Análisis y Evaluación de la Estructura de Datos *Treap*

Fabio Eduardo Dávila Venturo
Facultad de Computación
Universidad de Ingeniería y Tecnología (UTEC)
Lima, Perú

Cesar Stefano Flores Uriarte
Facultad de Computación
Universidad de Ingeniería y Tecnología (UTEC)
Lima, Perú

Llorent Eloy Nunayalle Brañes
Facultad de Computación
Universidad de Ingeniería y Tecnología (UTEC)
Lima, Perú

Jordinn Martín Reyes Medina
Facultad de Computación
Universidad de Ingeniería y Tecnología (UTEC)
Lima, Perú

***Resumen*—El *Treap* es una estructura de datos que combina las propiedades de un árbol binario de búsqueda y un *heap*, utilizando prioridades aleatorias para mantener su equilibrio. Este trabajo analiza su estructura, operaciones principales y complejidad algorítmica, comparándolo con otras estructuras balanceadas como AVL, Red-Black y B-Tree. Se concluye que el Treap ofrece una implementación sencilla y un rendimiento eficiente en promedio, con operaciones de costo esperado $O(\log n)$.**

***Index Terms*—Treap, árbol binario de búsqueda, heap, estructura de datos, rotaciones, balanceo probabilístico.**

I. Introduction

En el ámbito de las estructuras de datos, los árboles binarios de búsqueda (BST) son fundamentales para mantener conjuntos de datos ordenados y permitir operaciones eficientes de búsqueda, inserción y eliminación. Sin embargo, su eficiencia depende de su equilibrio: si las claves se insertan en un orden desfavorable, el árbol puede convertirse en una lista enlazada, haciendo que las operaciones pasen de ser logarítmicas a lineales.

Para resolver este problema, se desarrollaron los árboles binarios de búsqueda auto-balanceables, como los árboles AVL y los árboles Rojo-Negro, que imponen condiciones estrictas para garantizar el equilibrio. No obstante, estas estructuras suelen tener implementaciones complejas.

El *Treap* es una estructura que combina la simplicidad de un BST con la capacidad de un *heap* de mantenerse balanceado. Su nombre es un acrónimo de *tree* + *heap*. La idea principal es asignar a cada nodo una clave (que sigue el orden del BST) y una prioridad aleatoria (que sigue el orden de un *max-heap*). Esta combinación tiene una alta probabilidad de que el árbol mantenga una altura logarítmica en el número de elementos, haciendo que las operaciones tengan un costo computacional esperado de $O(\log n)$.

Esta estructura fue descrita por primera vez por Cecilia R. Aragon y Raimund Seidel [1] en 1989 y se ha convertido en una herramienta valiosa en programación debido a su facilidad de implementación y robustez.

II. Definición y Fundamentos Teóricos

Para comprender la estructura del Treap, debemos definir los conceptos básicos que lo componen:

II-A. Árbol Binario de Búsqueda (BST)

Un Árbol Binario de Búsqueda es una estructura de datos en la que cada nodo tiene como máximo dos hijos (izquierdo y derecho) y cumple con la siguiente propiedad:

> Para cualquier nodo, todas las claves en el subárbol izquierdo son menores que la clave del nodo, y todas las claves en el subárbol derecho son mayores. [2]

II-B. Heap

Un Heap es un árbol binario completo que satisface la propiedad de orden del heap:

- En un *max-heap*, la prioridad de cada nodo es mayor o igual que la de sus hijos.
- En un *min-heap*, la prioridad de cada nodo es menor o igual que la de sus hijos. [3]

II-C. Combinación de BST y Heap

El Treap combina estas dos estructuras:

- **Clave:** determina la posición según el orden BST.
- **Prioridad:** valor aleatorio que determina la posición como un *max-heap* (o *min-heap*) [1].

II-D. Balanceo Probabilístico

La clave de los Treaps está en que las prioridades se asignan aleatoriamente a cada nodo durante la inserción. Esto genera una alta probabilidad de que la altura del árbol sea $O(\log n)$, donde n es el número de elementos. [4]

No obstante, esta propiedad depende directamente de la calidad del generador de números aleatorios utilizado. Un generador deficiente, con patrones sesgados o periodos cortos, puede afectar negativamente la distribución de prioridades, aumentando la probabilidad de que el árbol adopte configuraciones desbalanceadas. Por ello, en implementaciones prácticas se recomienda emplear generadores robustos, como los disponibles

en <random> de C++ (por ejemplo, std::mt19937), que garantizan distribuciones uniformes con buena independencia estadística. El correcto manejo de la aleatoriedad es esencial para preservar las garantías teóricas del Treap y mitigar la posibilidad de caer en casos degenerados.

III. Estructura

Cada nodo del Treap contiene cuatro componentes: una clave que determina su posición según el orden del BST, una prioridad aleatoria que determina su posición según el orden del heap, y dos referencias a sus subárboles izquierdo y derecho. La prioridad es generada de forma aleatoria durante la inserción y asegura que la estructura se mantenga balanceada, lo que previene que el árbol se vuelva una estructura lineal. [2]

Algorithm 1 Estructura de un nodo en un Treap

```
Tipo NodoTreap:
    key: tipo ordenable
    priority: entero
    left: referencia a NodoTreap
    right: referencia a NodoTreap
```

Cuando un hijo tiene una prioridad mayor que su padre, se realizan rotaciones a la izquierda o a la derecha para mantener el orden según la propiedad de BST. Por ejemplo, si el hijo izquierdo de un nodo tiene mayor prioridad, se hace una rotación derecha para que se vuelva el nuevo padre. [3]

A diferencia de los árboles AVL o Rojo-Negro que requieren operaciones de balanceo y guardar información adicional, el Treap logra equilibrarse mediante prioridades aleatorias. Esta característica no solo simplifica la implementación, sino que además garantiza una altura esperada logarítmcia. [1] La complejidad espacial se mantiene lineal según el número de elementos, ya que cada nodo solo almacena su clave, prioridad y dos punteros, sin necesidad de datos adicionales.

IV. Operaciones Fundamentales

IV-A. A. Búsqueda

La operación de búsqueda se comporta exactamente igual que en un árbol binario de búsqueda. No se consideran las prioridades, ya que estas no afectan el recorrido.

Fases del proceso:

- Se inicia el recorrido desde la raíz del árbol.
- Si la clave buscada es menor que la del nodo actual, se avanza hacia el subárbol izquierdo.
- Si la clave es mayor, se continúa por el subárbol derecho.
- El proceso finaliza al encontrar el nodo o llegar a un puntero nulo.

El pseudocódigo del procedimiento se presenta en el Algoritmo 2, mientras que un ejemplo visual del recorrido puede verse en la Figura 1.

Esta operación no modifica la estructura del árbol y presenta un tiempo promedio de ejecución $O(\log n)$, resultado del equilibrio aleatorio que mantiene el Treap.

Algorithm 2 Búsqueda en un Treap

```
procedure SEARCH(T : NodoTreap, k : tipo ordenable)
    if T = null then
        return null ▷ el árbol está vacío o no contiene la clave
    end if
    if k = T.key then
        return T ▷ se encontró el nodo solicitado
    else if k < T.key then
        return SEARCH(T.left, k)
    else
        return SEARCH(T.right, k)
    end if
end procedure
```

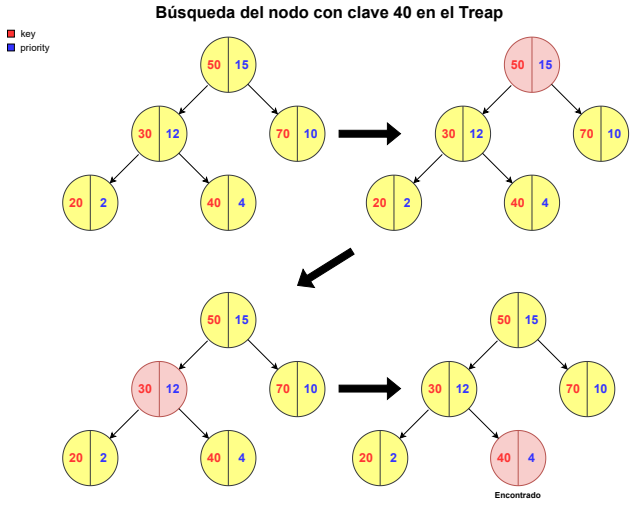

Figura 1. Búsqueda de una clave en el Treap.

IV-B. B. Inserción

La inserción de un nodo en el Treap consiste en ubicarlo según su clave y, posteriormente, ajustar la estructura para conservar la propiedad del heap.

Fases del proceso:

- Se determina la posición correcta del nuevo nodo a partir de la comparación de claves, siguiendo las reglas de un BST.
- Se asigna al nodo una prioridad aleatoria y se inserta como hoja.
- Si la prioridad del nuevo nodo supera la de su padre, se realizan rotaciones ascendentes hasta que se restablezca el orden del heap.
- El procedimiento concluye cuando el nodo alcanza una posición válida dentro de la jerarquía.

El procedimiento detallado se presenta en el Algoritmo 3, mientras que el comportamiento visual del reajuste de prioridades puede observarse en la Figura 2.

Este mecanismo permite que el árbol conserve sus propiedades de orden y equilibrio sin depender de reglas determinísti-

Algorithm 3 Inserción en un Treap

```
1:  procedure INSERT(T : referencia a NodoTreap, k : tipo ordenable, p : entero)
2:      if T = null then
3:          T ← NEWNODE
4:          T.key ← k
5:          T.priority ← p
6:          T.left ← null
7:          T.right ← null
8:          return                    ▷ nodo creado; fin de la inserción
9:      end if
10:     if k < T.key then
11:         INSERT(T.left, k, p)
12:         if T.left.priority > T.priority then
13:             ROTATERIGHT(T)
14:         end if
15:     else if k > T.key then
16:         INSERT(T.right, k, p)
17:         if T.right.priority > T.priority then
18:             ROTATELEFT(T)
19:         end if
20:     else                          ▷ la clave ya existe en el Treap
21:     end if
22: end procedure
```

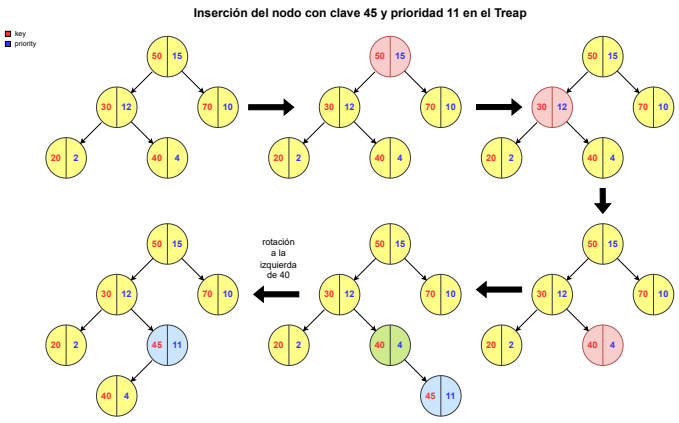

Figura 2. Inserción de un nodo y reajuste por prioridad.

cas de balanceo.

IV-C. C. Eliminación

La eliminación de un nodo se realiza desplazándolo progresivamente hacia una hoja mediante rotaciones, hasta que pueda ser removido sin alterar la estructura general del árbol.

Fases del proceso:

- Se localiza el nodo que contiene la clave a eliminar.
- Mientras el nodo tenga hijos, se selecciona aquel con mayor prioridad y se ejecuta una rotación para mover el nodo hacia niveles inferiores.
- Cuando el nodo se convierte en una hoja, se elimina de la estructura.

Algorithm 4 Eliminación de un nodo en un Treap

```
1:  procedure DELETE(T : referencia a NodoTreap, k : tipo ordenable)
2:      if T = null then
3:          return                    ▷ la clave no existe en el Treap
4:      end if
5:      if k < T.key then
6:          DELETE(T.left, k)
7:      else if T.key < k then
8:          DELETE(T.right, k)
9:      else
10:         DELETEROOT(T)             ▷ se encontró la clave
11:     end if
12: end procedure
```

Algorithm 5 Eliminación de la raíz de un subárbol en un Treap

```
1:  procedure DELETEROOT(T : referencia a NodoTreap)
2:      if T.left = null  and  T.right = null then
3:          T ← null                  ▷ el nodo es una hoja
4:          return
5:      end if
6:      if T.left ≠ null  and  T.right = null then
7:          ROTATERIGHT(T)
8:          DELETEROOT(T.right)
9:          return
10:     end if
11:     if T.left = null  and  T.right ≠ null then
12:         ROTATELEFT(T)
13:         DELETEROOT(T.left)
14:         return
15:     end if
16:     if T.left.priority > T.right.priority then
17:         ROTATERIGHT(T)
18:         DELETEROOT(T.right)
19:     else
20:         ROTATELEFT(T)
21:         DELETEROOT(T.left)
22:     end if
23: end procedure
```

El procedimiento lógico de búsqueda y reestructuración se detalla en los Algoritmos 4 y 5, mientras que la Figura 3 muestra visualmente el proceso de rotaciones descendentes que permiten mantener las propiedades del Treap tras cada eliminación.

De este modo, el Treap mantiene la propiedad de heap y el orden incluso después de múltiples eliminaciones.

IV-D. D. División (Split)

La operación de división, o *split*, separa un Treap en dos subárboles a partir de una clave de referencia a. El resultado son dos Treaps independientes: uno con las claves menores que a y otro con las mayores.

Fases del proceso:

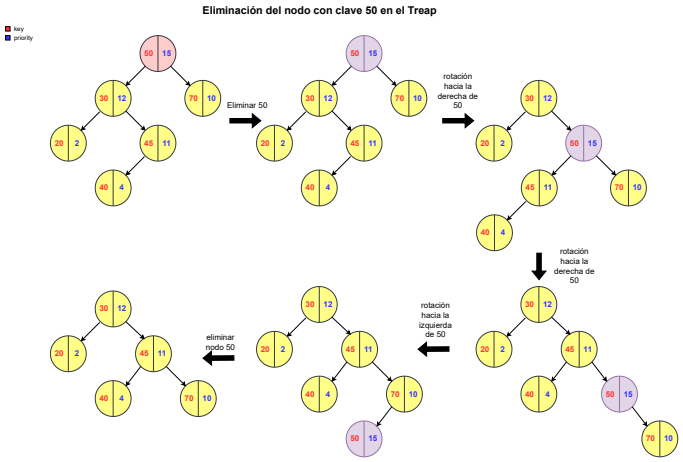

Figura 3. Eliminación de un nodo mediante rotaciones descendentes.

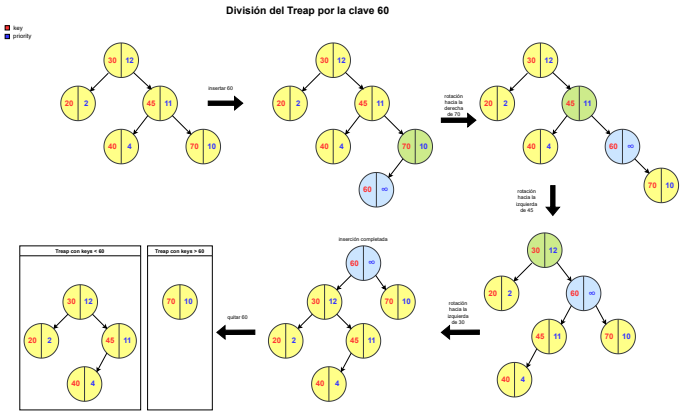

Figura 4. División del Treap mediante la operación *split*.

- Se inserta un nodo centinela con clave a y una prioridad mayor que todas las existentes.
- Debido a su alta prioridad, el nodo asciende hasta la raíz del árbol.
- Los subárboles izquierdo y derecho representan, respectivamente, los conjuntos de claves menores y mayores a a.
- Finalmente, se quita el nodo centinela, quedando dos Treaps válidos y separados.

La Figura 4 ilustra el proceso de separación de los subárboles resultantes.

Esta operación se utiliza frecuentemente en algoritmos que requieren manipular rangos o combinar resultados parciales de manera independiente.

IV-E. E. Unión (Merge)

La unión, o *merge*, realiza el proceso inverso al *split*, combinando dos Treaps disjuntos en una única estructura balanceada. Para ello, es necesario que todas las claves del primer árbol sean menores que las del segundo.

Fases del proceso:

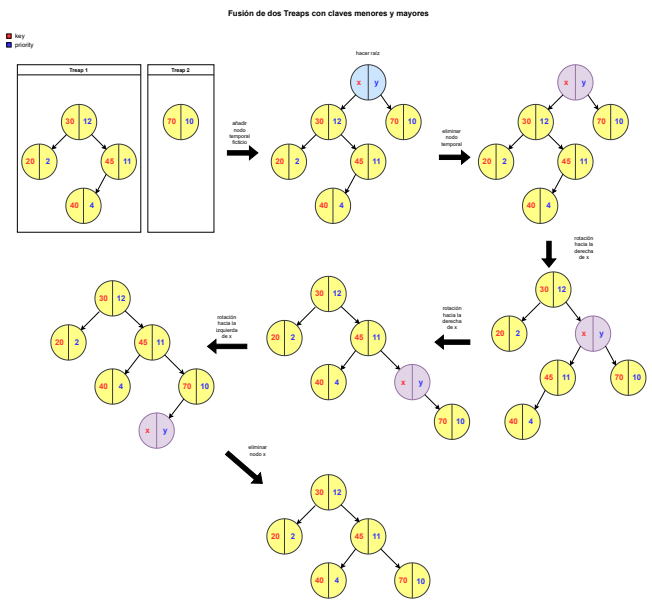

Figura 5. Unión de dos Treaps utilizando un nodo temporal.

- Se crea un nodo temporal de prioridad máxima que actúa como raíz provisional.
- El primer Treap se asigna al subárbol izquierdo y el segundo al derecho.
- Se elimina el nodo temporal mediante el mismo procedimiento de rotaciones utilizado en la eliminación común.
- El resultado es un único Treap equilibrado que preserva las propiedades de ambas estructuras originales.

Un ejemplo visual puede observarse en la Figura 5.

La operación *merge* es clave en algoritmos que requieren construir árboles compuestos de forma dinámica sin perder eficiencia.

V. Análisis y complejidad algorítmica

El Treap se presenta como una estructura de datos híbrida que combina lo mejor de dos mundos: la organización jerárquica y eficiente de un árbol binario de búsqueda, junto con la capacidad de mantener un equilibrio dinámico mediante prioridades aleatorias, propias de un heap. Gracias a esta fusión, el Treap logra un rendimiento estable en la mayoría de sus operaciones. En las secciones siguientes se estudia en detalle la complejidad de dichas operaciones, analizando tanto su comportamiento promedio como las situaciones en las que puede alcanzar sus límites teóricos.

V-A. Inserción

Cómo se mencionó en la sección anterior, el proceso de inserción en el Treap consta de dos fases. En la primera, el nuevo nodo se inserta siguiendo las reglas de un BST, En la segunda, se realiza un reajuste estructural mediante rotaciones según las propiedades del heap basado en prioridades aleatorias.

1. **Fase de inserción:** Se localiza la posición para el nuevo nodo, al igual que un BST. Este procedimiento requiere recorrer una rama del árbol, cuya longitud esperada es proporcional a la altura promedio del Treap. Por tanto, su complejidad temporal esperada es de $O(\log n)$.
2. **Fase de rotación:** puede ser necesario aplicar una serie de rotaciones hacia arriba para mantener la propiedad del heap. Cada rotación tiene un costo constante, y en el peor de los casos se realizan tantas rotaciones como la altura del árbol, lo que también da una complejidad esperada de $O(\log n)$.
   Dado que ambas fases tienen complejidad esperada $O(\log n)$, la complejdad total de la operación en promedio es $O(\log n)$. En el peor de los casos, cuando el Treap pierde su equilibrio (por ejemplo, si las prioridades aleatorias generan un árbol degenerado), la complejidad puede degradarse hasta $O(n)$.

V-B. Eliminación

La operación de eliminación en un Treap también se realiza en dos fases, combinando las propiedades del árbol binario de búsqueda (BST) y del heap. En primer lugar, se localiza el nodo que se desea eliminar, y luego se aplican rotaciones para mantener el equilibrio probabilístico del Treap.

1. **Fase de búsqueda:** Se recorre el Treap siguiendo las reglas del BST hasta encontrar el nodo a eliminar. Este proceso tiene una complejidad esperada de $O(\log n)$, ya que la altura promedio del árbol es logarítmica.
2. **Fase de reestructuración:** Una vez localizado el nodo, se aplica una secuencia de rotaciones hacia abajo (rotando con su hijo de mayor prioridad) hasta que el nodo a eliminar se convierte en una hoja. Finalmente, se elimina de manera directa. Dado que cada rotación tiene un costo constante y su número está acotado por la altura del árbol, esta fase también tiene una complejidad esperada de $O(\log n)$.

Por lo tanto, la complejidad total promedio de la eliminación es $O(\log n)$. En el peor de los casos, si el Treap se encuentra desbalanceado (por ejemplo, si las prioridades generan una estructura degenerada), el tiempo de ejecución puede alcanzar $O(n)$.

V-C. Búsqueda

La búsqueda en un Treap se realiza siguiendo la misma lógica que en un árbol binario de búsqueda tradicional. Dado que las prioridades no afectan el orden de las claves, la estructura mantiene la propiedad de búsqueda binaria.

1. **Procedimiento:** Se compara la clave buscada con la del nodo actual y se avanza hacia la izquierda o derecha dependiendo del resultado, hasta encontrar el elemento o determinar que no existe en el árbol.

La complejidad esperada de esta operación es $O(\log n)$, dado que la altura promedio de un Treap balanceado aleatoriamente es logarítmica en el número de nodos. En el peor de los casos, cuando el árbol se vuelve lineal, la complejidad puede ser $O(n)$.

V-D. Otros: acceso por índice y obtención del elemento superior

Además de las operaciones básicas, el Treap puede extenderse para soportar operaciones adicionales de manera eficiente, especialmente cuando se mantiene información auxiliar en cada nodo, como el tamaño del subárbol.

1. **Acceso por índice:** Si cada nodo almacena el tamaño de su subárbol, es posible acceder al k-ésimo elemento (según orden en el BST) en tiempo esperado $O(\log n)$, recorriendo el árbol y decidiendo en cada paso si avanzar hacia la izquierda o derecha.
2. **Obtención del elemento superior (top):** En un Treap donde la prioridad representa la "importancia" o "peso", el elemento superior (mayor prioridad) se encuentra en la raíz. Su acceso o lectura tiene una complejidad $O(1)$. Si se requiere eliminarlo, la complejidad es $O(\log n)$, al igual que en la operación de eliminación.

Estas extensiones hacen del Treap una estructura flexible, capaz de adaptarse a distintas necesidades como estructuras de colas de prioridad ordenadas o índices dinámicos.

Operación	Caso promedio	Peor caso
Inserción	$O(\log n)$	$O(n)$
Eliminación	$O(\log n)$	$O(n)$
Búsqueda	$O(\log n)$	$O(n)$
Acceso por índice	$O(\log n)$	$O(n)$
Obtener el top	$O(1)$	$O(1)$

Cuadro I
RESUMEN DE COMPLEJIDAD ALGORÍTMICA DEL TREAP.

Complejidad espacial:
El Treap requiere un espacio total de $O(n)$, ya que cada elemento se almacena en un nodo independiente con punteros a sus hijos y, opcionalmente, información auxiliar como la prioridad o el tamaño del subárbol. Esta complejidad es comparable a la de otras estructuras de árboles binarios balanceados, como los AVL o los Red-Black Trees.

VI. COMPARACIÓN FRENTE A OTRAS ESTRUCTURAS DE DATOS

En esta sección se comparará el Treap con varias estructuras de datos relacionadas, destacando sus fortalezas y debilidades en comparación con cada una. La comparación se centra en los enfoques compartidos, como eficiencia en búsqueda, balanceo y manejo de prioridades.

VI-A. Comparación con AVL

El AVL es un árbol binario de búsqueda balanceado que garantiza búsqueda y actualización en $O(\log n)$ de manera determinística.

Ventajas del Treap frente al AVL:

- Implementación más sencilla, sin necesidad de mantener factores de balance estrictos.
- Balance probabilístico que funciona bien en promedio para inserciones y eliminaciones.
- Posibilidad de extenderse fácilmente para operaciones adicionales como acceso por índice o prioridad.

Desventajas del Treap frente al AVL:

- No garantiza balanceo determinista; en el peor caso, la altura puede ser $O(n)$.
- Rendimiento en casos límite puede ser inferior al AVL.

VI-B. Comparación con B-Tree y B+Tree

Los B-Trees y B+Trees son estructuras diseñadas para almacenamiento en disco y bases de datos, optimizando accesos secuenciales y bloques de datos.

Ventajas del Treap frente a B-Tree / B+Tree:

- Simplicidad de implementación en memoria principal.
- Eficiencia comparable en búsquedas y actualizaciones en entornos donde no se requiere optimización de bloques de disco.

Desventajas del Treap frente a B-Tree / B+Tree:

- No está optimizado para operaciones en disco; acceso secuencial y paginado es menos eficiente.
- No permite almacenar múltiples claves en un nodo, limitando la eficiencia en bases de datos grandes.

VI-C. Comparación con Red-Black Tree

Los Red-Black Trees son árboles binarios de búsqueda balanceados que garantizan altura $O(\log n)$ de manera determinista y se usan ampliamente en bibliotecas estándar.

Ventajas del Treap frente al Red-Black Tree:

- Implementación más simple, sin reglas de color y rotaciones condicionadas.
- Permite extender operaciones fácilmente (por ejemplo, acceso por índice o top-priority).

Desventajas del Treap frente al Red-Black Tree:

- Balance probabilístico; peor caso $O(n)$ es posible.
- Menor predictibilidad en entornos donde la eficiencia determinista es crítica.

VI-D. Comparación con Binary Heap

El Binary Heap es una estructura de heap clásica utilizada para implementar colas de prioridad, optimizando inserciones y extracción del máximo o mínimo.

Ventajas del Treap frente al Binary Heap:

- Mantiene orden de claves, permitiendo búsquedas por valor o rango.
- Soporta operaciones adicionales como acceso por índice o eliminación de elementos específicos.
- Combinación de propiedades de BST y heap, lo que lo hace más versátil.

Desventajas del Treap frente al Binary Heap:

- Inserciones y eliminaciones son ligeramente más costosas en promedio debido al mantenimiento del orden de claves ($O(\log n)$ frente a $O(1)$ para heap insert en el caso promedio de heap binario completo).
- Implementación más compleja que un heap simple si solo se necesita prioridad.

VI-E. Benchmark de tiempos de ejecución

Para complementar el trabajo, se realizó un benchmark orientado a comparar el desempeño del *Treap* frente las estructuras ya mencionadas. El objetivo principal fue observar su comportamiento práctico en inserciones, búsquedas y eliminaciones, manteniendo al Treap como punto de referencia. Se realizó una implementación ligera de todas las estructuras con este fin. Los tiempos obtenidos corresponden a promedios de múltiples ejecuciones sobre 10,000 claves aleatorias. El procedimiento del experimento puede verse en este notebook de Google Colab.

Estructura	Insert (ms)	Search (ms)	Delete (ms)
Treap	26.000	8.009	1.102
AVL Tree	92.074	7.735	1.289
Red-Black Tree	56.267	10.053	1.666
Binary Heap	0.892	1.309	0.875
B+Tree	20361.558	131.689	0.000

Cuadro II
TIEMPOS PROMEDIO (EN MILISEGUNDOS).

VI-E0a. Inserción.: La Figura 6 muestra el tiempo de inserción en las tres estructuras. El Treap logra un tiempo de ejecución bajo y supera al AVL y al Red-Black Tree. El gráfico también muestra al Binary Heap, cuya eficiencia anómala se debe a la implementación perticular en el lenguaje Python. No se incluye el B+Tree pues sus tiempos extremadamente altos dificultan la visualización del gráfico.

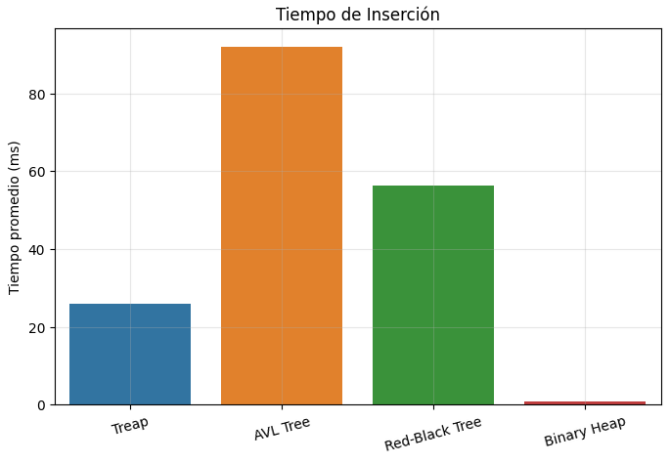

Figura 6. Tiempos de inserción

VI-E0b. Búsqueda.: En la Figura 7 se aprecia que el Treap obtiene un tiempo muy cercano al del AVL y ligeramente mejor que el del Red-Black Tree. Esto confirma que, aún siendo una estructura probabilística, su desempeño en consultas se mantiene competitivo frente a árboles balanceados determinísticos.

VI-E0c. Eliminación.: La Figura 8 muestra que el Treap mantiene un rendimiento superior al del Red-Black Tree y del AVL. Este resultado coincide con la teoría del Treap; las rotaciones esperadas suelen ser menos rígidas, por lo que la eliminación resulta más ágil y consistente.

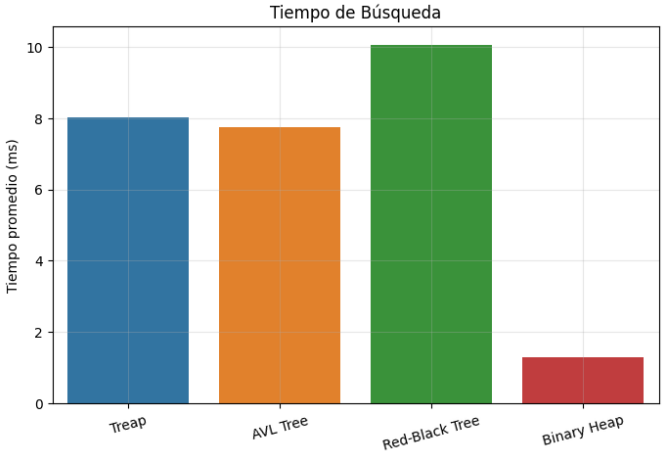

Figura 7. Tiempos de búsqueda

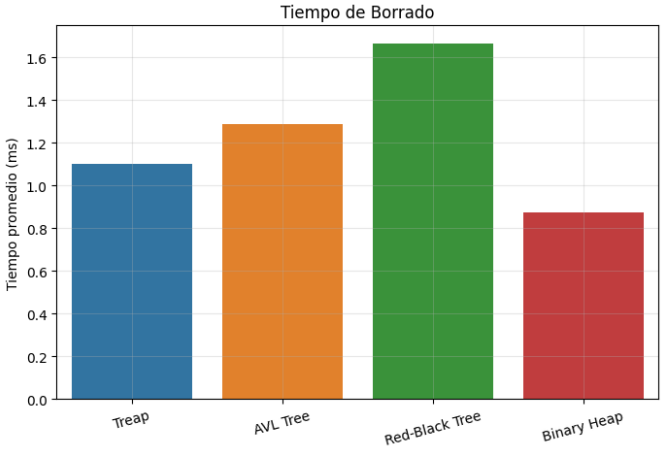

Figura 8. Tiempos de eliminación: Lista, AVL y Heap.

VII. Aplicaciones

Gracias a su estructura híbrida que combina un Ábol Binario de Búsqueda y un Heap, el Treap se presenta como una estructura flexible. Puede extenderse para soportar operaciones eficientes adicionales, adaptándose a distintas necesidades.

Algunas de sus aplicaciones incluyen:

- **Colas de prioridad ordenadas:** A diferencia de un Binary Heap, el Treap no solo gestiona prioridades, sino que también mantiene las claves ordenadas, permitiendo búsquedas por valor o rango. Esto lo hace útil para implementar colas de prioridad ordenadas.
- **Índices dinámicos:** Si cada nodo almacena información auxiliar, como el tamaño de su subárbol, el Treap permite el acceso al k-ésimo elemento (acceso por índice) en tiempo esperado $O(log\ n)$.
- **Algoritmos de rangos:** La operación de división (split) se utiliza frecuentemente en algoritmos que requieren manipular rangos o combinar resultados parciales de manera independiente.
- **Construcción dinámica de árboles:** La operación de unión (merge) es clave en algoritmos que necesitan construir árboles compuestos de forma dinámica sin perder eficiencia.

VIII. Conclusiones

El Treap es una estructura de datos que combina las propiedades de un árbol binario de búsqueda (BST) y un heap. Su principal ventaja radica en que ofrece una implementación sencilla y un rendimiento eficiente en promedio, comparado con otras estructuras auto-balanceables como los árboles AVL o Rojo-Negro, que suelen tener implementaciones más complejas.

El balanceo del Treap se logra mediante el uso de prioridades aleatorias asignadas a cada nodo, lo que sigue la propiedad de un max-heap. Esta aleatoriedad genera una alta probabilidad de que la altura del árbol se mantenga en $O(log\ n)$. Como resultado, las operaciones fundamentales de inserción, eliminación y búsqueda tienen un costo computacional esperado de $O(log\ n)$.

A pesar de su eficiencia promedio, el Treap no ofrece garantías determinísticas como el AVL. En el peor de los casos, si las prioridades aleatorias generan un árbol degenerado o lineal, la complejidad de las operaciones puede degradarse hasta $O(n)$. No obstante, su simplicidad y flexibilidad lo convierten en una herramienta valiosa en programación.

IX. Repositorio del Proyecto

El código fuente completo de la implementación del Treap, junto con la interfaz gráfica desarrollada en Qt para su visualización, se encuentra disponible de manera pública en el siguiente repositorio:

https://github.com/stefano565-Utec/Treap-Visual-AED.git

El repositorio incluye:

- La implementación del Treap en C++ con soporte para *search*, *insert*, *delete*, *split* y *merge*.
- La interfaz gráfica basada en `QGraphicsScene` para representar y manipular múltiples Treaps en tiempo real.
- Archivos del proyecto Qt (`.pro`) y código auxiliar para interacción visual.
- Instrucciones de compilación y ejecución.

El código se publica con fines académicos y educativos, permitiendo su uso, análisis y extensión.

Referencias

[1] Seidel, R., & Aragon, C. R. (1996). Randomized search trees. *Algorithmica*, 16(4–5), 464–497. https://doi.org/10.1007/bf01940876

[2] Cormen, T.H., Leiserson, C.E., Rivest, R.L. and Stein, C. (2009) Introduction to Algorithms. The MIT Press, Cambridge, Massachusetts. - References - Scientific Research Publishing. (n.d.). https://www.scirp.org/reference/referencespapers?referenceid=2144584

[3] Weiss, M. A. (2005). Data Structures and Algorithm Analysis in C++ (3rd Edition). In Addison-Wesley Longman Publishing Co., Inc. eBooks. https://dl.acm.org/citation.cfm?id=1204644

[4] Sedgewick, R., & Wayne, K. (2015). Algorithms, Fourth Edition (Deluxe): Book and 24-Part lecture series. https://www.amazon.com/Algorithms-Fourth-Deluxe-24-Part-Lecture/dp/0134384687