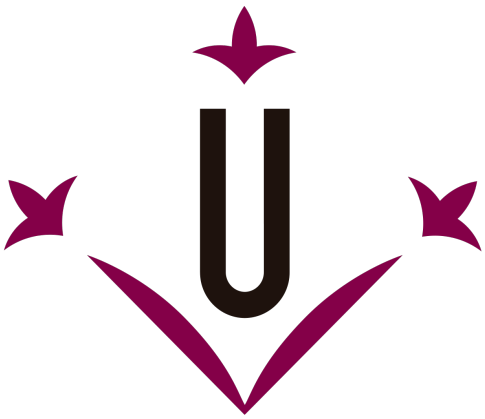

Universitat de Lleida

Treaps

Data Structure

Escola Politècnica Superior

Computer Science

Yhislaine Nataly, Jaya Salazar

Valeria Adriana, Escalera Flores

22.12.2024

Contents

1 Introduction

This practice focuses on exploring and implementing randomized treaps, a type of binary search tree that maintains balance through randomized priorities. The objectives include understanding the use of randomness in data structures, implementing unbalanced binary search trees and randomized treaps, and analyzing their performance through experiments. Additionally, the practice involves reasoning about class invariants to ensure correctness in insertion and deletion operations. These concepts are essential for comprehending advanced data structure techniques and their applications.

2 *randomized treaps*

The main idea behind *randomized treaps* is to make two properties compatible:

- All elements have the same probability of being at any level.
- Elements can be inserted in any order.

The solution is based on combining two data structures:

- Binary search trees (which will be used to implement the key-value association in the structure).
- Min-heaps, where each node will have a level in the tree based on its priority value (as if we were implementing a priority queue).

In other words, these two properties will be the invariants of the representation, and the operations can assume that they are satisfied beforehand, and will have to ensure that they remain satisfied afterward.

The goal of this part of the practice is for you to implement the `Map` type using *randomized treaps*.

2.1 Implementation

The implementation of *randomized treaps* aims to achieve a balanced binary search tree structure that maintains its properties through randomized priorities assigned to each node. Below is the Java implementation and explanation of the critical methods.

2.2 Code Explanation

```
@Override
public void put(Key key, Value value) {
    if (key == null) throw new NullPointerException("Key is null");
    root = put(root, key, value);
}
```

The put method allows the insertion of a new key-value pair into the treap. If the key already exists, its associated value is updated. Otherwise, a new node is created and positioned within the treap according to its binary search tree (BST) property.

```
private Node<Key, Value> put(Node<Key, Value> h, Key key, Value value)
   {
    if (h == null) return new Node<>(key, value);

    int cmp = key.compareTo(h.key);
    if (cmp < 0) {
        h.left = put(h.left, key, value);
        if (h.left.priority < h.priority) {
            h = rotateRight(h);
        }
    } else if (cmp > 0) {
        h.right = put(h.right, key, value);
        if (h.right.priority < h.priority) {
            h = rotateLeft(h);
        }
    } else {
        h.value = value;
    }

    return h;
}
```

The put helper method recursively traverses the treap to find the correct position for the new node. Upon insertion, the method ensures the heap property by performing rotations when a child node's priority is smaller than its parent's priority.

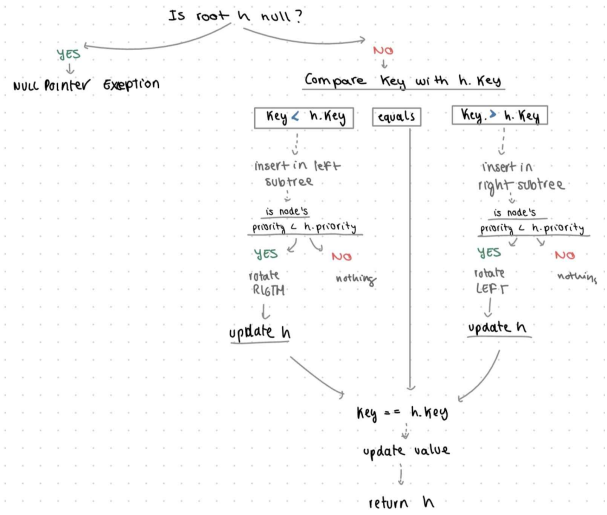

Figure 1: Put schema

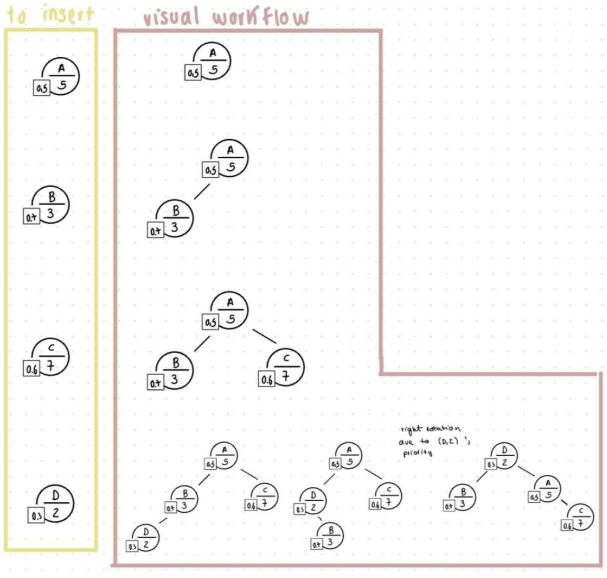

Figure 2: Put workflow

2.3 Rotation Methods

Rotations are key operations in treaps to restore their heap properties after insertion or deletion.

```
01 | private Node<Key, Value> rotateRight(Node<Key, Value> h) {
02 |     Node<Key, Value> x = h.left;
03 |     h.left = x.right;
04 |     x.right = h;
05 |     return x;
06 | }
07 |
08 | private Node<Key, Value> rotateLeft(Node<Key, Value> h) {
09 |     Node<Key, Value> x = h.right;
10 |     h.right = x.left;
11 |     x.left = h;
12 |     return x;
13 | }
```

The `rotateRight` and `rotateLeft` methods are used to adjust the tree's structure when priorities are violated. A right rotation moves the left child up, while a left rotation moves the right child up, preserving the BST and heap properties.

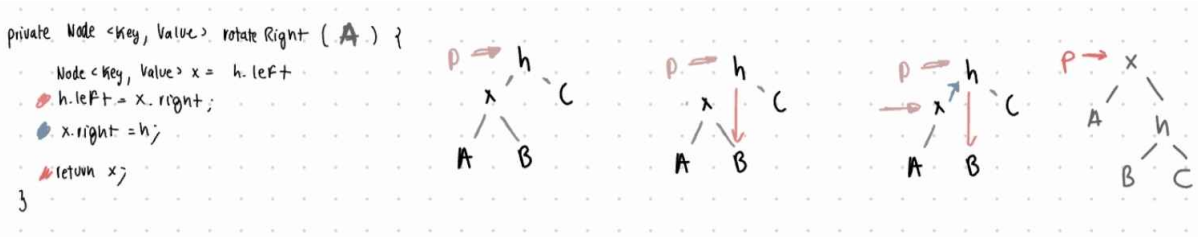

Figure 3: Rotation explanation

2.4 Deletion Method

```
01 | @Override
02 | public void remove(Key key) {
03 |     if (key == null) throw new NullPointerException("Key is null");
04 |     root = remove(root, key);
05 | }
06 |
```

```
07 | private Node<Key, Value> remove(Node<Key, Value> h, Key key) {
08 |     if (h == null) return null;
09 |
10 |     int cmp = key.compareTo(h.key);
11 |     if (cmp < 0) {
12 |         h.left = remove(h.left, key);
13 |     } else if (cmp > 0) {
14 |         h.right = remove(h.right, key);
15 |     } else {
16 |         if (h.left == null) return h.right;
17 |         if (h.right == null) return h.left;
18 |
19 |         if (h.left.priority < h.right.priority) {
20 |             h = rotateRight(h);
21 |             h.right = remove(h.right, key);
22 |         } else {
23 |             h = rotateLeft(h);
24 |             h.left = remove(h.left, key);
25 |         }
26 |     }
27 |     return h;
28 | }
```

The `remove` method deletes a key from the treap while maintaining its properties. If the node to be removed has two children, the method performs rotations to push the node down the tree until it becomes a leaf, at which point it can be removed safely.

3 Tests

This test suite verifies the functionality of our `HeapSort` class implementation using JUnit. These tests cover essential scenarios. Each test follows the next code structure:

3.1 add tests

Add tests evaluate the heap's ability to insert elements and rearrange them while maintaining its structure. We have the test `testAdd`, `testAddToHeap`, `testAddWithIntegers`

and addElements. The first test initialises an empty heap with a capacity of five elements and inserts different values. The `testAddToHeap` test initialises a heap with an array 16, 14, 10, 8, 8, 7, 9, 3, 4, 2 using a reverse-order comparator. The test `testAddWithIntegers` is very similar to testadd. To conclude with the addition tests, in the addElements test a heap is initialised with an array containing ten elements and an empty position at the end, represented by a `null`. The heap uses a natural order comparator.

Here is an example:

```
01 | @Test
02 |     void addElement() {
03 |         // We initialise the heap with an array of fixed size.
04 |         Integer[] elements = {16, 14, 10, 8, 7, 9, 3, 2, 4, 1, null };
05 |         var heap = new HeapSort.BinaryHeap<>(elements, Comparator.
       naturalOrder());
06 |         heap.heapSize = 10; // Define the current size of the heap.
07 |
08 |         // We add a new element to the heap.
09 |         heap.add(12);
10 |
11 |         // Expected result after reorganising the heap.
12 |         Integer[] expected = {16, 14, 10, 8, 12, 9, 3, 2, 4, 1, 7};
13 |
14 |         // We check that the array in the heap matches the expected
       result.
15 |         assertArrayEquals(expected, elements);
16 |     }
```

3.2 remove tests

As for the tests of the remove method, they evaluate the correct removal of the root element and the reorganisation of the heap to preserve its structure. We have the `testRemoveMax`, `testRemoveRoot`, `deleteRoot` and `deleteRoot` tests. In the first test, a max-heap initialised with 4, 3, 2, 1 is used and two consecutive elements are deleted. The test `testRemoveRoot` evaluates a more complex case in a max-heap with the initial array 16, 14, 10, 8, 7, 9, 3, 4, 2. After removing the root 16, the heap is reorganised and the new max-heap, 14, occupies the root, confirming that the remove method respects

the heap property. In the `deleteRoot` test, the root element of the heap is removed and the remaining elements are reorganised. A reverse order comparator is used. Finally, the testRemoveElement test, quite similar to the other tests, with an initial array of 9 elements and a couple of nulls.

Here is an example:

```
01 |     @Test
02 |     void testRemoveElement() {
03 |         Integer[] elements = {16, 14, 10, 8, 7, 9, 3, 4, 2, null, null
   };
04 |         var heap = new HeapSort.BinaryHeap<>(elements, Comparator.
   reverseOrder());
05 |         heap.heapSize = 9;
06 |
07 |         Integer removed = heap.remove();
08 |         assertEquals(16, removed); // Deletes the root.
09 |         assertEquals(14, heap.elements[0]); // New root.
10 |         assertEquals(8, heap.elements[1]);
11 |     }
```

3.3 other methods tests

To check the correct behaviour of the rest of the methods we have the test `testSwap`, to check our swap method, we also have the `testHeapDown, testHeapUpper`, the test `testHasParent` and the test `testHasLeftAndHasRight`, to check the methods of the `BinaryHeap<E>`. On the other side, we also have `testAddAndRemove` and `testAddAndRemoveHeap` which are tests that combine the add and remove methods.

The tests provide adequate coverage to validate the behaviour of the add and remove methods, checking their functionality and verifying the correct handling of insertions and removals in various situations. This ensures that the implementation meets the requirements and behaves reliably in the tested cases.

A remarkable point is the use of `Comparator.naturalOrder()` and `Comparator.reverseOrder()`, which brings flexibility to the `HeapSort` algorithm. They allow the same code to work for both ascending and descending order. This makes the implementation reusable and adaptable, which is crucial in contexts where sorting requirements may vary.

For example:

```
Integer[] elements = {16, 14, 10, 8, 7, 9, 3, 2, 4, 1, null };
     var heap = new HeapSort.BinaryHeap<>(elements, Comparator.
naturalOrder());
```

and:

```
Integer[] elements = {16, 14, 10, 8, 7, 9, 3, 4, 2};
     var heap = new HeapSort.BinaryHeap<>(elements, Comparator.
reverseOrder());
```

4 Experiment 1: perfectly balanced tree

The first experiment aimed to construct a perfectly balanced binary search tree (BST) from an ordered list of integers. To do so, we designed and implemented the recursive function `balancedInsertions(BST bst, ArrayList elems, int begin, int end)`.

First, an ordered list of integers is generated using the `makeList` method. Then, the `balancedInsertions` method is used to insert the elements into the BST so that the tree is balanced. The key is the recursive function, which selects the middle element as the root, splits the list into two halves and processes them recursively, and terminates when the sublist has no elements, in other words, the base case `(begin > end)`.

```
    private static void balancedInsertions(
        BST<Integer, Integer> bst,
        ArrayList<Integer> elems,
        int begin,
        int end) {

    if (begin > end) return;

    int mid = (begin + end) / 2;
    bst.put(elems.get(mid), elems.get(mid));

   balancedInsertions(bst, elems, begin, mid - 1); //left
   balancedInsertions(bst, elems, mid + 1, end); // right
 }
```

More specifically, in this experiment we generated a list of size `2 (log + 1) - 1` (in this case, with log = 20, the size is 2,097,151). Then, the `balancedInsertions` function

is used to insert the elements so that the resulting tree is balanced. Finally, its height is calculated with `bst.height()`. The expected and theoretical result is 20, which corresponds to a balanced tree with that number of nodes. The result also shows that the height obtained is equal to 20, which confirms that the tree is perfectly balanced. This confirms that the method works correctly in creating an optimal tree for efficient searches.

```
01 |         public class Experiment1 {
02 |
03 |     public static void main(String[] args) {
04 |         int log = 20;
05 |         int size = 1 << (log + 1) - 1;
06 |         var elems = makeList(size);
07 |         var bst = balancedInsertions(elems);
08 |         System.out.println(bst.height());
09 |     }
10 | }
```

5 Experiment 2: perfectly balanced tree

In this part of the practice, we have repeated experiment 2 with our implementation of BST. The graph obtained for 100 runs of the experiment is the following:

The results show that even with random insertions, the height of the BST remains roughly proportional to O(log2 n), with a height ratio that, in our case, ranges between 2.25 and 2.75.

```
01 |         public class Experiment1 {
02 |
03 |     public static void main(String[] args) {
04 |         int log = 20;
05 |         int size = 1 << (log + 1) - 1;
06 |         var elems = makeList(size);
07 |         var bst = balancedInsertions(elems);
08 |         System.out.println(bst.height());
09 |     }
10 | }
```

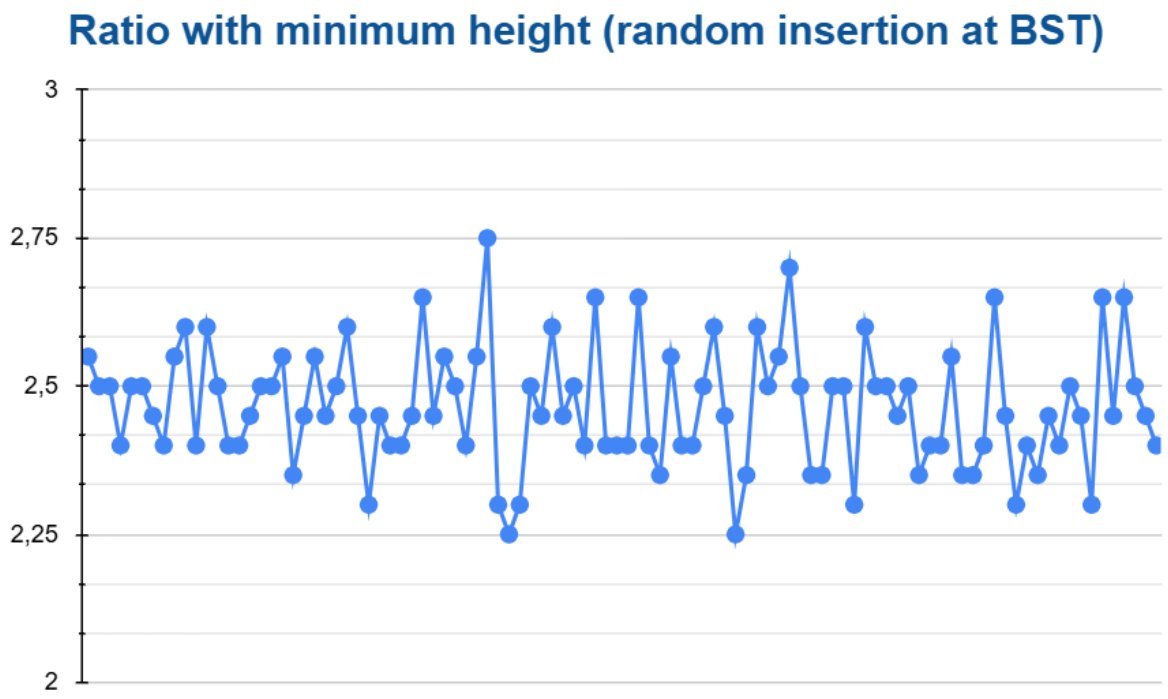

Figure 4: Graph obtained for 100 executions

6 Experiment 3

The experiment creates an ordered list of elements and inserts them sequentially into a Treap. It measures the resulting Treap's height and calculates the ratio with the expected minimum height (log 2 (n) log 2 (n)).

The graph showing the height ratios (actual height / minimum height) for 100 trials would be presented here. The expected behavior is that the ratios remain close to a small constant factor, demonstrating that randomized Treaps maintain balance even with non-randomized data.

The *randomized treap* combines the strengths of binary search trees and heaps, ensuring efficient operations for insertion, deletion, and search with an average time complexity of $O(\log n)$. Randomized priorities provide balance without the overhead of additional rebalancing algorithms, making treaps an elegant solution for dynamic sets.

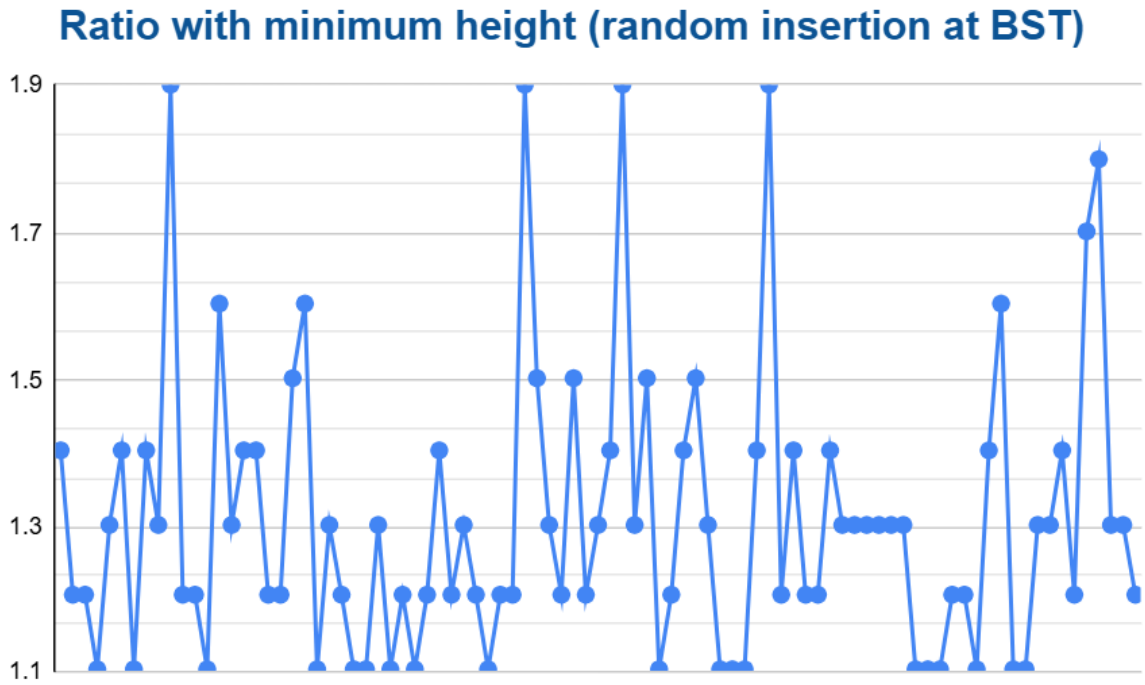

Figure 5: Graph obtained for 100 executions

7 Conclusions

Randomized treaps efficiently combine the properties of binary search trees (BSTs) and heaps, achieving an average time complexity of $O(\log n)$ for insertion, deletion, and search operations. This efficiency is maintained thanks to randomized prioritization, which ensures balance with minimal computational overhead, making treaps a lightweight yet powerful data structure for dynamic operations.

The experiments conducted validate the theoretical properties of treaps. Balanced insertions yielded optimal heights, confirming the correctness of the implementation. Similarly, randomized insertions demonstrated height ratios close to $O(\log n)$, showcasing the robustness of the treap's balancing mechanism even under random insertion patterns.

In comparison to traditional balanced trees like AVL or Red-Black trees, treaps offer a simpler implementation by relying on randomness rather than deterministic rebalancing methods. This simplicity does not come at the cost of performance, as treaps achieve comparable efficiency while requiring fewer adjustments during updates.

The scalability of treaps is another notable advantage. Experimental results with large datasets, including 10^6 nodes, demonstrate that treaps maintain efficient operations and

balance across varying input sizes, making them suitable for real-world applications where data size and complexity grow dynamically.

Treaps are especially applicable in scenarios requiring frequent updates and searches, such as priority queues and associative arrays. Their ability to maintain balance dynamically without prior knowledge of the data order ensures reliability and performance in diverse use cases.

Future research could explore adapting treaps to other data types and priority schemes. Evaluating their behavior in distributed or concurrent systems presents another promising area for exploration, potentially broadening their applicability in modern computational environments.

References

[1] JM Gimeno, *Tree Structures*, accessed December 20, 2024. Available at: https://cv.udl.cat/access/content/group/102010-2425/Temario/4%20-%20Tree%20Structures%20_v10_.pdf.