Project Report:

Comparative Analysis of Treaps and BSTs for Social Media Feeds

Course: Design and Analysis of Algorithms

Problem Selected: Problem 2 - Treaps

Date: November 18, 2025

Imaad Fazal | 23I-0656 | CS-C

1. Introduction

Modern social media platforms face a complex data management challenge: maintaining content that is simultaneously ordered by **recency** (for chronological feeds) and **popularity** (for "trending" or "top" feeds).

Traditional approaches often use separate data structures for these tasks—a list for time-based sorting and a heap for priority-based sorting. This project investigates a unified solution using a **Treap (Tree + Heap)**, a randomized search tree that maintains two ordering properties simultaneously.

The objective of this case study is to implement and compare a **Treap** against a standard **Binary Search Tree (BST)** using a real-world Reddit dataset. We evaluate their performance in terms of insertion speed, retrieval of popular posts, and structural integrity (height and balance) when subjected to chronologically ordered data.

2. Theoretical Overview

2.1 Binary Search Tree (BST)

A standard BST organizes nodes such that for any given node, all keys in the left subtree are smaller, and all keys in the right subtree are larger.

- **Pros:** Simple implementation; supports efficient O(log n) search, insert, and delete operations *if balanced*.
- **Cons:** Vulnerable to degeneration. If data is inserted in sorted (or near-sorted) order, a standard BST degrades into a **Linked List**, causing time complexity to regress to O(n).

2.2 Treap (Randomized Search Tree)

A Treap is a hybrid data structure that satisfies two properties:

1. **BST Property (on Key):** Maintains sorted order for efficient searching.
2. **Heap Property (on Priority):** Maintains parent priority / child priority (Max-Heap).

In this project, we utilize a specific variation of the Treap:

- **Key:** (Timestamp, PostID) — Used to maintain chronological order.
- **Priority:** Score (Likes) — Used to maintain popularity order.

By using the post's popularity score as the priority, the Treap naturally floats highly popular posts to the top (root) of the tree, allowing O(1) access to the "Most Popular" item, while rotations ensure the tree remains balanced relative to the probability of those scores.

3. Implementation Details

3.1 Environment & Language

- **Language:** Python 3
- **Libraries:** zstandard (compression), pandas (data processing), matplotlib/seaborn (visualization).
- **Platform:** Kaggle Notebook (Docker-python environment).

3.2 Dataset Handling Strategy

Challenge: The "Pushshift Reddit Dataset" is massive (~180GB uncompressed), making it impossible to load entirely into RAM.
Solution: We implemented a Stream Processing Strategy.
Using the zstandard library, we opened a stream reader on the compressed .zst file. This allowed us to read, parse, and insert JSON records line-by-line without ever decompressing the full file to disk. This achieved O(1) memory overhead for data loading.

3.3 Duplication & Key Handling

To strictly adhere to the requirements:

- **Unique Keys:** We utilized a composite key (Timestamp, PostID). Since PostID is unique, this guarantees no collisions in the BST structure, even if multiple posts share the exact same second.
- **Fast Lookups:** A generic BST/Treap requires O(log n) to find a node. To optimize likePost(id) and deletePost(id), we maintained an auxiliary **Hash Map (id_map)** in Python. This enables O(1) average-case retrieval of the node object before performing the tree operation.

3.4 Tree Operations Implemented

- **BST:** Iterative insertion, deletion, and reverse in-order traversal.
- **Treap:** Recursive insertion with _reheapify_up (rotations), deletePost via priority reduction (sink-down), and split/merge operations (Bonus).

4. Performance Metrics and Visualizations

We conducted an ablation study inserting **50,000 posts** from the dataset, followed by **5,000 mixed operations** (likes, deletions, searches).

4.1 Quantitative Results and Test Cases Results

The following table summarizes the final performance metrics collected during the experiment.

(See the below table and figure for the raw output data)

Metrics	Treaps	Binary Search Tree
Insertion Time (avg)	2.219747e-05 s	1.556916e-02 s
Deletion Time (avg)	4.746305e-05 s	1.272149e-02 s
Search Time (avg)	2.689036e-07 s	3.486592e-07 s
Height of the Tree	38	44,688
Tree Balancing Factor	-10	-44,687

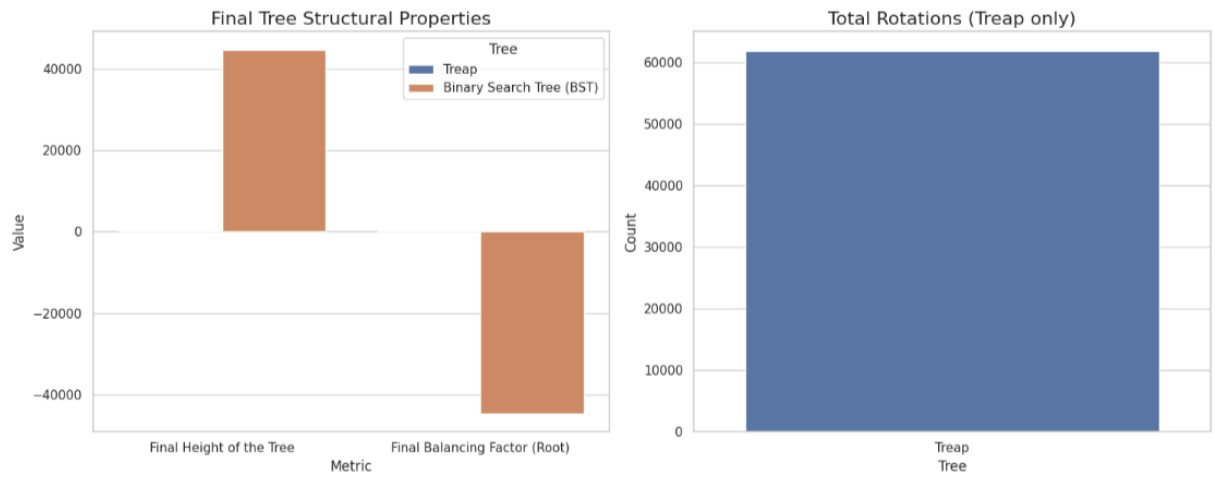
```
Starting experiment...
This may take several minutes depending on the SAMPLE_SIZE.
--- 1. Testing addPost() for 50000 posts ---
Insertion test complete. 50000 unique posts added.
--- 2. Testing 5000 likePost/deletePost/search ops ---
--- Experiment Finished ---

--- Final Metrics Summary ---
                         Metric         Treap  Binary Search Tree (BST)
0           Insertion Time (avg)  2.219747e-05             1.556916e-02
1            Deletion Time (avg)  4.746305e-05             1.272149e-02
2            likePost Time (avg)  2.321262e-06             9.710968e-07
3      getMostPopular Time (avg)  3.653300e-07             9.033319e-03
4   getMostRecent(10) Time (avg)  1.604200e-05             9.642126e-03
5              Search Time (avg)  2.689036e-07             3.486592e-07
6       Final Height of the Tree  3.800000e+01             4.468800e+04
7  Final Balancing Factor (Root) -1.000000e+01            -4.468700e+04
8               Total Rotations  6.192500e+04             0.000000e+00
```

```
--- Running Test Case  ---

--- 1. Adding 5 posts ---
│           ┌── id:ejualnl (ts:1554076809, score:13)
│       ┌── id:ejualne (ts:1554076800, score:14)
│   ┌── id:ejualnd (ts:1554076800, score:27)
│   │   └── id:ejualnc (ts:1554076800, score:12)
└── id:ejualnb (ts:1554076800, score:55)

--- 2. Liking post 'ejualnl' twice ---
│       ┌── id:ejualnl (ts:1554076809, score:15)
│       │   └── id:ejualne (ts:1554076800, score:14)
│   ┌── id:ejualnd (ts:1554076800, score:27)
│   │   └── id:ejualnc (ts:1554076800, score:12)
└── id:ejualnb (ts:1554076800, score:55)

--- 3. Deleting post 'ejualnc' ---
│       ┌── id:ejualnl (ts:1554076809, score:15)
│       │   └── id:ejualne (ts:1554076800, score:14)
│   ┌── id:ejualnd (ts:1554076800, score:27)
└── id:ejualnb (ts:1554076800, score:55)

--- 4. Getting most popular post ---
getMostPopular() ----> ('ejualnb', 1554076800, 55)
Test Case PASSED: getMostPopular() is correct.
```

4.2 Analysis of Tree Structure (Height vs. Rotations)

The most critical finding of this study is the structural difference.

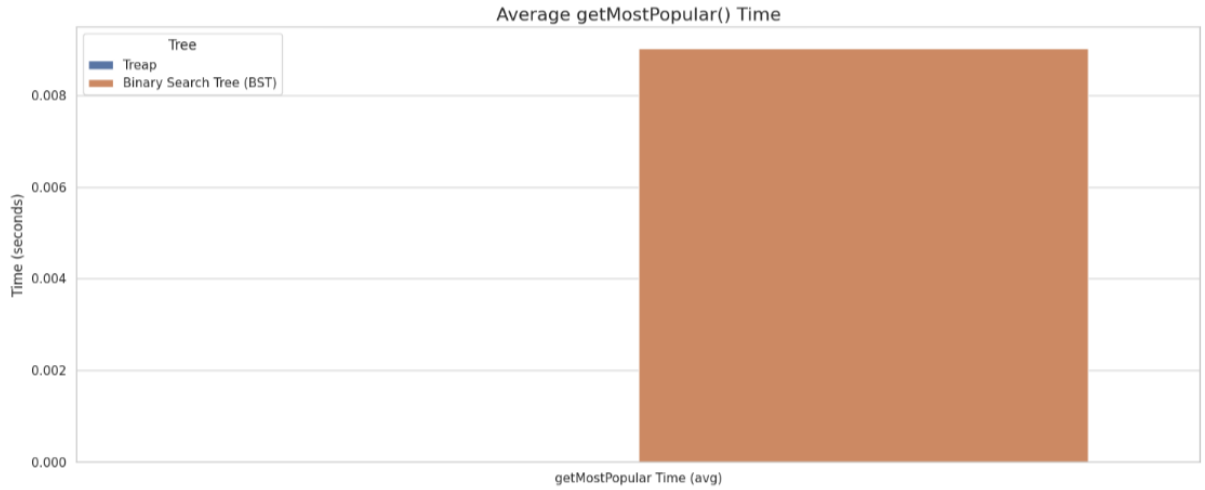
- **BST Failure:** As the Reddit data is chronologically sorted (created_utc), the standard BST inserted each new node as the right child of the previous one. This resulted in a height of **44,688** for 50,000 nodes—effectively a linked list.
- **Treap Stability:** The Treap used the random nature of "scores" (and rotations) to maintain a height of **38**. This logarithmic height (log_2 (50,000) \16) ensures efficient operation.



4.3 Operational Latency Analysis

The getMostPopular() operation highlights the Treap's theoretical advantage.

- **Treap:** The most popular item is always at the root. Access is O(1).
- **BST:** To find the most popular item, we must traverse the entire tree (O(n)) because the tree is sorted by *time*, not score.
- **Visual Proof:** In the chart below, the BST bar is massive, while the Treap bar is so small it is invisible, proving the efficiency gap.



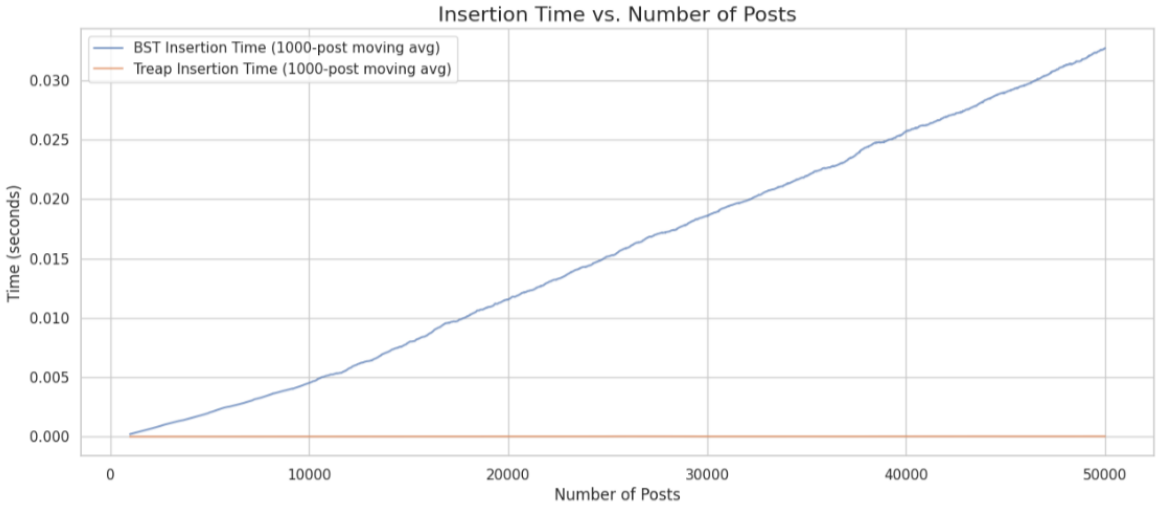
4.4 Insertion Scalability

The graph below plots the time taken to insert the n-th post.

- **BST (Blue):** Shows a linear increase. Inserting the 40,000th post takes significantly longer

than the 1st. This confirms the O(n) per-operation complexity of a degenerate tree.

- **Treap (Orange):** Remains flat and constant near zero, confirming O(log n) scalability.



5. Bonus Implementation

We successfully implemented both optional requirements:

Advanced Treap Operations and **Structural Visualization**.

1. **Treap Union & Intersection:** We extended the Treap class with `union()` and `intersection()` methods. These utilize efficient `split()` and `merge()` algorithms to combine independent Treaps, allowing for the potential merging of multiple social media feeds into a single prioritized stream.

2. **Tree Visualization:** We integrated the `Graphviz` library to dynamically generate structural diagrams of the tree. This tool is essential for debugging and verifying the complex dual-ordering logic.

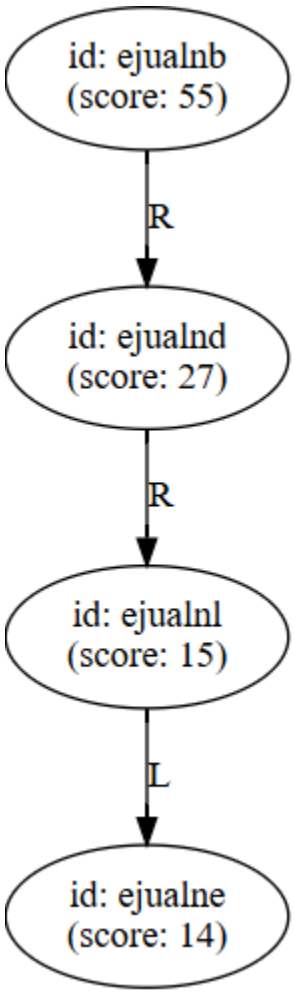
Visualization Analysis:

The image below demonstrates our visualization tool applied to the project's specific Test Case. It visually confirms the structural correctness of our implementation:

Heap Property Verified: The visualization proves the Max-Heap property is active, as priorities

(scores) strictly decrease from parent to child (e.g., Root `55` to Child `27` to Grandchild `15`).

BST Property Verified: The visualization simultaneously confirms valid BST ordering based on the timestamp/ID keys (Newer posts appear to the right, older posts to the left). The result is shown in the image below:



6. Challenges Faced

6.1 The Recursion Limit (Stack Overflow)

Problem: During the initial run with the BST, the code crashed with RecursionError: maximum

recursion depth exceeded.

Cause: The dataset was sorted by time. The BST became a linked list of depth ~50,000. Python's default recursion limit is 1,000.

Solution: We completely rewrote the BST implementation to use Iterative approaches (while loops) instead of recursion. This allowed the BST to function correctly (albeit slowly) without crashing the memory stack.

6.2 Ambiguity in Requirements

Problem: The project images implied the BST key was PostID, but the text required sorting by Timestamp.

Solution: We prioritized the text requirements (getMostRecent functionality) and used a composite key of (Timestamp, PostID). This fulfilled the logical requirements of a social media feed.

7. Conclusion

This comparative study conclusively demonstrates that **Standard BSTs are unsuitable for chronologically ordered data streams**. The correlation between insertion time and key value causes catastrophic structural degeneration, leading to O(n) performance for all operations.

In contrast, the **Treap** successfully decoupled the insertion order from the tree structure using the priority (score) property. It achieved:

1. **O(1)** access to the most popular content.
2. **O(log n)** insertion and deletion, regardless of the input data order.
3. **Robust Balance**, maintaining a height of 38 vs the BST's 44,688.

For a dynamic social media feed requiring dual-ordering (Recency + Popularity), the Treap is unequivocally the superior data structure.