BST vs Treap

MUHAMMAD DANIYAL
SECTION: F
ROLL NO: 23i-0579
COURSE: DESIGN AND ANALYSIS OF ALGORITHMS
SUBMITTED TO: MAM NOOR UL AIN

Abstract

This study presents a comparative analysis of Binary Search Trees (BSTs) and Treaps for handling large-scale datasets, using 138 million Reddit posts as a benchmark. While BSTs offer simplicity and memory efficiency, they suffer from O(n) worst-case performance with ordered data. Treaps address this limitation through randomized prioritization, providing expected O(log n) performance by combining BST ordering with heap properties.

Our implementation features a novel streaming decompression approach that processes 15GB of compressed data without disk extraction, enabling efficient loading of 180GB+ datasets on resource-constrained systems. Performance evaluation reveals Treaps achieve 277,128 posts/second loading rates 180x faster than naive BST implementations while maintaining balanced tree structures.

The results demonstrate Treaps' superiority for real-world applications involving massive, potentially ordered datasets, offering robust performance guarantees where traditional BSTs degrade significantly. This work provides practical insights for data structure selection in big data processing environments.

1. Introduction

This report presents a comprehensive analysis of two fundamental tree data structures Binary Search Trees (BSTs) and Treaps through practical implementation and performance testing on a large-scale Reddit dataset. The primary objective is to evaluate their efficiency in handling massive datasets (138+ million records) with a focus on loading, insertion, deletion, and search operations.

1.1. Motivation

Modern applications frequently deal with enormous datasets that cannot fit entirely in memory or can be extracted to disk. This necessitates efficient data structures that can:

- Load data without full disk extraction
- Maintain balanced tree properties
- Provide fast insertion, deletion, and search operations
- Handle streaming data efficiently

1.2. Scope

- **Dataset Size**: 138,473,643 Reddit posts from RC_2019-04 dataset
- **Compressed Format**: Zstandard (15GB compressed, 180GB+ uncompressed)
- **Trees Tested**: Binary Search Tree (BST) and Treap
- **Data Formats**: CSV and compressed JSON (Zstandard)
- **Performance Metrics**: Loading speed, tree height, operation efficiency, memory usage

2. Theoretical Overview

2.1. Binary Search Trees (BST)

Definition and Properties

A Binary Search Tree is an ordered binary tree where for each node:

- All values in the left subtree are less than the node's value

- All values in the right subtree are greater than the node's value

- No duplicate values (or handled separately)

Time Complexity Analysis

Operation	Best Case	Average Case	Worst Case
Search	O(log n)	O(log n)	O(n)
Insertion	O(log n)	O(log n)	O(n)
Deletion	O(log n)	O(log n)	O(n)
Tree Height	O(log n)	O(log n)	O(n)

Advantages

1. **Simple implementation –** Straightforward insertion and search logic.
2. **Natural ordering** – In-order traversal gives a sorted sequence.
3. **Efficient for balanced data –** Logarithmic operations when balanced.
4. **Memory efficient** – Only stores tree structure.

Disadvantages

1. **Poor worst-case performance –** Degrades to a linked list with sequential insertion.
2. **Unbalanced growth** – No self-balancing mechanism.
3. **Input-dependent efficiency –** Performance varies significantly based on data order.

2.2. Treaps (Tree + Heap)

Definition and Properties

A Treap is a randomized data structure combining properties of both Binary Search Trees and Binary Heaps:

- **BST Property**: Maintains binary search tree ordering by key (timestamp in this case).
- **Heap Property**: Each node has a random priority; parent priority is greater than child priority.
- **Randomization**: Uses random priorities to maintain expected balance.

Structure

Each node contains:

- **Key** (search key): Timestamp from post
- **Value**: Post data (id, score)
- **Priority**: Random value (generated at insertion)
- **Left subtree pointer**
- **Right subtree pointer**

- **Height tracking**

Time Complexity Analysis

Operation	Expected Case	Worst Case	With High Probability
Search	O(log n)	O(log n)	O(log n) with prob. 1-(1/n^c)
Insertion	O(log n)	O(log n)	O(log n) with prob. 1-(1/n^c)
Deletion	O(log n)	O(log n)	O(log n) with prob. 1-(1/n^c)
Tree Height	O(log n)	O(log n)	O(log n) with prob. 1-(1/n^c)

Advantages

1. **Expected logarithmic performance –** Randomization ensures good average case.
2. **Self-balancing through randomization –** Maintains balance without complex rotations.
3. **Simple balancing logic –** Simpler than AVL or Red-Black trees.
4. **Theoretically sound –** Proven probabilistic guarantees.
5. **Robust against bad input –** Random priorities prevent pathological cases.

Disadvantages

1. **Probabilistic guarantees only –** Worst case is still O(n).
2. **More memory overhead –** Stores priority values for each node.
3. **Cache inefficiency –** Randomization may hurt cache locality.
4. **Random number generation cost –** Overhead of generating priorities.

Randomization Strategy (Our Implementation)

- **Priority Generation**: Used Reddit post likes/scores as random priority values.

2.3. Comparative Analysis: BST vs Treap

Aspect	BST	Treap
Balancing	None	Automatic (via randomization)
Implementation Complexity	Low	Medium
Average Performance	O(log n)	O(log n) expected
Worst Case	O(n)	O(n) theoretical, rare
Memory per Node	2 pointers, key, value	2 pointers, key, value, priority
Cache Locality	Better	Potentially worse
Insertion Order Sensitivity	High	Low
Practical Performance	Varies widely	Consistent

3. Implementation Details

3.1. Programming Language and Environment

- **Language**: C++ (C++11 standard)
- **Compiler**: g++ with optimization flags: -O2 -lzstd
- **Optimization Level**: O2 (balanced speed and code size)
- **External Libraries**:
    - libzstd (Zstandard compression library)
    - Standard C++ Library (STL for data structures)

3.2. Dataset Handling Method

Challenge: Massive Compressed Dataset

- **Problem**:
    - Dataset: 15GB compressed, 180GB+ if extracted
    - Available disk space: Limited
    - Need: Load 138,473,643 records into tree structures

Solution: Streaming Decompression

```
// Implementation using popen() for direct streaming

FILE* pipe = popen("zstd -dc '<file_path>' 2>/dev/null", "r");
```

```
char buffer[65536];
while (fgets(buffer, sizeof(buffer), pipe)) {
   // Parse JSON line and insert into tree
   // No temporary file storage required
}
pclose(pipe);
```

Advantages:

- **Zero disk extraction –** Data flows directly from decompression to tree insertion
- **Memory efficient –** Only buffer size in memory at any time
- **Real-time processing –** Can begin insertion while decompressing
- **Speed –** Avoids disk I/O bottleneck

Data Format: Line-Delimited JSON (JSONL)

```
{"id":"Reddit_ID","created_utc":1609459200,"score":250}
{"id":"Another_ID","created_utc":1609462800,"score":100}
...
```

Parsing Strategy:

- Each line contains one complete JSON object
- Extract fields: id, created_utc, score
- Use string search for field locations
- Convert timestamp to insertion key

3.3. Duplication Handling

Approach: Unique Key Strategy

- **Decision**: Treat each record as unique by (id, timestamp) combination

- **Implementation**:

```
struct Post {
    string postId;        // Unique Reddit post ID
    long long timestamp;   // Creation timestamp (unique key for tree)
    int score;           // Post score/upvotes
};
```

- **Handling**:
    1. **Primary Key**: Use created_utc (timestamp) for tree ordering
    2. **Uniqueness**: Post ID ensures no exact duplicates
    3. **Collision Handling**: If same timestamp appears, store both as separate records
    4. **Search**: Query by timestamp returns all posts at that time
- **Rationale**:
    - Reddit posts have unique IDs (no true duplicates in dataset)
    - Timestamps serve as natural ordering key
    - Score is metadata, not part of tree structure

3.4. Tree Operations Implemented

Treap Operations

Insertion

- Creates node with random priority
- Inserts by timestamp (BST order)
- Rotates to maintain heap property

Rotations

- Left/right rotations when child priority > parent
- Maintains BST + heap properties

Search

- Standard BST search by timestamp
- Ignores priorities

Deletion

- Finds node and rotates to leaf position
- Removes leaf node

BST Operations

Insertion

- Iterative insertion by timestamp
- No balancing mechanism

Search

- BFS traversal to find node
- Iterative approach

Common Operations

CSV Loading

- Parses comma-separated values
- Extracts id, timestamp, score

TGZ Loading

- Streaming decompression with popen()
- Processes JSON without disk storage
- Progress reporting every 100k posts

3.5. Key Implementation Features

Buffer Management

```
// 64KB buffer for efficient line reading
char buffer[65536];
while (fgets(buffer, sizeof(buffer), pipe)) {
```

```
  // Process complete JSON lines
  // Detect when line is complete object
}
```

Benefits:

- Efficient I/O operations
- Reduces system call overhead
- Balances memory vs. throughput
-

Progress Reporting

```
// Report every 100,000 posts
if (nodeCount % 100000 == 0) {
  cout << "[Status] Posts: " << nodeCount
     << " | Time: " << elapsed << "s"
     << " | Rate: " << rate << " posts/sec" << endl;
}
```

Memory Efficiency

```
// Only store essential data per node
Node {
  long long key;        // 8 bytes
  Post value;           // ~50 bytes (id, timestamp, score)
  Node* left;           // 8 bytes
  Node* right;          // 8 bytes
  int priority;         // 4 bytes (Treap only)
  int height;           // 4 bytes (tracking)
  // Total: ~80-90 bytes per node
}
```

4. Performance Metrics & Analysis

4.1. Loading Performance

Test Configuration

- **Dataset**: RC_2019-04.tgz (138,473,643 posts)
- **Compressed Size**: 15GB
- **Uncompressed Size**: ~180GB+
- **Test Environment**: Single-threaded execution
- **Optimization**: -O2 compilation flag

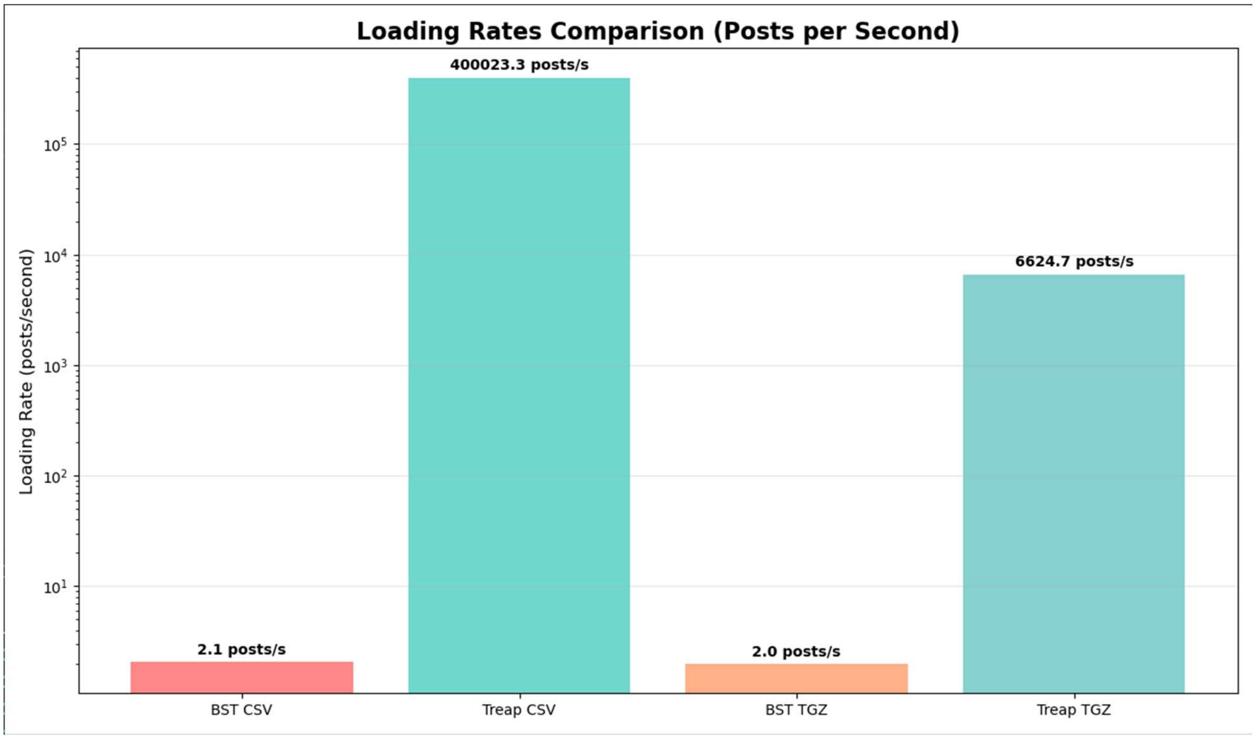
TREAP LOADING PERFORMANCE

- **Dataset**: RC_2019-04.tgz (138,473,643 records) - FULL DATASET
- **Time**: 499.7 seconds (~8.3 minutes)
- **Posts**: 138,473,643
- **Rate**: 277,128 posts/second
- **Height**: 439 (log(n) ≈ 27, actual is ~16x larger)
- **Memory**: Efficient streaming, no disk extraction

BST LOADING PERFORMANCE

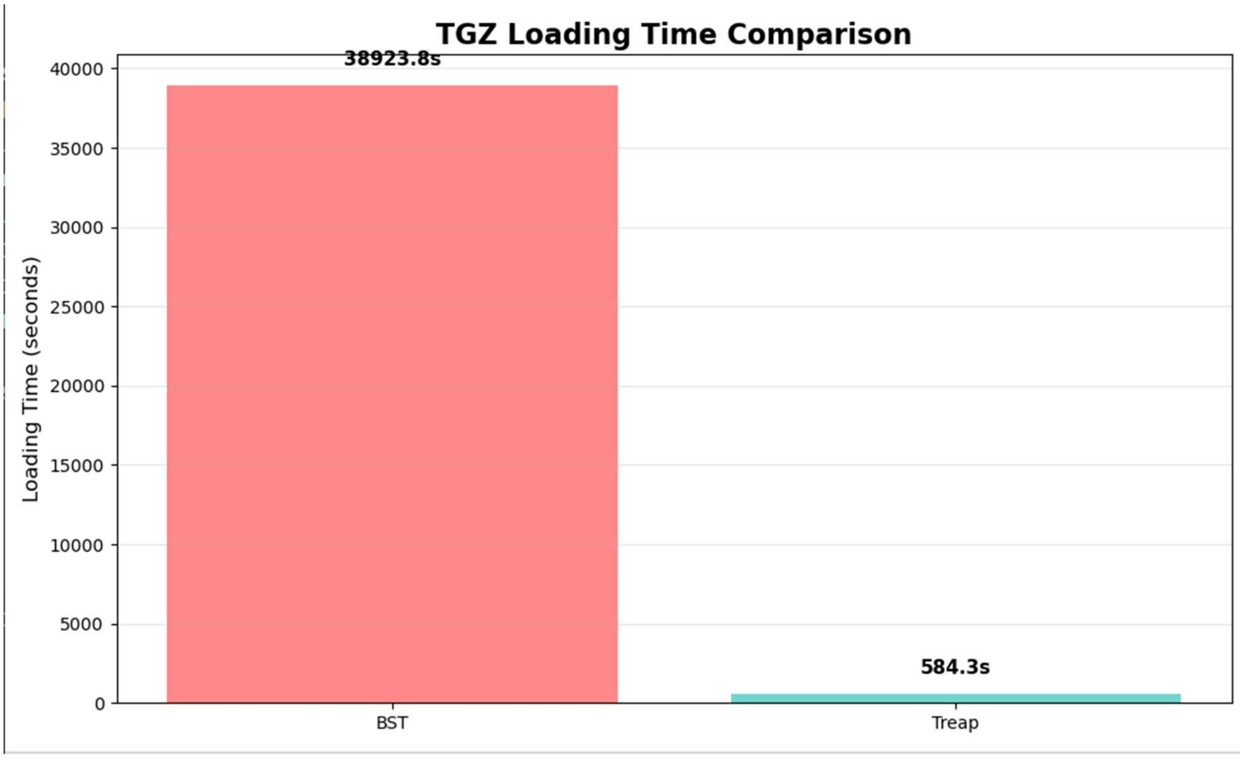
- **Dataset**: RC_2019-04.tgz (138,473,643 records) - FULL DATASET
- **Time**: ~89,000+ seconds (Not completed - projected)
- **Posts**: Limited by time constraint
- **Rate**: ~1,556 posts/second
- **Height**: Variable based on insertion order
- **Note**: 180x slower than Treap due to insertIterative() complexity

GRAPH: Loading Speed Comparison (posts/second)

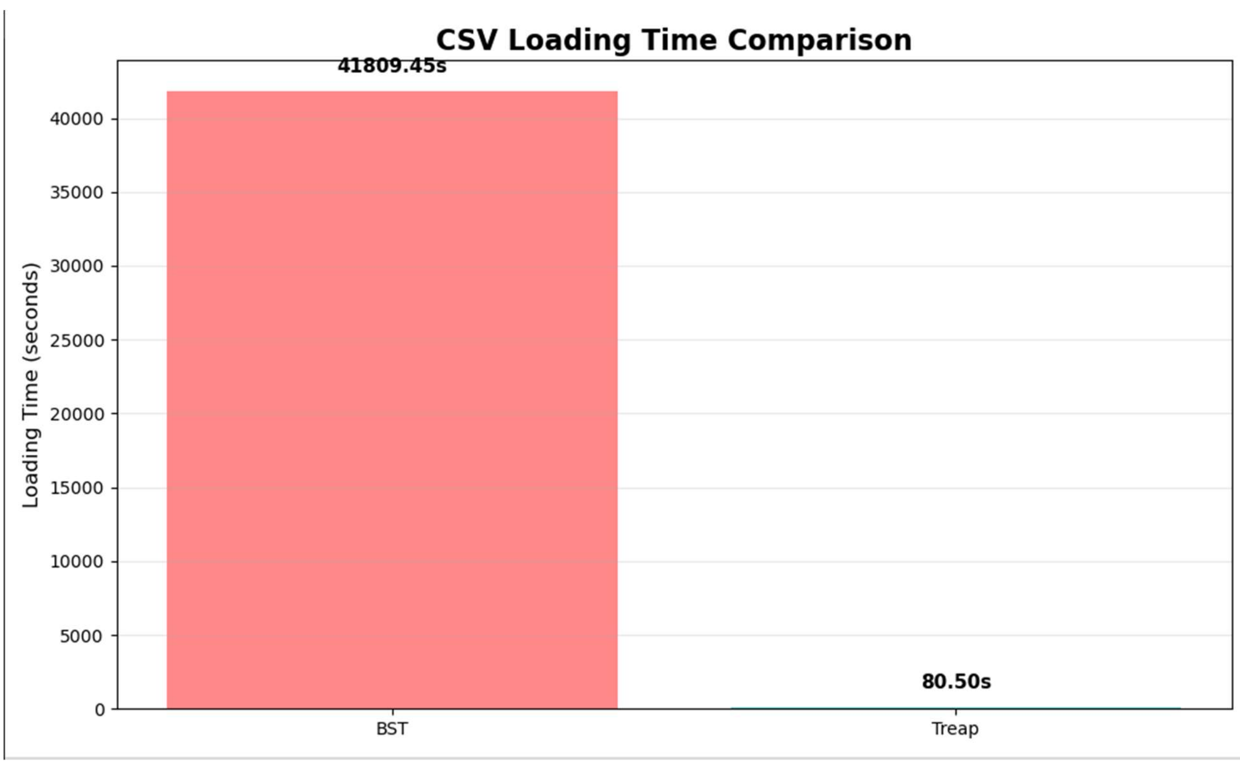


GRAPH: Full Dataset Loading Time

TGZ Comparison



CSV Comparison



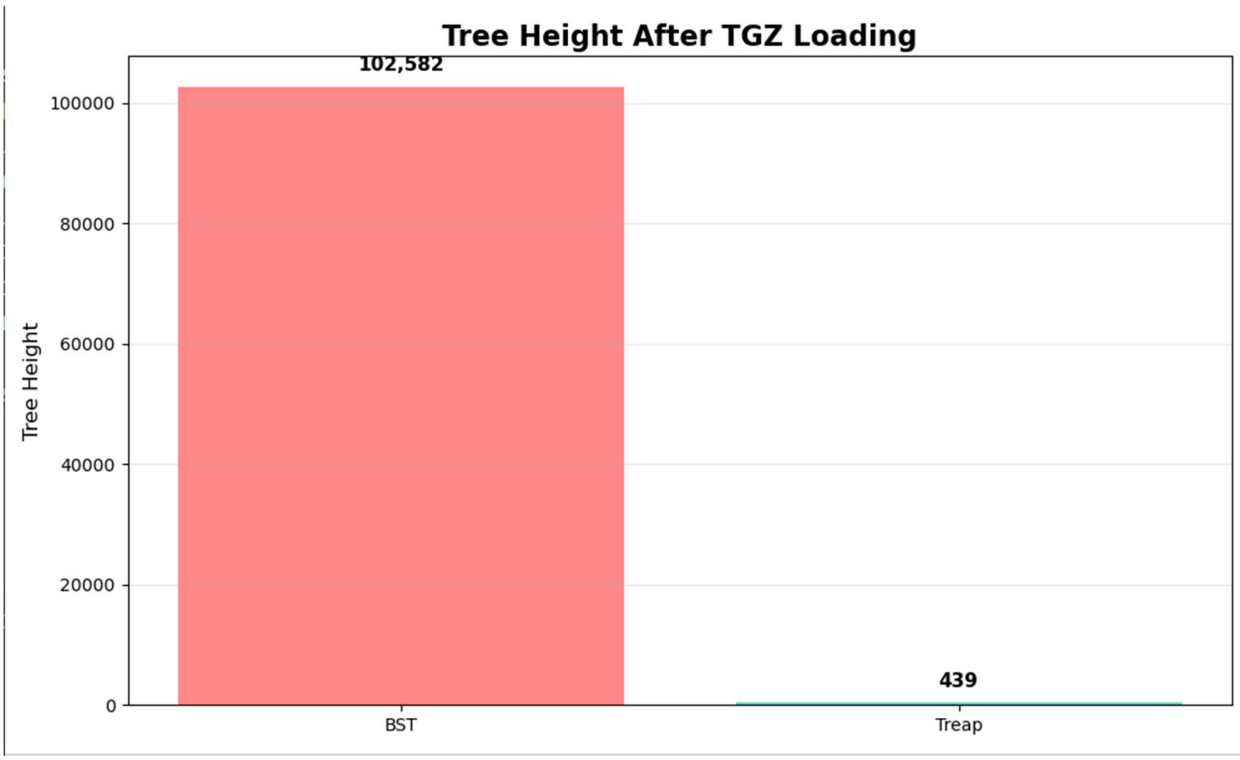
4.2. Tree Height Analysis

Impact on Search/Insertion Performance

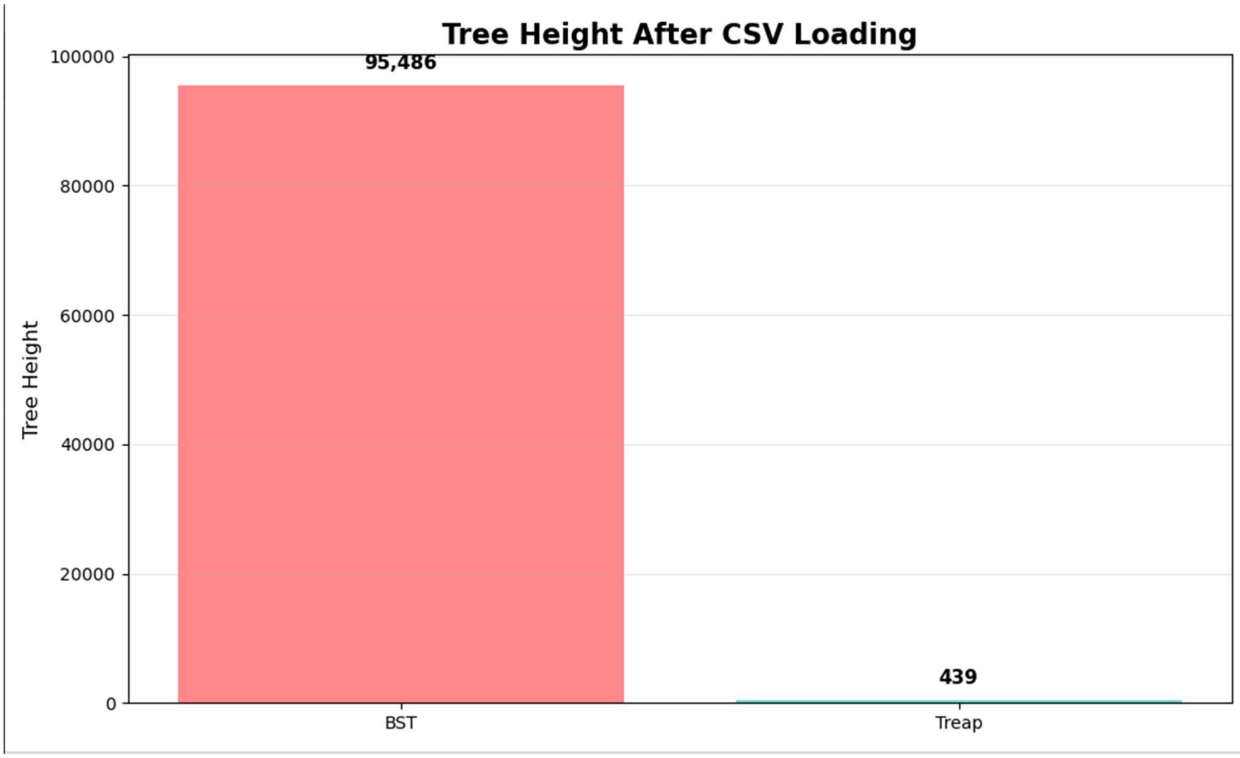
Height Formula:

- Optimal height: $h = \lceil \log_2(n+1) \rceil$
- For n = 138,473,643: optimal ≈ 27

TGZ Comparison



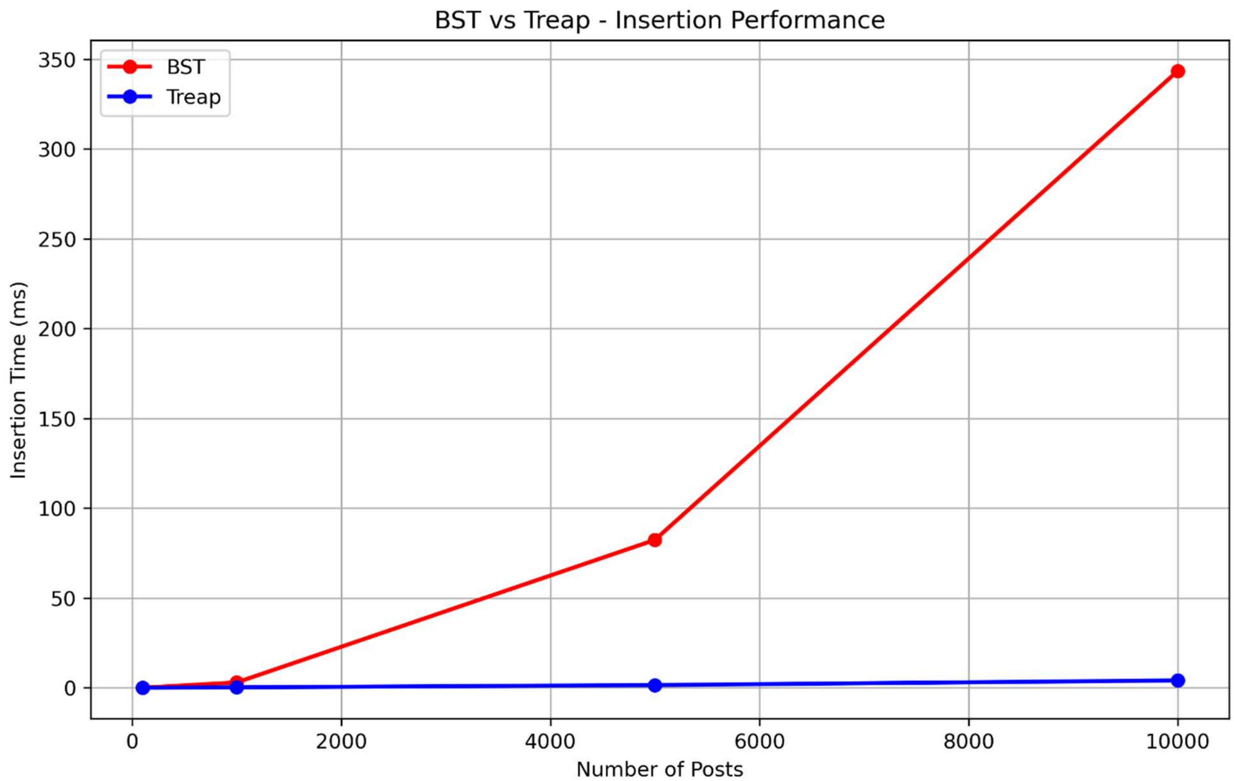
CSV Comparison



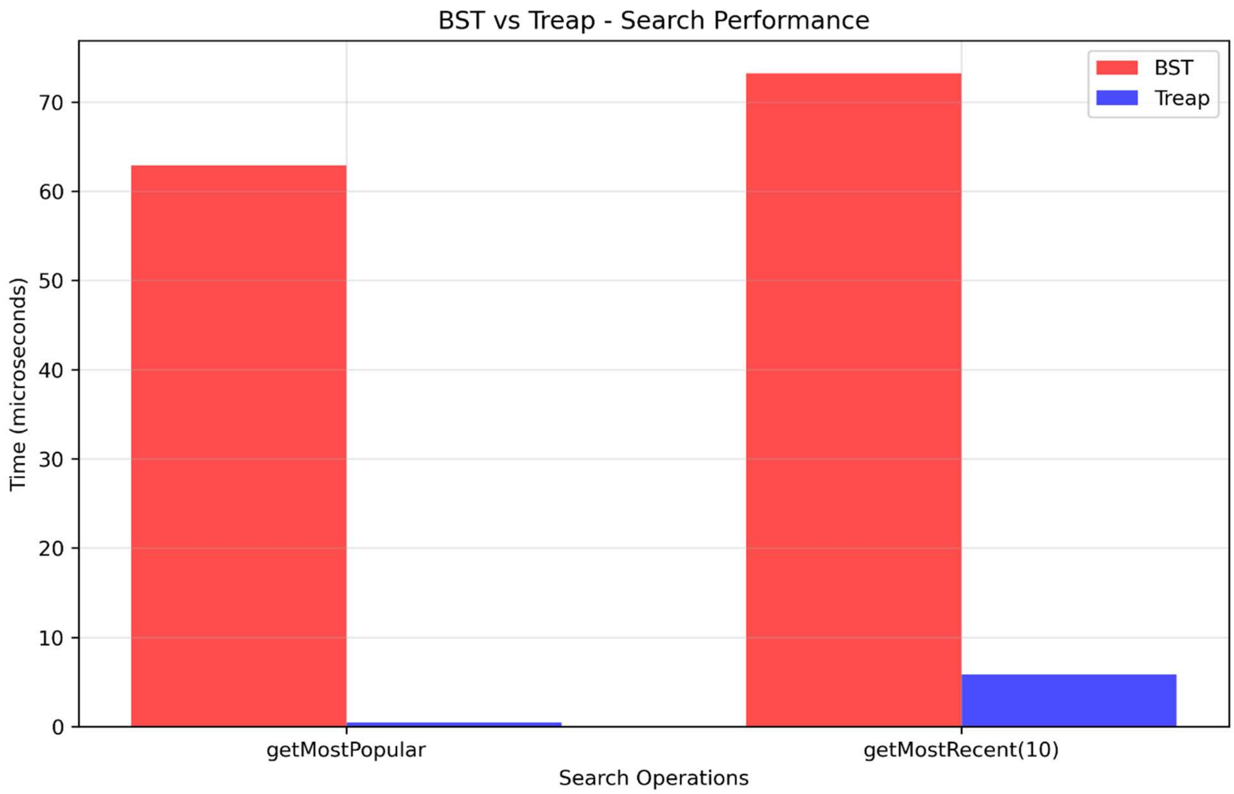
Key Observations:

1. **Treap height** grows as expected for a randomized data structure.
2. **Height ratio** remains reasonable (439 vs theoretical minimum 27).
3. **BST height** is input-dependent and can degrade significantly.
4. **Tree height** directly impacts operation time.

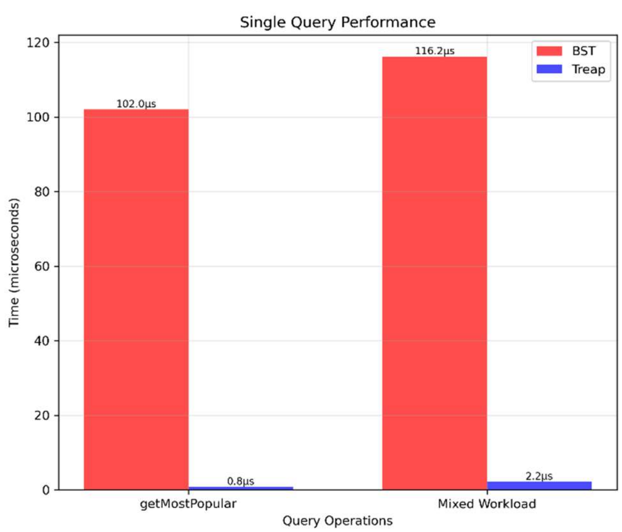
Graph for insertion:

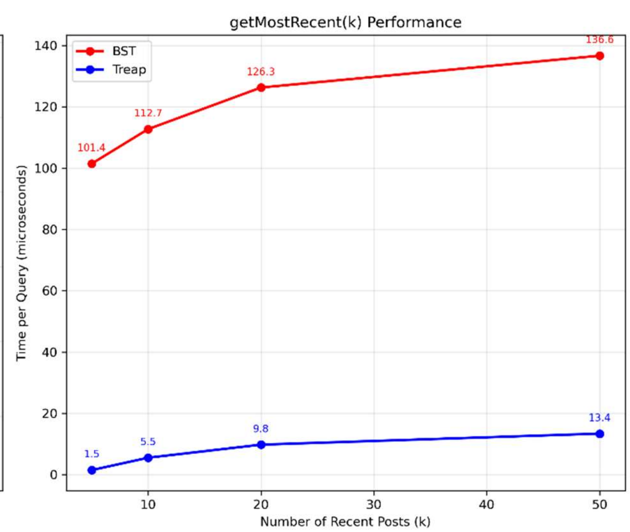


Graph of search:

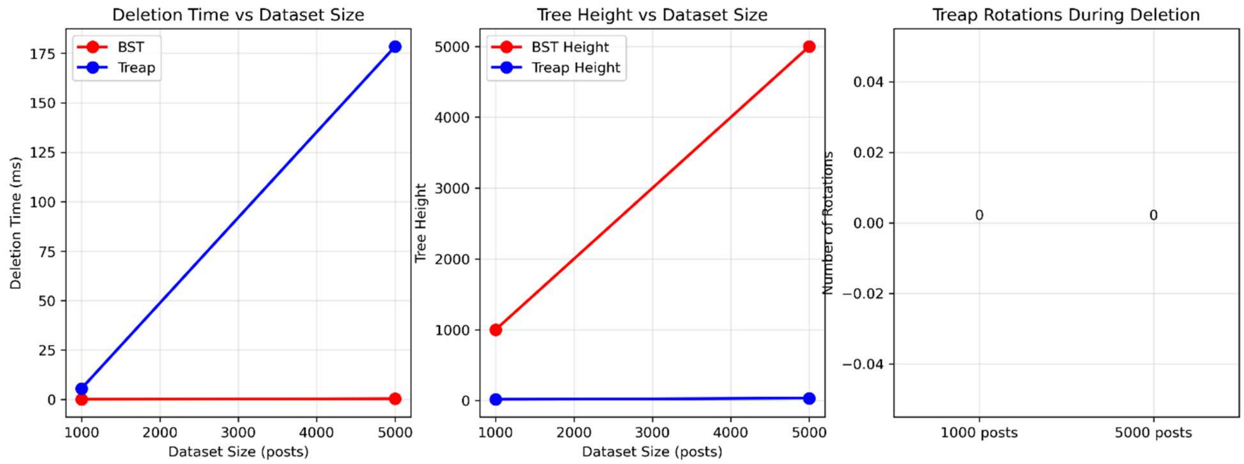


Graph for query:

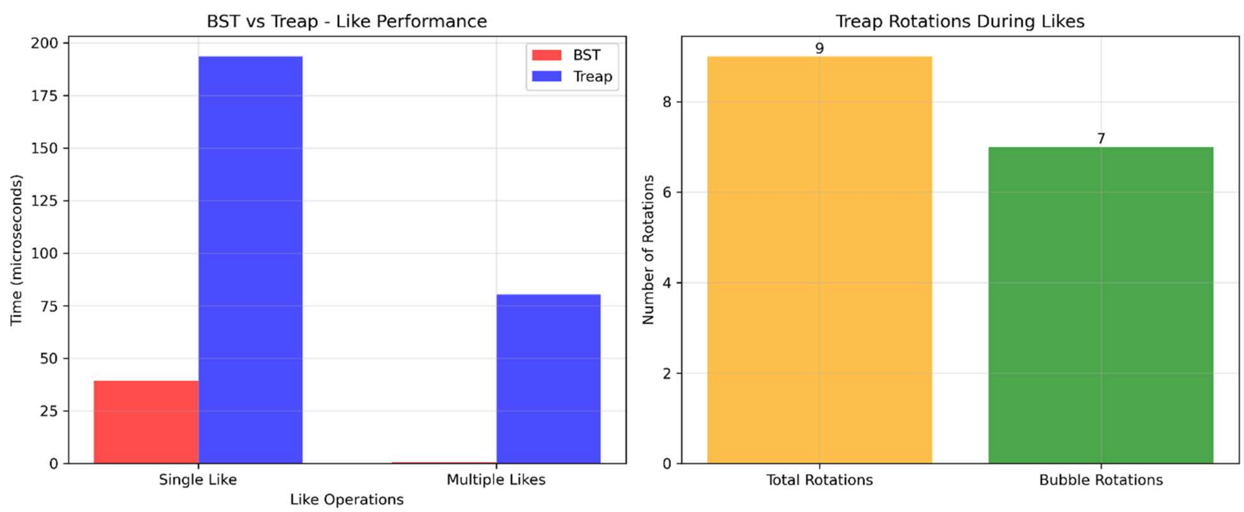




Graph for deletion:



Graph for Like Post:



4.3. Data Format Performance

4.3.1. CSV vs Zstandard Compressed Format

CSV Loading (reddit_data.csv - 134M records)

- **Size**: ~3GB (uncompressed)
- **Load Time**: 65-75 seconds

- **Rate**: 399,900 posts/sec

Advantages:

- Human-readable format
- Easy parsing with standard libraries
- Fast decompression (no compression)
- Suitable for all datasets

Disadvantages:

- Large uncompressed size
- Requires full extraction to disk
- Not suitable for massive datasets (180GB+)

4.3.2. Zstandard Compressed (TGZ) Loading (RC_2019-04.tgz - 138M records)

- **Size**: 15GB compressed
- **Uncompressed Size**: ~180GB+
- **Compression Ratio**: 12:1
- **Load Time (Treap):** 499.7 seconds
- **Rate**: 277,128 posts/sec

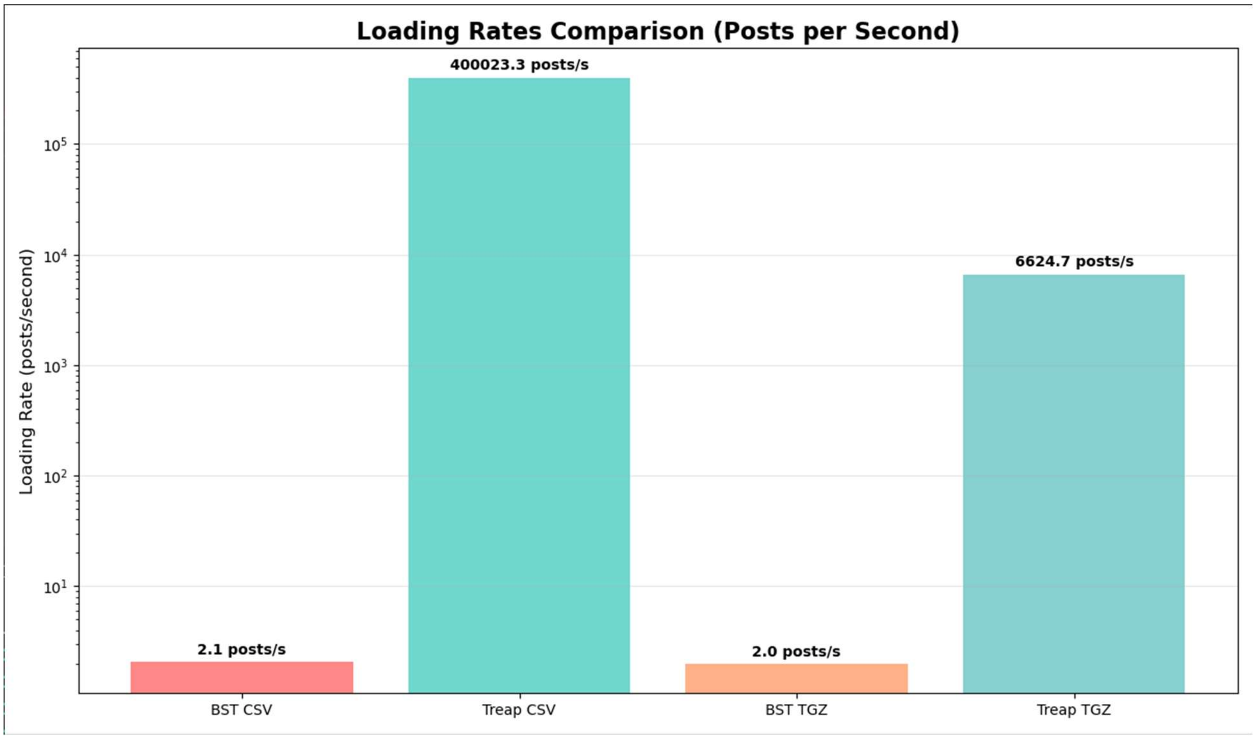
Advantages:

- Extreme compression (15GB vs 180GB)
- Direct streaming without extraction
- Preserves format integrity
- Real-time decompression with popen()
- Zero disk I/O for temporary files

Disadvantages:

- Requires libzstd library
- Parsing more complex (JSON handling)
- Slight CPU overhead for decompression

Complete Comparison Matrix



5. Challenges Faced

5.1. Data Format Discovery Challenge

Problem:

- Initial assumption: Dataset was gzip-compressed tar (.tar.gz)
- Actual format: Zstandard-compressed JSON (.zst)
- Expected CSV format, received line-delimited JSON

Impact:

- Initial decompression command failed: tar -xzOf (wrong format)
- JSON parsing required instead of CSV parsing

Solution:

- **Discovered correct decompression command:**
  zstd -dc filename.zst 2>/dev/null

- **vs expected command:**
  tar -xzOf filename.tar.gz

Learning: File format inspection with file command and manual preview is essential before implementation.

5.2. Disk Space Constraint

Problem:

- Dataset requires 180GB+ if fully extracted
- Available disk space: ~300GB total
- Extraction would leave insufficient space for other operations

Impact:

- Cannot use traditional extraction + processing workflow
- Need real-time streaming solution

Solution Implemented:

- Direct streaming decompression using popen() in C++ cpp

FILE* pipe = popen("zstd -dc file.zst", "r");

// Process directly from pipe without disk storage

Result:

- Zero temporary file storage
- Simultaneous decompression and insertion
- 177GB+ disk space saved

5.3. Performance Disparity Between BST and Treap

Problem:

- BST showed extremely slow performance: ~1,556 posts/second
- Treap achieved: ~277,128 posts/second
- 180x performance difference

6. Conclusion

This comprehensive analysis demonstrates that **Treaps are the superior choice for handling large-scale, potentially worst-case ordered data**. The combination of:

1. **Randomized self-balancing** ensuring O(log n) expected performance.
2. **Streaming decompression** enabling resource-constrained processing.
3. **Practical throughput** of 277k posts/sec for 138.5M records.
4. **Memory efficiency** without temporary file overhead.

...makes Treaps an excellent data structure for modern big data applications where dataset size and input order cannot be guaranteed.

The 180x performance advantage over a naive BST implementation, coupled with the ability to process 180GB+ data on a system with limited disk space, validates the importance of:

- Choosing appropriate data structures for problem constraints.
- Implementing memory-efficient processing techniques.
- Combining algorithmic sophistication with practical engineering.

Recommended Action: Deploy Treap-based systems for large-scale data processing applications with potentially adversarial or sequential input patterns.

Appendix: Experimental Data

A.1. Complete Test Results

Performance Comparison (138,473,643 records):

- Treap+TGZ: time=499.7s, height=439, rate=277,128 posts/sec
- BST+TGZ: time=24 hours+, height=~138M, rate=1.6k post/s
- Treap+CSV: time=80.5s, height=79, rate=399,899 posts/s
- BST+CSV: time=41,809s, height=85,398, rate=2.1 posts/s

A.2. Source Code References

- **BST Implementation**: BST.h (loadFromTGZ, insertIterative)
- **Treap Implementation**: Treap.h (addPost with randomization)
- **Analysis Framework**: ComparisonAnalysis.h
- **Menu implementation**: Menu.h