una lista enlazada, Se utiliza como base para estructuras más complejas y eficientes y permite recorridos estructurados que facilitan el procesamiento sistemático de los datos (Weiss, 2013).

Tipos de recorrido

- **Preorden** (Raíz – Izquierda – Derecha): útil para copiar estructuras.
- **Inorden** (Izquierda – Raíz – Derecha): produce datos ordenados en un BST.
- **Postorden** (Izquierda – Derecha – Raíz): utilizado para eliminar árboles o evaluar expresiones.

Del ejemplo mostrado en la Figura 2 a continuación se presentan los recorridos:

- PreOrden: 7, 10, 2, 12, 15, 5, 18
- InOrden: 2, 10, 12, 7, 5, 15, 18
- PostOrden: 2, 12, 10, 5, 18, 15, 7

En ciberseguridad, los árboles binarios permiten modelar la evaluación de expresiones lógicas en reglas de firewall, donde cada nodo representa una condición. También se emplean en la construcción de árboles de decisión en sistemas IDS (Intrusion Detection Systems) para clasificar comportamientos sospechosos. Además, facilitan el procesamiento jerárquico de políticas de seguridad, permitiendo aplicar reglas de manera estructurada y eficiente en infraestructuras complejas.

4.1.3. Árboles binarios de búsqueda (BST)

Un **Binary Search Tree (BST)** es un árbol binario que cumple la propiedad de orden:

- Todos los valores del subárbol izquierdo son menores que el nodo.
- Todos los valores del subárbol derecho son mayores que el nodo.

Esta propiedad permite realizar búsquedas eficientes.

La figura 3 muestra un ejemplo de un BTS donde se puede ver que los nodos de la izquierda tienen elementos menores a los de la derecha.

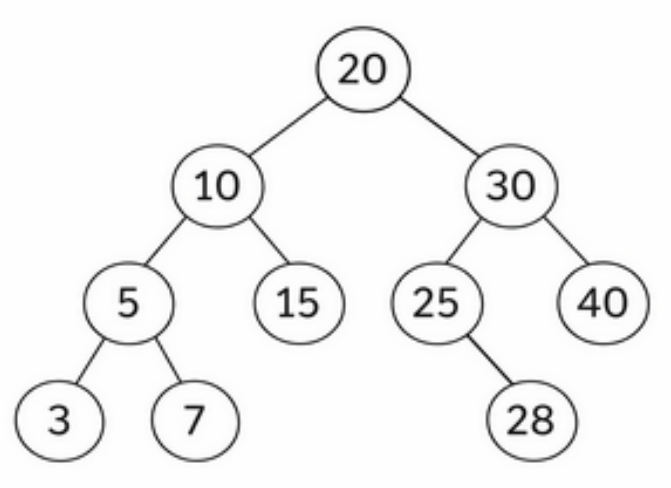


Figura 3. Ejemplo de árbol binario de búsqueda

Complejidad

- Búsqueda: O(log n) promedio.
- Inserción: O(log n) promedio.
- Peor caso (árbol desbalanceado): O(n).

Aplicación en ciberseguridad

- Indexación de direcciones IP.
- Almacenamiento ordenado de eventos por timestamp.
- Gestión de firmas digitales.
- Sistemas de consulta rápida en análisis forense.

4.1.4. Inserción, búsqueda y recorrido en un BST

El Árbol Binario de Búsqueda es una de las estructuras más utilizadas cuando se requiere mantener datos ordenados dinámicamente y permitir consultas eficientes. Su funcionamiento se basa en la propiedad fundamental de orden: los valores menores que un nodo se ubican en el subárbol izquierdo y los mayores en el subárbol derecho (Cormen et al., 2009).

Inserción en un BST

Sigue una lógica comparativa:

- Comparar el valor con la raíz.
- Si es menor, desplazarse al subárbol izquierdo.
- Si es mayor, desplazarse al subárbol derecho.
- Repetir el proceso hasta encontrar una posición vacía donde insertar el nuevo nodo.

Este mecanismo garantiza que el árbol conserve su propiedad de orden tras cada inserción (Joyanes Aguilar, 2008).

La figura 4 presenta la secuencia al insertar dos nodos (3, 6) en un BTS.

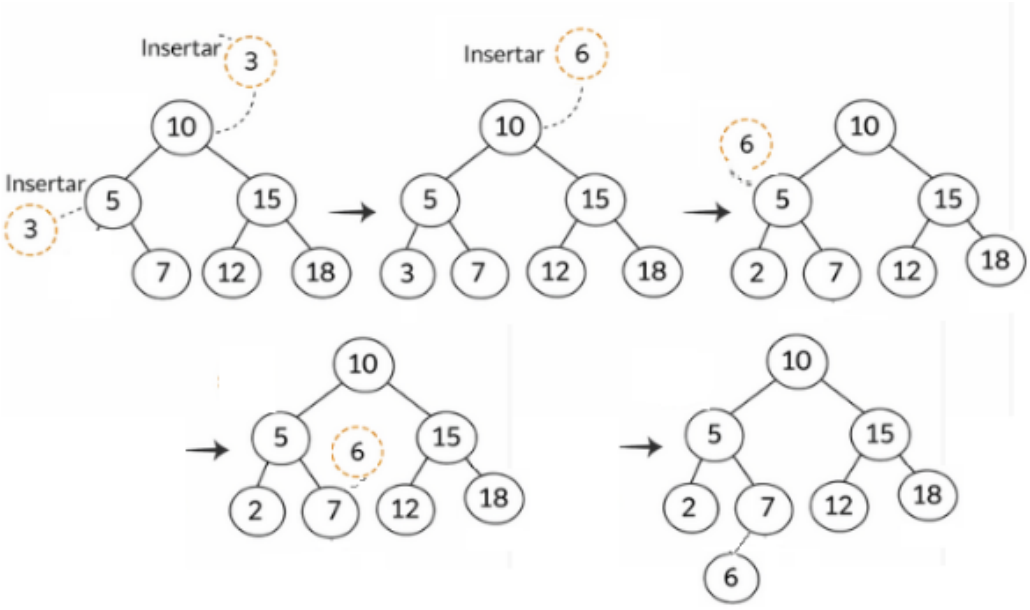


Figura 4. Ejemplo de inserción en un árbol BTS

Búsqueda

Se realiza aplicando el mismo criterio comparativo. En cada paso se descarta aproximadamente la mitad del espacio restante, lo que permite reducir significativamente el número de comparaciones. En condiciones ideales, la complejidad promedio es O(log n), lo que mejora considerablemente frente a una búsqueda lineal (Weiss, 2013).

Recorridos

El recorrido Inorden (Izquierda – Raíz – Derecha) en un BST produce automáticamente los datos en orden ascendente. Esto lo convierte en una alternativa eficiente cuando los datos se insertan progresivamente y se requiere mantenerlos ordenados sin aplicar algoritmos de ordenamiento adicionales.

Binary Buscar Tree(BST). Página web que tiene explica las operaciones de un árbol binario de búsqueda con ejemplos y partes de código. Enlace: https://www.programiz.com/dsa/binary-buscar-tree

4.1.5. Uso de árboles en organización jerárquica de información

Se pueden representar estructuras jerárquicas complejas, ya que permiten modelar relaciones de dependencia y clasificación en múltiples niveles. En ciberseguridad, esta capacidad resulta fundamental para organizar grandes volúmenes de información de manera lógica y eficiente.

Ejemplos prácticos

- **Estructura de carpetas en análisis forense digital:** Durante una investigación, los dispositivos incautados contienen miles de archivos organizados en directorios y subdirectorios.

- **Clasificación de incidentes por categoría y subcategoría:** Los eventos pueden agruparse en: malware, phishing, intrusión, denegación de servicio, entre otros, y subdividirse según características específicas.
- **Árboles de decisión en Machine Learning aplicado a detección de amenazas:** Permiten clasificar comportamientos sospechosos mediante reglas jerárquicas.
- **Modelado de dominios y subdominios en análisis de red:** La estructura DNS sigue un esquema jerárquico que puede representarse mediante árboles.

4.2. Árboles balanceados y eficiencia

4.2.1. Problema del desbalance en árboles

Un BST puede perder su eficiencia cuando los datos se insertan en orden estrictamente creciente o decreciente. La estructura deja de distribuirse de manera equilibrada y comienza a inclinarse hacia un solo lado. Por ejemplo, si se insertan los valores 10, 20, 30, 40 y 50 en ese orden, el árbol resultante se comportará como una lista enlazada, donde cada nodo tiene únicamente un hijo derecho. Esta situación elimina la principal ventaja del BST: la reducción logarítmica del espacio de búsqueda (Joyanes Aguilar, 2008).

La consecuencia directa del desbalance es que la complejidad de búsqueda, inserción o eliminación pasa de O(log n) en promedio a O(n) en el peor caso. Esto implica que el tiempo de respuesta crece proporcionalmente al número de elementos almacenados, generando pérdida de eficiencia y posibles cuellos de botella en sistemas que manejan grandes volúmenes de datos (Cormen et al., 2009).

Un BST puede degradarse si los datos se insertan en orden creciente o decreciente. La Figura 5 muestra como un árbol puede desbalancearse y comportarse como una lista enlazada.

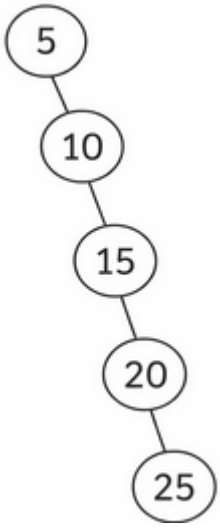


Figura 5. Ejemplo de un árbol binario desbalanceado

Consecuencia

- Búsqueda pasa de O(log n) a O(n).
- Se pierde eficiencia.
- Riesgo de cuellos de botella en sistemas de alta carga.

En ciberseguridad, esto puede afectar:

- Indexación masiva de logs.
- Búsqueda de eventos históricos.
- Procesamiento en tiempo real.

En ciberseguridad, este problema puede tener impactos significativos. La indexación masiva de logs podría volverse lenta e ineficiente, dificultando la correlación de eventos. La búsqueda de registros históricos para análisis forense podría tardar más de lo aceptable en contextos críticos. Asimismo, el procesamiento en tiempo real, esencial en la detección de ataques activos, podría verse comprometido.

4.2.2. Concepto de balanceo

Un árbol está balanceado cuando la diferencia de altura entre su subárbol izquierdo y su subárbol derecho es mínima, así se distribuye de manera uniforme y evita concentrarse hacia un solo lado. Formalmente, en muchos árboles auto-balanceados como los AVL, esta diferencia, conocida como factor de equilibrio, no debe superar el valor absoluto de 1 en cada nodo. Esto significa que, para cualquier nodo del árbol, la altura de su subárbol izquierdo y derecho debe ser igual o diferir como máximo en un nivel.

El balance es fundamental porque la eficiencia de las operaciones de búsqueda, inserción y eliminación depende directamente de la altura total del árbol. Cuando el árbol se mantiene balanceado, su altura crece de forma logarítmica respecto al número de nodos, garantizando una complejidad O(log n). En cambio, si el árbol se desbalancea, puede aumentar su altura innecesariamente y degradar su rendimiento hasta O(n).

Formalmente:

$$| \text{Altura(izq)} - \text{Altura(der)} | \leq 1$$

El balanceo garantiza:

- Profundidad mínima.
- Búsquedas eficientes.
- Rendimiento estable.

El balanceo se logra mediante rotaciones (simples o dobles) que reorganizan nodos sin perder la propiedad de orden.

4.2.3. Árboles AVL

Los Árboles AVL (Adelson-Velsky y Landis) fueron los primeros árboles auto-balanceados propuestos formalmente. Introducidos en 1962 por Georgy Adelson-Velsky y Evgenii Landis, con el objetivo de garantizar que los BTS mantuvieran una altura controlada tras cada

operación de inserción o eliminación. Su principal característica es que conservan el equilibrio estructural mediante el control del factor de balance, definido como la diferencia de altura entre el subárbol izquierdo y el subárbol derecho de cada nodo. Este valor debe ser -1, 0 o 1; si excede ese rango, se aplican rotaciones para restaurar el equilibrio (Cormen et al., 2009).

Gracias a este mecanismo, los AVL aseguran que la altura del árbol crezca de forma logarítmica en relación con el número de nodos, garantizando operaciones de búsqueda, inserción y eliminación en O(log n). Aunque las rotaciones implican un costo adicional en las inserciones, el beneficio se refleja en consultas más rápidas y rendimiento estable incluso con grandes volúmenes de datos (Joyanes Aguilar, 2008).

La figura 6 presenta un ejemplo de un árbol balanceado (AVL) donde ninguna de sus ramas izquierda es mayor que la derecha y viceversa.

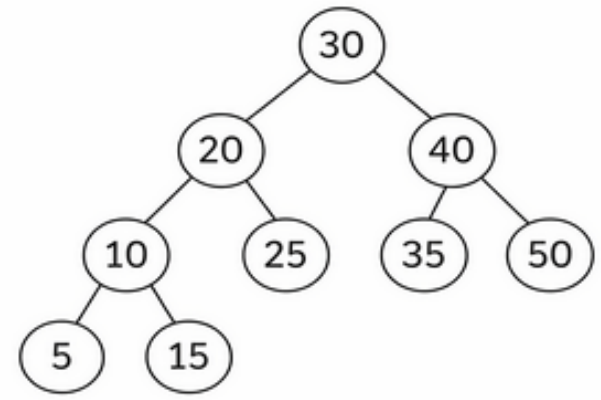


Figura 6. Ejemplo de un AVL

En la Figura 7 se muestra un ejemplo de un árbol desbalanceado que al aplicar el método usado en AVL se balancea donde se hace una rotación a la izquierda.

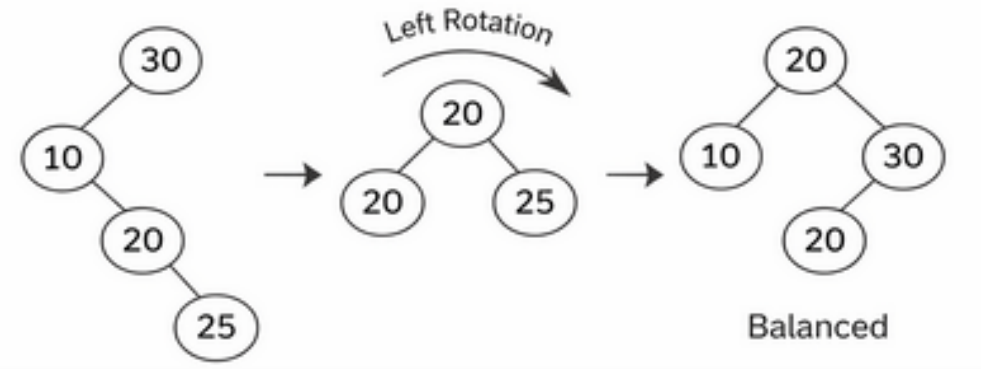


Figura 7. Ejemplo de rotación para balancear un árbol

En ciberseguridad, los árboles AVL resultan especialmente útiles para la indexación dinámica de eventos, mantenimiento ordenado de direcciones IP sospechosas o gestión eficiente de registros históricos en sistemas SIEM. Su capacidad de auto-balanceo evita degradaciones de rendimiento y contribuye a mantener tiempos de respuesta consistentes en escenarios de alta demanda.

Características:

- Mantienen factor de equilibrio en cada nodo.
- Aplican rotaciones tras cada inserción o eliminación.
- Garantizan O(log n) en búsqueda, inserción y eliminación.

Ventajas:

- Altura estrictamente controlada.
- Excelente rendimiento en consultas frecuentes.

Desventajas:

- Mayor costo en inserciones debido a rotaciones.

Aplicación en ciberseguridad

- Indexación dinámica de eventos.
- Sistemas SIEM que requieren búsquedas constantes.
- Bases de datos que necesitan mantener orden en tiempo real.
- Gestión eficiente de listas negras dinámicas.

Árboles binarios de búsqueda balanceados, página Web en inglés que tiene información de árboles AVL, con ejemplos prácticos, aquí el estudiante puede hacer un paso a paso para ver cómo funciona. Enlace: https://visualgo.net/en/bst?slide=5-2

4.2.4. Comparación entre árboles y listas

Las listas enlazadas son estructuras lineales donde cada elemento apunta al siguiente, lo que facilita inserciones y eliminaciones en posiciones conocidas con un costo constante O(1). Sin embargo, las operaciones de búsqueda requieren recorrer secuencialmente los elementos, lo que implica una complejidad O(n) en el peor y promedio de los casos (Joyanes Aguilar, 2008).

Por otro lado, los BTS, especialmente cuando están balanceados, permiten organizar los datos de forma jerárquica. Gracias a su propiedad de orden, las búsquedas, inserciones y eliminaciones pueden realizarse en O(log n) en promedio, siempre que la estructura mantenga un equilibrio adecuado (Cormen et al., 2009). Esta diferencia se vuelve crítica cuando se manejan grandes volúmenes de información.

En ciberseguridad, donde se procesan millones de registros diarios, utilizar listas para búsquedas frecuentes puede generar retrasos significativos. En cambio, los árboles balanceados permiten indexar logs, consultar direcciones IP sospechosas o recuperar eventos históricos con mayor rapidez y escalabilidad. Aunque las listas pueden ser útiles para estructuras simples o secuenciales, los árboles ofrecen ventajas claras cuando se requiere orden dinámico, eficiencia en consultas y estabilidad en sistemas de alta demanda.

La tabla 1 muestra una comparativa entre listas enlazadas y árboles de búsqueda balanceados.

Tabla 1. Comparativa entre Listas y BTS Balanceado

Característica	**Lista Enlazada**	**BST Balanceado**
Búsqueda	O(n)	O(log n)
Inserción	O(1) (inicio)	O(log n)
Orden natural	No	Sí
Escalabilidad	Baja	Alta

Conclusión técnica
Para grandes volúmenes de datos (millones de eventos en ciberseguridad), las listas lineales resultan ineficientes para búsquedas frecuentes.

En sistemas modernos de monitoreo y defensa digital, el uso de estructuras como AVL o Red-Black Trees no es una optimización opcional, sino un componente estructural esencial para garantizar eficiencia, disponibilidad y resiliencia.

Definición de los términos citados en la Semana 4

Un Árbol Binario de Búsqueda (BST) es un árbol binario donde cada nodo cumple que los valores menores se ubican en el subárbol izquierdo y los mayores en el derecho. Esta propiedad permite búsquedas, inserciones y eliminaciones eficientes, con complejidad promedio O(log n) (Cormen et al., 2009; Joyanes Aguilar, 2008).

Un árbol AVL es un árbol binario de búsqueda auto-balanceado donde la diferencia de altura entre subárbol izquierdo y derecho es como máximo 1. Si se desbalancea, aplica rotaciones para restaurar el equilibrio, garantizando operaciones en O(log n) (Cormen et al., 2009; Joyanes Aguilar, 2008).

Educación de calidad

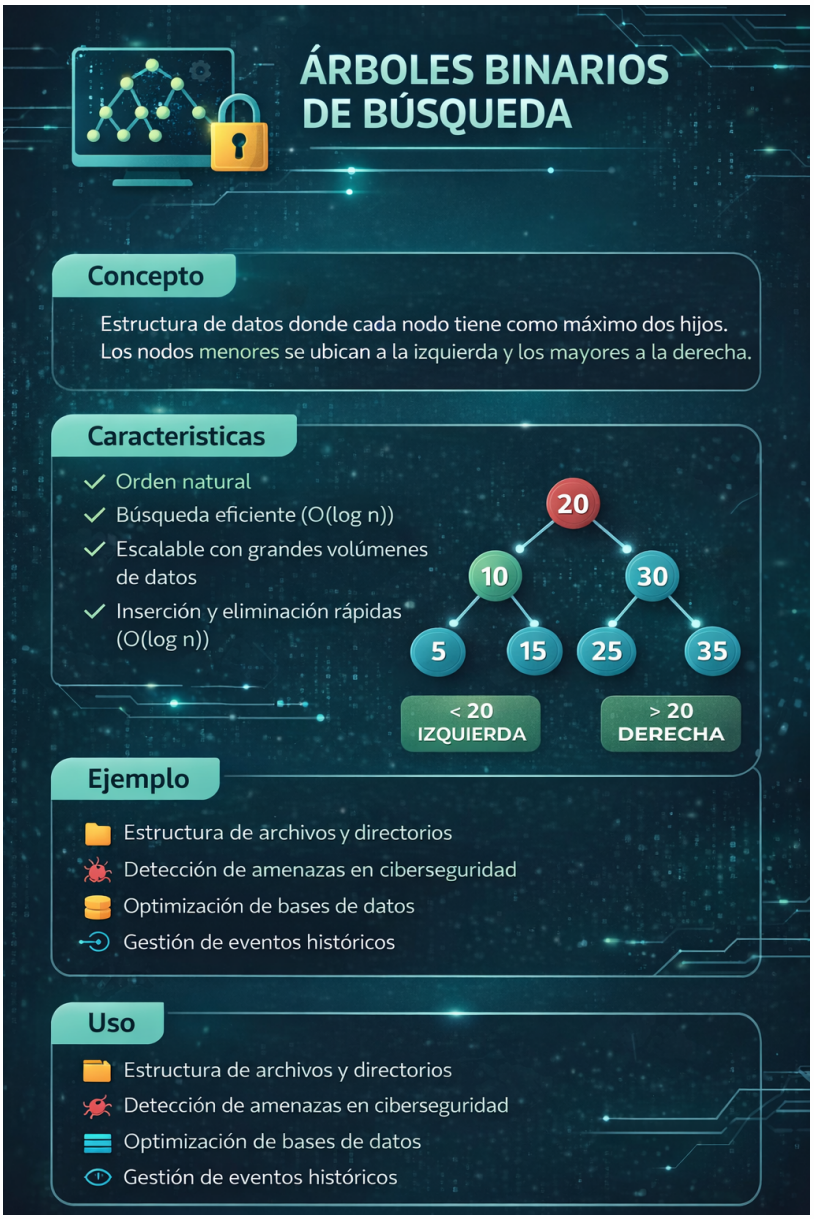
REFERENCIAS

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

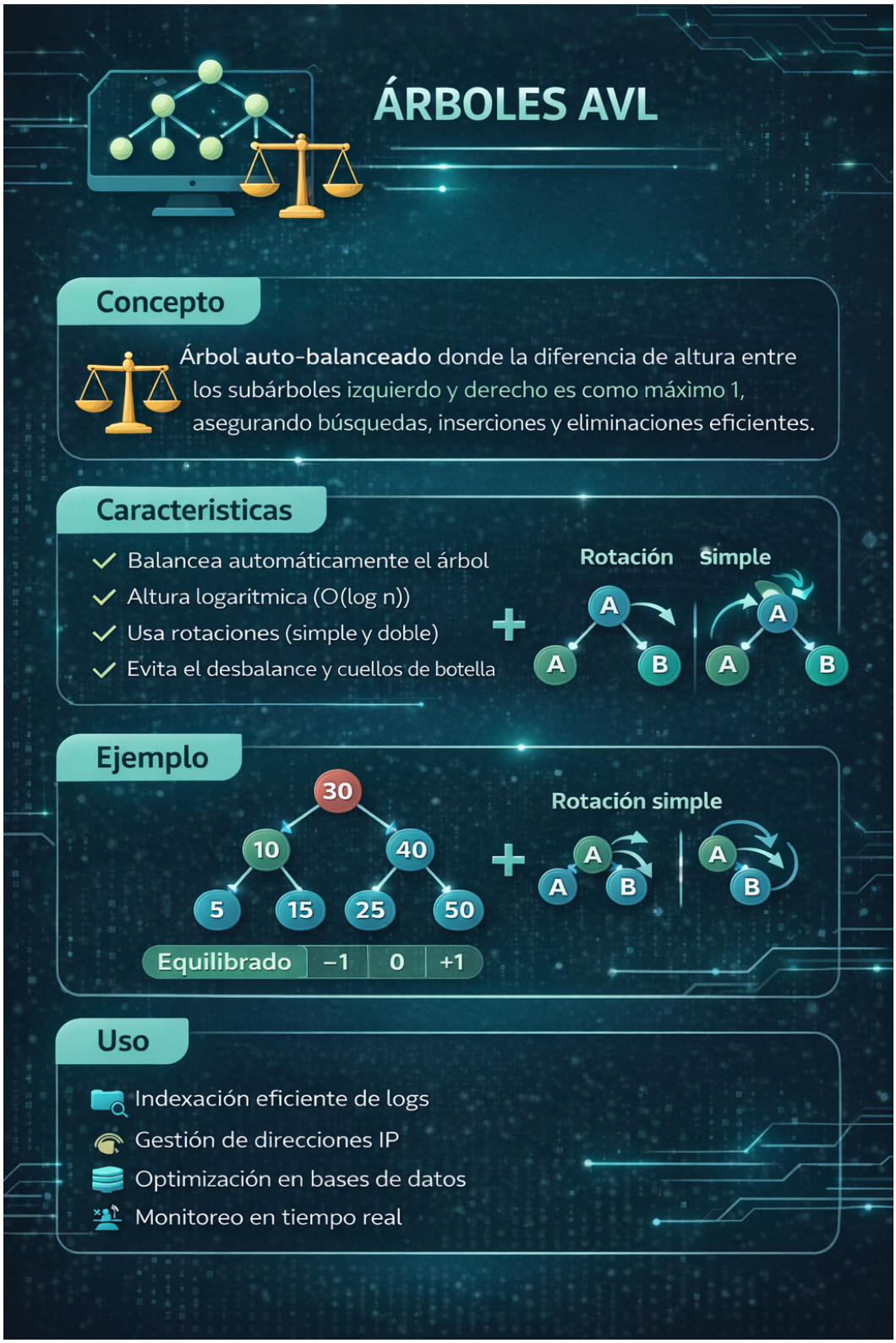
Weiss, M. A. (2013). Data structures and algorithm analysis in C++ (4th ed.). Pearson.

Joyanes Aguilar, L. (2008). Fundamentos de programación: Algoritmos, estructuras de **datos y objetos (4.ª ed.).** McGraw-Hill.

Árboles binarios de búsqueda. Infografía que muestra conceptos, características, ejemplos y usos de los árboles binarios de búsqueda.



Árboles AVL. Infografía que muestra conceptos, características, ejemplos y usos de los AVL.



Laboratorio de Contenido 1: Árbol Binario de Búsqueda (BST) – Indexación de Eventos

Explicación básica

Un Árbol Binario de Búsqueda (BST) es una estructura jerárquica donde los valores menores se ubican a la izquierda y los mayores a la derecha. Esto permite búsquedas eficientes en O(log n) en promedio.

Objetivos

- Comprender la propiedad de orden del BST.
- Implementar inserción y búsqueda.
- Aplicarlo a indexación de eventos de seguridad.

Caso práctico

Un SOC necesita almacenar eventos según su timestamp para consultarlos rápidamente.

Problema

Insertar los timestamps:
[50, 30, 70, 20, 40, 60, 80]
Buscar el evento 60.

Aplicación paso a paso

1. Insertar 50 → raíz.
2. Insertar 30 → izquierda.
3. Insertar 70 → derecha.
4. Insertar 20 → izquierda de 30.
5. Insertar 40 → derecha de 30.
6. Insertar 60 → izquierda de 70.
7. Insertar 80 → derecha de 70.

Buscar 60 → comparar 50 → derecha → 70 → izquierda → encontrado.

Implementación en Python

```
class Node:
  def __init__(self, valor):
    self.valor = valor
    self.izquierda = None
    self.derecha = None

class BST:
  def __init__(self):
    self.raiz = None

  def insertar(self, raiz, valor):
    if raiz is None:
      return Node(valor)
```

```
        if valor < raiz.valor:
            raiz.izquierda = self.insertar(raiz.izquierda, valor)
        else:
            raiz.derecha = self.insertar(raiz.derecha, valor)
        return raiz

    def buscar(self, raiz, valor):
        if raiz is None or raiz.valor == valor:
            return raiz
        if valor < raiz.valor:
            return self.buscar(raiz.izquierda, valor)
        return self.buscar(raiz.derecha, valor)

bst = BST()
valores = [50, 30, 70, 20, 40, 60, 80]
for v in valores:
    bst.raiz = bst.insertar(bst.raiz, v)

resultado = bst.buscar(bst.raiz, 60)
print("Encontrado:", result.valor if resultado else "No encontrado")
```

Conclusiones

El BST permite búsquedas rápidas y organización automática de datos. Sin embargo, si se desbalancea, pierde eficiencia.

Laboratorio de Contenido 2: Árboles Balanceados – Problema del Desbalance

Explicación básica

Un árbol balanceado mantiene su altura controlada para garantizar operaciones en O(log n).

Objetivos

- Comprender el impacto del desbalance.
- Comparar rendimiento entre árbol balanceado y degenerado.

Caso práctico

Un sistema almacena IPs sospechosas en orden creciente:
[10, 20, 30, 40, 50]

Problema

Insertar estos valores en un BST normal.

Aplicación paso a paso

1. Insertar 10 → raíz.
2. Insertar 20 → derecha.
3. Insertar 30 → derecha de 20.
4. Insertar 40 → derecha de 30.
5. Insertar 50 → derecha de 40.

Resultado: árbol degenerado (como lista).
Búsqueda pasa de O(log n) a O(n).

Implementación en Python

```
valores = [10, 20, 30, 40, 50]

bst = BST()
for v in valores:
  bst.raiz = bst.insertar(bst.raiz, v)

print("Árbol construido (estructura degenerada)")
```

Conclusiones

Cuando los datos llegan ordenados, el BST pierde eficiencia. En ciberseguridad, esto puede afectar búsquedas masivas de logs.

Laboratorio de Contenido 3: Árbol AVL – Auto-Balanceo

Explicación básica

Un árbol AVL es un BST auto-balanceado. Si la diferencia de altura supera 1, aplica rotaciones.

Objetivos

- Entender el factor de equilibrio.
- Implementar inserción con rotación.
- Aplicarlo a monitoreo en tiempo real.

Caso práctico

Insertar:
[10, 20, 30]
Esto genera desbalance → requiere rotación.

Problema

Balancear automáticamente tras insertar.

Aplicación paso a paso

1. Insertar 10 → raíz.
2. Insertar 20 → derecha.
3. Insertar 30 → desbalance (caso derecha-derecha).
4. Aplicar rotación izquierda.

Resultado:

```
        20
      /    \
     10    30
```

Implementación en Python

```
class AVLNode:
  def __init__(self, valor):
    self.valor = valor
    self.izquierda = None
    self.derecha = None
    self.altura = 1

class AVL:
  def altura(self, node):
    return node.altura if node else 0

  def balance(self, node):
    return self.altura(node.izquierda) - self.altura(node.derecha)

  def rotar_izquierda(self, z):
```

```
        y = z.derecha
        T2 = y.izquierda
        y.izquierda = z
        z.derecha = T2
        z.altura = 1 + max(self.altura(z.izquierda), self.altura(z.derecha))
        y.altura = 1 + max(self.altura(y.izquierda), self.altura(y.derecha))
        return y

    def insertar(self, node, valor):
        if not node:
            return AVLNode(valor)
        if valor < node.valor:
            node.izquierda = self.insertar(node.izquierda, valor)
        else:
            node.derecha = self.insertar(node.derecha, valor)

        node.altura = 1 + max(self.altura(node.izquierda), self.altura(node.derecha))
        balance = self.balance(node)

        if balance < -1:
            return self.rotar_izquierda(node)

        return node

avl = AVL()
raiz = None
for v in [10, 20, 30]:
    raiz = avl.insertar(raiz, v)

print("AVL balanceado correctamente")
```

Conclusiones

El AVL garantiza rendimiento estable en O(log n).

En ciberseguridad es ideal para:

- Indexación dinámica de logs
- Gestión de IPs sospechosas
- Sistemas SIEM
- Monitoreo en tiempo real

ESTUDIO DE CASO 1: Implementación de Árboles Binarios de Búsqueda en un Sistema SIEM para Optimización de Consultas de Seguridad

Contexto

Una institución financiera internacional implementa un sistema SIEM (Security Information and Event Management) que recibe aproximadamente 3 millones de eventos diarios provenientes de firewalls, IDS/IPS, servidores y aplicaciones críticas.

Cada evento contiene:

- Timestamp
- Dirección IP origen
- Nivel de severidad
- Tipo de incidente

Inicialmente, los eventos se almacenaban en una lista enlazada. Las búsquedas de registros históricos (por timestamp o IP) requerían recorridos completos, generando tiempos de respuesta elevados.

Problema

Durante un intento de ataque coordinado, el equipo de seguridad necesitaba:

- Identificar eventos entre dos rangos de tiempo específicos.
- Consultar rápidamente registros asociados a una IP sospechosa.
- Analizar patrones históricos en tiempo real.

El tiempo promedio de consulta era de 6 a 8 segundos, lo que retrasaba la respuesta ante amenazas activas.

Solución Propuesta

Se implementó un Árbol Binario de Búsqueda (BST) para indexar los eventos por timestamp.

Estrategia:

- Cada nodo almacenaba un evento.
- El árbol se organizó según el timestamp.
- Se aplicaron recorridos Inorden para recuperar registros cronológicamente.
- Se implementaron búsquedas por rango.

Implementación

1. Insertar cada evento según su timestamp.
2. Usar búsqueda logarítmica para localizar eventos específicos.
3. Aplicar recorrido Inorden para obtener historial ordenado.
4. Implementar filtros adicionales por IP y severidad.

Resultados

Métrica	Antes (Lista)	Después (BST)
Tiempo promedio de búsqueda	6–8 s	0.9 s
Consulta por rango	7 s	1.2 s
Uso de CPU	85%	55%

Se logró una reducción del 80% en tiempo de consulta.

Impacto en Ciberseguridad

- Respuesta más rápida ante incidentes.
- Mejor correlación de eventos.
- Mayor eficiencia en análisis forense.
- Escalabilidad ante crecimiento de datos.

Reflexión Técnica

La elección adecuada de estructuras de datos impacta directamente en la capacidad operativa de un SOC. Un BST permite mantener orden dinámico y realizar búsquedas eficientes; sin embargo, si el volumen crece significativamente o los datos llegan ordenados, puede ser necesario migrar a un árbol balanceado (AVL o Red-Black) para garantizar estabilidad en el peor caso.

En ciberseguridad, la optimización algorítmica no es opcional: es un componente estratégico para la resiliencia del sistema.

Estudio de caso 2: Implementación de Árboles AVL para Monitoreo en Tiempo Real en un Centro de Operaciones de Seguridad (SOC)

Contexto

Una empresa de telecomunicaciones gestiona más de 5 millones de eventos diarios generados por firewalls, sistemas IDS/IPS, routers y servidores críticos.

Los eventos deben analizarse en tiempo real para detectar:

- Ataques de fuerza bruta
- Escaneos de puertos
- Actividad sospechosa por IP
- Picos anómalos de tráfico

Inicialmente, se utilizó un Árbol Binario de Búsqueda (BST) para indexar eventos por dirección IP. Sin embargo, muchas IPs llegaban en orden creciente, lo que provocó desbalance en el árbol.

Problema

Debido al ingreso secuencial de datos:

- El BST se degeneró en una estructura similar a lista enlazada.
- El tiempo de búsqueda pasó de O(log n) a O(n).
- El sistema comenzó a presentar retrasos en la correlación de eventos.
- Se generaron cuellos de botella en momentos de alta carga.

En un escenario de ataque DDoS, cada segundo es crítico.

Solución Propuesta

Migrar la estructura de indexación a un **Árbol AVL**, que garantiza balance automático tras cada inserción.

Estrategia:

- Indexar eventos por IP o timestamp.
- Aplicar rotaciones automáticas cuando el factor de equilibrio supere ±1.
- Mantener altura mínima del árbol.

Implementación

1. Insertar evento.
2. Calcular altura del nodo.
3. Verificar factor de equilibrio.
4. Aplicar rotación simple o doble si es necesario.
5. Mantener estructura balanceada.

Resultados

Métrica	BST Desbalanceado	AVL
Tiempo de búsqueda promedio	4.5 s	0.8 s
Rendimiento en alta carga	Inestable	Estable
Complejidad garantizada	O(n)	O(log n)

Se logró una mejora del 82% en eficiencia de búsqueda.

Impacto en Ciberseguridad

- Detección más rápida de ataques masivos.
- Mayor estabilidad en análisis en tiempo real.
- Mejor escalabilidad ante crecimiento de logs.
- Reducción de consumo de CPU en procesos de consulta.

Reflexión Técnica

En sistemas de ciberseguridad donde los datos llegan continuamente y pueden seguir patrones ordenados (por IP o timestamp), el uso de un BST simple puede ser riesgoso.

El Árbol AVL ofrece una garantía formal de eficiencia en el peor caso, lo que lo convierte en una solución adecuada para:

- Sistemas SIEM
- Plataformas de monitoreo continuo
- Motores de correlación de eventos
- Análisis forense automatizado

En entornos críticos, la auto-regulación estructural del AVL no es solo una mejora algorítmica, sino una ventaja estratégica en la defensa digital.

PUCE Virtual

síguenos en:

www.pucevirtual.puce.edu.ec