LABORATORIO 4

Estructura de datos - Treap

2024-2025

Abraham Ruiz Mosteirín y Míriam Rodríguez Carranza

Contenido

1. Introducción

El presente informe detalla la implementación y pruebas de estructuras de datos aleatorizadas, con un enfoque en los Árboles Binarios de Búsqueda (BST) no balanceados y los Treaps aleatorizados. Estas estructuras combinan principios fundamentales de búsqueda eficiente y aleatorización para garantizar un buen rendimiento incluso bajo inserciones arbitrarias.

Los objetivos principales de este proyecto son:

- Comprender el funcionamiento de estas estructuras, sus propiedades y sus invariantes.
- Implementar un BST no balanceado y un Treap aleatorizado que respeten sus invariantes.
- Realizar pruebas exhaustivas para validar su correcto funcionamiento.
- Analizar y comparar el comportamiento de ambas estructuras en diferentes escenarios.

Te explicaré las implementaciones de las clases que has proporcionado: **BST**, **Treap**, y sus correspondientes pruebas. Estas clases representan estructuras de datos de árboles binarios de búsqueda (BST) y árboles treap, que combinan las propiedades de los árboles binarios de búsqueda con un valor adicional llamado **prioridad**. Además, también explico las pruebas unitarias para verificar que estas implementaciones funcionen correctamente.

1. BST (Binary Search Tree)

Descripción general: El **BST** es una estructura de datos donde cada nodo tiene dos hijos: el hijo izquierdo contiene valores menores que el nodo y el hijo derecho contiene valores mayores. En este caso, la clase implementa una estructura de mapa (Map<Key, Value>) con operaciones comunes como inserción (put), búsqueda (get), eliminación (remove) y comprobación de existencia (containsKey).

Componentes clave:

- **Node:** Esta clase interna representa un nodo del árbol y contiene las siguientes propiedades:
  - key: La clave que se usa para ordenar los nodos.
  - value: El valor asociado a la clave.
  - left y right: Referencias a los hijos izquierdo y derecho del nodo.
- **put(Key key, Value value):** Inserta una clave y su valor en el árbol. Si el árbol está vacío, se crea un nuevo nodo. Si la clave ya existe, simplemente actualiza el valor.
- **get(Key key):** Busca un valor asociado con una clave específica. Si la clave no existe, devuelve null.
- **remove(Key key):** Elimina un nodo del árbol. Si el nodo tiene dos hijos, se elige un nodo sustituto (generalmente el menor del subárbol derecho) para mantener la propiedad del árbol.
- **containsKey(Key key):** Verifica si una clave está presente en el árbol.
- **height():** Calcula la altura del árbol, definida como la longitud del camino más largo desde la raíz hasta una hoja. La altura de un árbol vacío es -1.

2. Treap (Tree + Heap)

El **Treap** es una estructura de datos que combina las propiedades de un árbol binario de búsqueda (BST) y un montón binario (heap). En un Treap, cada nodo tiene una clave que se organiza como en un BST, pero también tiene una prioridad aleatoria que se utiliza para equilibrar el árbol de acuerdo con las propiedades de un heap (por lo general, un **heap máximo**).

Componentes clave:

- **Node:** Cada nodo tiene:
  - key: Similar al BST, representa la clave que organiza el árbol.
  - value: El valor asociado a la clave.
  - priority: Un valor aleatorio entre 0 y 1 que se asigna al nodo y se usa para mantener el orden de heap.
- **put(Key key, Value value):** Al igual que en el BST, se inserta una clave y su valor. Sin embargo, después de la inserción, se mantiene la

propiedad del heap realizando rotaciones (derecha o izquierda) si el nodo violó la propiedad de heap. Si el nodo tiene un hijo con mayor prioridad, se rota para "subir" el hijo y mantener el orden de las prioridades.

- **get(Key key):** Realiza una búsqueda similar a la del BST, pero también mantiene las propiedades del BST y del Treap.
- **remove(Key key):** La eliminación en un Treap también sigue el procedimiento estándar del BST, pero después de cada eliminación, se realizan rotaciones de manera que se preserven las propiedades tanto del árbol binario de búsqueda como del heap.
- **Rotaciones (rotateRight y rotateLeft):** Estas operaciones son fundamentales en el Treap, ya que mantienen las propiedades de heap (prioridades) después de la inserción o eliminación de nodos. En una rotación:
    - **rotateRight:** Realiza una rotación a la derecha, moviendo el nodo izquierdo del árbol hacia arriba y el nodo actual hacia abajo.
    - **rotateLeft:** Realiza una rotación a la izquierda, moviendo el nodo derecho del árbol hacia arriba.

3. Pruebas de Treap (TreapTest)

Las pruebas unitarias para el **Treap** se realizan utilizando JUnit. Estas pruebas aseguran que el Treap funciona correctamente bajo diversas condiciones.

Componentes clave de las pruebas:

- **setUp():** Inicializa una nueva instancia de Treap antes de cada prueba.
- **testHeigh():** Inserta elementos en el Treap y verifica que la altura del árbol después de la inserción no supere un múltiplo de log n. Esto asegura que el Treap mantiene una altura logarítmica (o cercana) a pesar de las inserciones.
- **testRandomInsertions():** Inserta elementos aleatorios en el Treap y verifica que la altura permanezca baja, lo que indica que el Treap está equilibrado.
- **testKeyRetrieval():** Inserta un par clave-valor en el Treap y luego verifica si el valor se puede recuperar correctamente mediante la clave.

- **testKeyRemoval():** Inserta un par clave-valor en el Treap, elimina la clave y verifica que la clave se haya eliminado correctamente.

4. Pruebas de BST (BSTTest)

Las pruebas unitarias para el **BST** aseguran que el árbol binario de búsqueda funcione correctamente con varias operaciones de inserción, eliminación y búsqueda.

Componentes clave de las pruebas:

- **testAscendingInsertion():** Inserta elementos en orden ascendente y verifica que el árbol sea degenerado (es decir, una lista enlazada). También verifica que la altura sea la correcta y que los elementos sean accesibles.
- **testDescendingInsertion():** Inserta elementos en orden descendente y realiza las mismas verificaciones que en el test anterior.
- **testRemovalInDegenerateTree():** Después de crear un árbol degenerado (por ejemplo, insertando elementos en orden ascendente), elimina un nodo intermedio y verifica que el árbol se actualice correctamente y que otros elementos sean accesibles.
- **testLargeUnbalancedTree():** Inserta una gran cantidad de elementos en orden y verifica la altura del árbol, asegurando que el árbol sea altamente desequilibrado debido a la inserción secuencial.
- **testDuplicateInsertion():** Verifica que si se inserta una clave duplicada, el valor asociado se actualice correctamente.
- **testRandomInsertion():** Inserta elementos en un orden aleatorio y luego verifica que todos los elementos sean accesibles a través de las claves correspondientes.

Resumen de la Implementación:

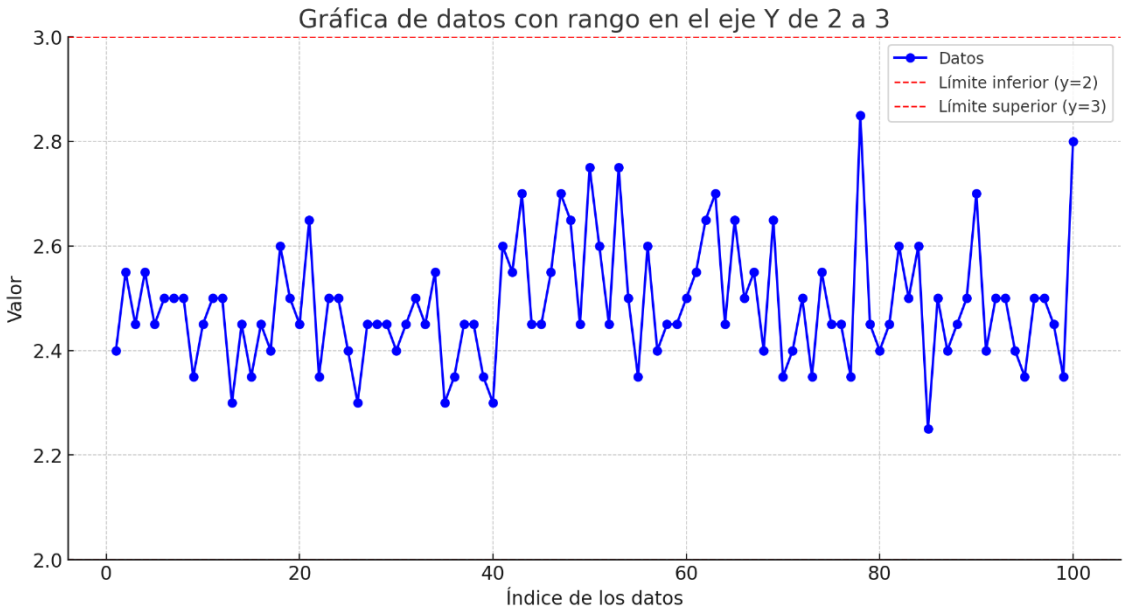
- **BST:** Implementación de un árbol binario de búsqueda clásico con las operaciones estándar.

- **Treap:** Una estructura de datos que combina un árbol binario de búsqueda con un heap, utilizando rotaciones para equilibrar el árbol y mantener las prioridades.
- **Pruebas unitarias:** Ambas clases (BST y Treap) tienen pruebas unitarias para verificar la funcionalidad de inserción, eliminación, y recuperación de claves, además de asegurarse de que las estructuras se comporten correctamente bajo condiciones aleatorias y extremas.

Esta implementación de Treap ofrece una forma eficiente de mantener un árbol balanceado, mientras que el BST es más sencillo pero puede volverse ineficiente si no se equilibra adecuadamente. Las pruebas aseguran que ambas estructuras funcionen correctamente y mantengan su eficiencia en términos de tiempo de ejecución.

5. Experimentos y Observaciones

Experimento 2

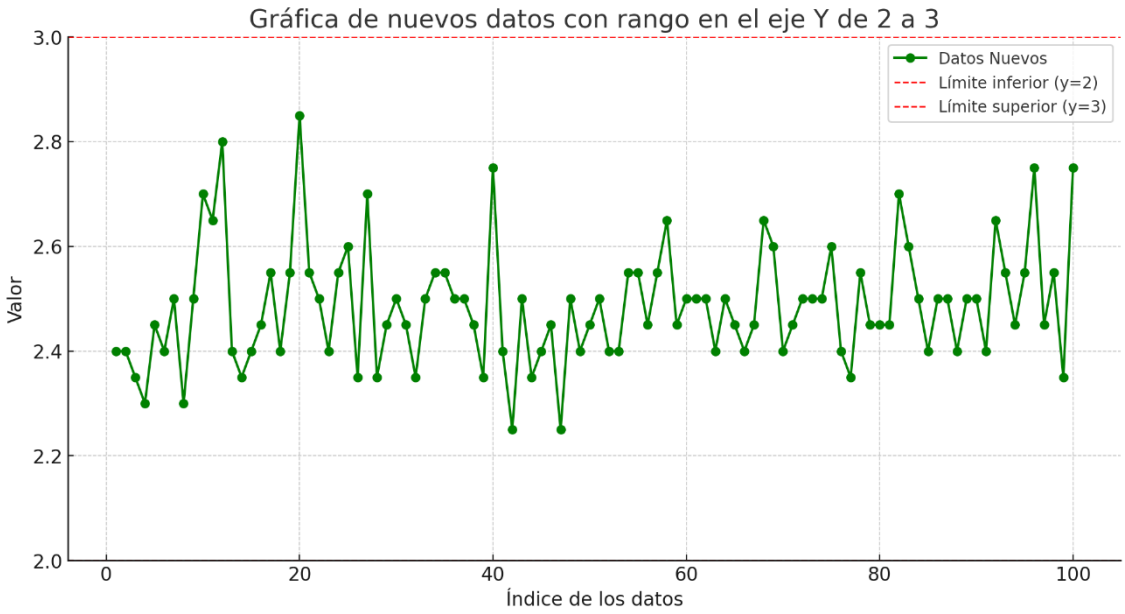


Datos: Representados con una línea azul y marcadores circulares.
Los valores oscilan entre 2.2 y 2.8 aproximadamente.
Se observa una tendencia fluctuante con picos y valles, manteniéndose dentro de los límites establecidos.
Límite inferior: Una línea roja discontinua ubicada en y = 2.
Límite superior: Una línea roja discontinua ubicada en y = 3.

Eje X (horizontal): Etiquetado como "Índice de los datos", indicando que los datos están numerados del 0 al 100.

Eje Y (vertical): Etiquetado como "Valor", con un rango que se extiende de 2.0 a 3.0.

Experimento 3



Datos Nuevos: Los valores oscilan entre 2.2 y 2.8 aproximadamente, aunque con variaciones sutilmente diferentes respecto al primer gráfico.

La tendencia es igualmente fluctuante y se mantiene dentro de los límites definidos.

Límite inferior: Una línea roja discontinua en y = 2.

Límite superior: Una línea roja discontinua en y = 3.

Eje X (horizontal): Etiquetado como "Índice de los datos", numerado del 0 al 100.

Eje Y (vertical): Etiquetado como "Valor", con un rango que va de 2.0 a 3.0.

Título: "Gráfica de nuevos datos con rango en el eje Y de 2 a 3".

6. Conclusiones

El BST es una estructura sencilla, pero sufre degradación en inserciones ordenadas. Por otro lado, el Treap mantiene un equilibrio eficiente gracias a la aleatorización, demostrando ser más adecuado para aplicaciones dinámicas.

7. Bibliografia

Robert Sedgewick, Kevin Wayne, Algorithms, Fourth Edition.Pearson Education, 2011.

Maurice Naftalin, Philip Wadler. Java Generics and Collections O'Reilly, 2007. USA

GeeksforGeeks

GitHub