Executive Summary: Treap

Huarui Liu, Zicheng He, Jingzhou Qiu

April 23, 2025

1 Introduction / Problem Statement:

Efficient data structures are crucial for managing dynamic sets of ordered data, supporting operations like insertion, deletion, and search. Traditional self-balancing binary search trees (BSTs), such as AVL and Red-Black trees, guarantee logarithmic height through strict balancing rules, which can be complex to implement and maintain.

While AVL and Red-Black trees are designed to maintain balanced structures, they come with inherent complexities:

- **AVL Trees:** These trees enforce strict balance by ensuring that the height difference between left and right subtrees of any node is at most one. This strictness necessitates frequent rotations during insertions and deletions, leading to increased overhead and more complex implementation.
- **Red-Black Trees:** Although they offer a more relaxed balancing approach compared to AVL trees, Red-Black trees still involve intricate rules and multiple cases for rebalancing. This complexity can complicate implementation and maintenance.

Thus, there is a need for a simpler yet efficient data structure that not only maintains balanced tree height to ensure logarithmic-time operations such as insert, delete, and search, potentially extendable to support advanced capabilities like range queries and range modifications in logarithmic time.

2 Overall Intuitive Approach / Solution:

2.1 Overview

A **Treap** is a randomized binary search tree (BST) that maintains its balance through randomly assigned priorities, effectively combining properties of both trees and heaps—hence the name (*tree* + *heap*). Like standard BSTs, Treaps facilitate efficient searching, insertion, and deletion of keys. However, unlike deterministic balanced trees such as AVL or red-black trees, Treaps rely on probabilistic balancing by assigning a random priority to each node.

The heap property is crucial in this structure: each parent node must have a higher priority than its children. Consequently, the highest-priority node becomes the root, and priority strictly decreases down each branch. This ensures that, while the BST ordering maintains sorted traversal and search efficiency, the heap ordering imposes a randomized yet balanced structure. As a result, Treaps naturally and probabilistically balance themselves, guaranteeing expected logarithmic time for essential operations with significantly simpler implementation than deterministic balancing methods.

2.2 Structure and Key Properties

Each node in a Treap contains two essential values:

- **Key (BST Property):** Ensures that keys follow standard BST ordering. For any node, all keys in the left subtree are smaller, and all keys in the right subtree are larger. An in-order traversal yields the keys in sorted order.
- **Priority (Heap Property):** Each node has a randomly assigned priority, typically unique and independent of its key. The Treap maintains the heap property based on these priorities—each parent node has a higher priority than its descendants.

These two properties (BST and heap) uniquely determine the structure of the Treap for any given set of key–priority pairs. The random assignment of priorities effectively creates a Cartesian tree based on keys and priorities, inherently imposing a randomized BST structure.

2.3 Randomized Balancing and Efficiency

The use of random priorities ensures the Treap's structural balance independent of insertion or deletion order. A Treap can be thought of as inserting nodes by descending priority order into an initially empty BST, thereby simulating a random insertion sequence. Consequently, the Treap's structure closely resembles a randomly built BST, known for having height proportional to with high probability.

Implementation of fundamental operations is straightforward:

- **Insertion:** Nodes are first inserted using standard BST insertion rules based on keys. After insertion, rotations (either left or right) may be performed to restore the heap priority order.
- **Deletion:** Nodes are removed by rotating the target node down until it becomes a leaf, after which it is easily removed.
- **Search:** Standard BST search methods apply, leveraging the key-order property without any need for heap priorities.

This inherent balance ensures fundamental operations (search, insertion, and deletion) have expected logarithmic performance. Crucially, this guarantee is independent of key insertion order, as balancing arises purely from the internally generated random priorities. Extreme imbalance is statistically improbable, with probability diminishing exponentially as node count increases.

3 Implementation

A Treap maintains two invariants simultaneously:
1. BST property: for any node, all keys in its left subtree are smaller, and all in its right subtree are larger.
2. Heap property: each node's random priority is no greater than the priorities of its children.

All operations run in expected $O(\log n)$ by using local rotations to restore the heap property without disturbing the BST order.

Right Rotation When a node y has a left child x whose priority is higher (i.e. numerically smaller) than y's, we "lift" x up:

Algorithm 1 RightRotate(y)

```
1: procedure RightRotate(y)
2:     x ← y.left
3:     y.left ← x.right
4:     x.right ← y
5:     return x
6: end procedure
```

Left Rotation Symmetrically, if a node x has a right child y with higher priority than x, we lift y up:

Algorithm 2 LeftRotate(x)

```
1: procedure LeftRotate(x)
2:     y ← x.right
3:     x.right ← y.left
4:     y.left ← x
5:     return y
6: end procedure
```

Split To split a Treap T by key k into (L, R) with all keys in $L \le k$ and in $R > k$, recurse on the root:

Algorithm 3 split(T, k)

```
1: procedure SPLIT(T, k)
2:     if T = null then return (null, null)
3:     end if
4:     if k < T.key then
5:         (L, R') ← split(T.left, k)
6:         T.left ← R'
7:         return (L, T)
8:     else
9:         (L', R) ← split(T.right, k)
10:        T.right ← L'
11:        return (T, R)
12:    end if
13: end procedure
```

Merge Given two Treaps L and R with $\max L < \min R$, choose the root whose priority is smaller, then recurse to stitch the other tree underneath:

Algorithm 4 merge(L, R)

```
1: procedure MERGE(L, R)
2:     if L = null then return R
3:     end if
4:     if R = null then return L
5:     end if
6:     if L.priority < R.priority then
7:         L.right ← merge(L.right, R)
8:         return L
9:     else
10:        R.left ← merge(L, R.left)
11:        return R
12:    end if
13: end procedure
```

Insertion Insert by key like in a BST, assign a random priority, then rotate up until the new node's priority is no worse than its parent's:

Algorithm 5 insert(T, x)

```
1: procedure INSERT(T, x)
2:     if T = null then return x                          ▷ new leaf with random priority
3:     end if
4:     if x.key < T.key then
5:         T.left ← insert(T.left, x)
6:         if T.left.priority < T.priority then
7:             T ← RightRotate(T)
8:         end if
9:     else
10:        T.right ← insert(T.right, x)
11:        if T.right.priority < T.priority then
12:            T ← LeftRotate(T)
13:        end if
14:    end if
15:    return T
16: end procedure
```

Deletion To delete key k from Treap T, proceed as in a BST until you find the node z with $z.\text{key} = k$. Then, as long as z has two children, rotate it down towards its child of smaller priority. At that point z has at most one child; simply replace z with its non-null child (or null).

Algorithm 6 delete(T, k)

```
1: procedure DELETE(T, k)
2:     if T = null then
3:         return null
4:     end if
5:     if k < T.key then
6:         T.left ← delete(T.left, k)
7:     else if k > T.key then
8:         T.right ← delete(T.right, k)
9:     else
10:        if T.left ≠ null and T.right ≠ null then
11:            if T.left.priority < T.right.priority then
12:                T ← RightRotate(T)
13:            else
14:                T ← LeftRotate(T)
15:            end if
16:            return delete(T, k)
17:        else
18:            return (T.left ≠ null ? T.left : T.right)
19:        end if
20:    end if
21: end procedure
```

4 Summary of Programming Challenge:

The *Banana Quest* programming challenge introduces students to the practical application and implementation of the Treap data structure in a simulation-based problem. In this task, students must manage a system where bananas are added and removed from specific positions on a number line, and must efficiently process queries asking how many bananas a monkey can eat within a given time limit. The monkey moves to the right from position zero, deciding whether to wait for each banana to ripen or skip it based on the time remaining.

The core learning objectives of this challenge are:

- **Maintaining efficient data structures under updates:** Students practice managing a balanced binary search tree that must remain performant under frequent insertions and deletions. The Treap's randomized balancing and recursive operations make it a natural fit once students identify the core requirements of the problem.
- **Combining structural logic with behavioral reasoning:** Beyond just implementing standard operations, students must simulate traversal behavior that depends not only on ordering, but also on timing constraints. This adds a layer of decision-making that forces students to think carefully about how their data structure interacts with the logic of the problem.

One of the most valuable aspects of this challenge, we think, is the realization process itself. At first glance, the problem may not seem to require a Treap at all. However, as students work through the constraints—needing fast updates by key, ordered traversal, and efficient querying—they are led to consider more advanced structures like Treaps to achieve $O(\log n)$ runtimes. This makes the design decision part of the learning experience, not just the implementation.

In addition, the twist that sets this challenge apart from a typical Treap assignment is the integration of timing-aware simulation. Students must traverse bananas in increasing order of position while reasoning about current time, banana ripeness, and whether there's time to wait. This blends algorithmic reasoning with real-world constraints and pushes students to apply data structures in a more nuanced and applied way.

To ensure correctness and evaluate performance, we designed a diverse set of 20 test cases. These include both small edge cases to check logic correctness and large-scale cases to evaluate runtime behavior. The list below outlines each case:

1. Basic No Banana
2. *Edge case* – Banana Already Ripe
3. *Edge case* – Banana Not Yet Ripe
4. *Edge case* – Banana Ripens At Arrival
5. *Edge case* – Monkey Must Wait
6. *Edge case* – Monkey Chooses Not to Wait
7. *Edge case* – Remove Banana Before Query
8. *Edge case* – Multiple Bananas, All Edible
9. *Edge case* – Query with Deadline Before Arrival
10. *Edge case* – Banana Ripens Exactly at Deadline
11. *Runtime* – 1,000 randomly generated operations
12. *Runtime* – 5,000 randomly generated operations
13. *Runtime* – 10,000 randomly generated operations
14. *Runtime* – 15,000 randomly generated operations
15. *Runtime* – 30,000 randomly generated operations
16. *Runtime* – 70,000 randomly generated operations
17. *Runtime* – 100,000 randomly generated operations
18. *Runtime* – 150,000 randomly generated operations
19. *Runtime* – 200,000 randomly generated operations
20. *Runtime* – 200,000 randomly generated operations

5 Key Ideas for Solving Programming Challenge:

The critical component of solving the *Banana Quest* challenge lies in combining efficient Treap operations with accurate and performant simulation logic. Key ideas students must grasp include:

- **Dynamic Treap Management:** Maintaining a balanced Treap to quickly handle insertions and deletions in $O(\log n)$ average time complexity.
- **Sorted Traversal (In-order Traversal):** Utilizing the Treap structure to efficiently traverse bananas by position. This traversal must be efficient, as it directly simulates the monkey's linear path.
- **Simulation Logic Integration:** Properly handling multiple decision branches during traversal:
  - Immediate eating when bananas are ripe upon arrival.
  - Conditional waiting when bananas become ripe soon after arrival.
  - Skipping logic to efficiently ignore bananas that cannot be consumed within the given deadline.

The complexity of the provided model solution remains manageable yet instructive. Insert and delete operations run in average-case complexity of $O(\log n)$. Query operations may reach $O(n)$ in the worst case due to linear traversal, though in practice, early termination significantly reduces traversal time and demonstrates how algorithmic efficiency affects real-world performance.

6 Conclusion:

Treaps offer a neat way to keep binary search trees balanced without the headache of complex balancing rules. By mixing the order of a binary search tree with the heap property based on random priorities, they manage to keep operations like insert, delete, split, and merge running smoothly in expected $O(\log n)$ time. Instead of juggling intricate cases like in AVL or Red-Black trees, treaps use simple rotations to maintain balance. In the realm of caching, particularly in implementing Least Frequently Used (LFU) caches, treaps have been utilized to maintain a balance between access frequency and recency. The randomness in assigning priorities ensures that the tree doesn't become lopsided, even if the keys are added in a sorted order, acting as a safeguard against worst-case scenarios.