Binary Search Tree (BST) vs. Trie: A Library Cataloging System

Igor Cardoso, Nicole Mills-Dunning, Winston Tsia

1 Objective

In this lab, students will implement both a Binary Search Tree (BST) and a Trie and compare their operation speeds within the context of a real-world scenario: developing a cataloging system for a library.

1.1 Problem Statement

Imagine you are tasked with developing a cataloging system for a large library. The system needs to support operations like adding new books, removing books, and searching for books based on their titles. You are required to decide whether a Binary Search Tree (BST) or a Trie would be more suitable for implementing the dictionary that will store the book titles.

2 Data Structures

2.1 Binary Search Tree (BST)

A Binary Search Tree (BST) is a hierarchical data structure used in computer science and is particularly useful for organizing and storing sorted data efficiently. It is a type of binary tree where each node has at most two children, referred to as the left child and the right child. The structure of a BST follows a specific ordering property:

1. **Binary Tree Structure:** Each node in a BST has, at most, two children. The left child contains elements that are less than or equal to the parent node, and the right child contains elements that are greater than the parent node.
2. **Ordered Data:** The elements (values) stored in a BST are organized in a way that allows for efficient searching, insertion, and deletion operations. The left subtree of any node contains values less than the node's value, and the right subtree contains values greater than the node's value.

BSTs are commonly used for tasks such as searching for specific elements within a sorted dataset and implementing algorithms like binary search. They are also fundamental in various data structures and algorithms, including self-balancing trees like AVL trees and Red-Black trees.

2.2 Trie

A Trie is a tree-like data structure used for efficiently storing and searching a dynamic set of strings or keys, typically associated with text processing and dictionary applications. A Trie is often referred to as a "prefix tree" because it stores data in a way that optimizes prefix-based searching.

Key characteristics of a Trie include:

1. **Tree Structure:** A Trie is a tree-like structure where each node represents a character or part of a string. The root node typically represents an empty string.
2. **Path to Nodes:** The path from the root to any node represents a string or prefix formed by concatenating the characters along the path.
3. **Child Nodes:** Each node in a Trie can have multiple child nodes, each corresponding to a unique character in the alphabet.
4. **End-of-Word Marker:** Nodes in a Trie may have a marker to indicate that a valid word ends at that node. This marker distinguishes between prefixes and complete words.
5. **Efficient Searching:** Trie structures excel at searching for words, prefixes, or substrings efficiently. They can quickly determine the presence or absence of a word in the Trie.

Tries are widely used in text processing applications, such as autocomplete functionality in search engines, spell checkers, and dictionary implementations. They provide fast access to words with common prefixes and are especially valuable for storing and searching large sets of strings. However, Tries can consume memory, especially for large dictionaries with many unique words. To mitigate this, variations like compressed Tries or radix trees are used to optimize space.

3 Operation Speed Comparison

We compared the operation speeds of insertion, deletion, and search for both the BST and the Trie using a list of $10,000$ randomly generated book titles. We conducted experiments to measure the time taken for insertion, deletion, and search operations in both the BST and the Trie. The results are tabulated in `Table 1`.

3.1 O-Notation Cases

In a well-balanced BST, the height of the tree is minimized, ensuring that operations like searching, insertion, and deletion have a time complexity of $O(log\ n)$ on average. However, if the tree becomes unbalanced, these operations can degrade to $O(n)$ in the worst case.

As for the Trie, the time complexity of operations on a Trie is typically $O(N * avg(l))$, where l is the length of the key or string involved in the operation. The best case is $l = 1$, while the worst case is l. However, within time-complexity, this may be reduced to $O(n)$

In the **best** case, BST $O(log\ n)$ outperforms Trie $O(n)$. In the **average** case, the time complexity of BST is $O(log\ n) \leq O(f(T)) \leq O(n)$, meaning it likely performs better then Trie whose best case is $O(n)$. In the **worst** case, they have the same time-complexity.

Table 1: Time Performance Measurements

Trial	Operation	BST (nanoseconds)	Trie (nanoseconds)
1	Insertion	5997800	11000300
	Search	3000000	3000300
	Deletion	2999400	7000700
2	Insertion	5999100	10001400
	Search	3998500	3000900
	Deletion	1999300	6002700
3	Insertion	5999200	9998800
	Search	4001700	2998300
	Deletion	2000200	6000100
4	Insertion	6000000	8999900
	Search	4001400	2998500
	Deletion	2999600	5000600
5	Insertion	10000400	10001800
	Search	3998500	3000000
	Deletion	1999700	5002600
Avg	Insertion	6799300	10000440
	Search	3800020	2999600
	Deletion	2399640	5801340

3.2 Insertion

Average insertion time was $6,799,300$ nanoseconds (ns) for BST, and $10,000,440$ ns for Trie. Insertion was faster with **Binary Search Tree (BST)**, performing approximately 47% faster.

3.3 Search

Average search time was $3,800,020$ ns for BST, and $2,999,600$ ns with Trie. Search obtained faster execution with **Trie**, performing approx. 26% faster.

3.4 Deletion

Average Deletion Time was $2,399,640$ ns for BST, and $5,801,340$ ns for Trie. **BST** performed approx. 141% faster.

3.5 Methodology

The Java `time` library was utilized to track execution time, per `Tester.java`:

```
1  import ...
2
3  public class Tester {
4
5      public static void main(String[] args) {
```

```
        List<String> bookTitles = generateRandomBookTitles(10000);

        // Create BST and Trie instances
        BST<String> bst = new BST();
        Trie trie = new Trie();

        // Measure and compare insertion times
        Instant startTime = Instant.now();
        for (String title : bookTitles) {
            bst.insert(title);
        }
        Instant endTime = Instant.now();
        Duration bstInsertionTime = Duration.between(startTime, endTime);

        startTime = Instant.now();
        for (String title : bookTitles) {
            trie.insert(title);
        }
        endTime = Instant.now();
        Duration trieInsertionTime = Duration.between(startTime, endTime);

        // Measure and compare search times
        startTime = Instant.now();
        for (String title : bookTitles) {
            bst.search(title);
        }
        endTime = Instant.now();
        Duration bstSearchTime = Duration.between(startTime, endTime);

        startTime = Instant.now();
        for (String title : bookTitles) {
            trie.search(title);
        }
        endTime = Instant.now();
        Duration trieSearchTime = Duration.between(startTime, endTime);

        // Measure and compare deletion times
        startTime = Instant.now();
        for (String title : bookTitles) {
            bst.delete(title);
        }
        endTime = Instant.now();
        Duration bstDeletionTime = Duration.between(startTime, endTime);

        startTime = Instant.now();
        for (String title : bookTitles) {
            trie.delete(title);
        }
```

```
        endTime = Instant.now();
        Duration trieDeletionTime = Duration.between(startTime, endTime);

        // Display results
        System.out.println("** Measuring Insert Time **");
        System.out.println("BST Insertion Time : " + bstInsertionTime.toNanos() + " nanoseconds");
        System.out.println("Trie Insertion Time: " + trieInsertionTime.toNanos() + " nanoseconds\n");
        System.out.println("** Measuring Search Time **");
        System.out.println("BST Search Time    : " + bstSearchTime.toNanos() + " nanoseconds");
        System.out.println("Trie Search Time   : " + trieSearchTime.toNanos() + " nanoseconds\n");
        System.out.println("** Measuring Delete Time **");
        System.out.println("BST Deletion Time  : " + bstDeletionTime.toNanos() + " nanoseconds");
        System.out.println("Trie Deletion Time : " + trieDeletionTime.toNanos() + " nanoseconds\n");
    }

    // Helper method to generate random book titles
    private static List<String> generateRandomBookTitles(int count) {
        List<String> bookTitles = new ArrayList<>();
        Random random = new Random();
        for (int i = 0; i < count; i++) {
            StringBuilder title = new StringBuilder();
            int length = random.nextInt(10) + 5; // Random title length between 5 and 14 characters
            for (int j = 0; j < length; j++) {
                char randomChar = (char) (random.nextInt(26) + 'a'); // Random lowercase letter
                title.append(randomChar);
            }
            bookTitles.add(title.toString());
        }
        return bookTitles;
    }
}
```

Unit tests were deployed to verify Binary Search Tree and Trie functionality. Our tests found they correctly executed intended operations. The results were output using the following batch script to run the `Tester.java` file:

```
@echo off
set numberOfTimes=5
set outputFileName=output.txt

echo. > %outputFileName%

for /l %%i in (1, 1, %numberOfTimes%) do (
    java Tester >> %outputFileName%
)

echo "Batch script execution complete."
```

4 Recommendations

Based on the experiments and findings, we can make the following recommendations for the library's cataloging system:

- **Insertion:** In terms of insertion speed, the BST outperformed the Trie. BST insertion time was significantly faster than Trie insertion time.
- **Search:** Trie search operations were faster than BST search operations. The Trie's search time remained relatively consistent throughout all trials.
- **Deletion:** BST showed slightly better performance relative to Trie in our experiments.

Assuming the library's cataloging system requirements are utilized equally, we recommend using the **Binary Search Tree** data structure. While Trie beats BST in search times, the trade off is only a 26% search performance boost, at the cost of BST's superior Insertion and Deletion operations.

However, a limitation of this analysis involves the assumption that the requirements are utilized equally. Libraries rarely remove books given sufficient storage capacity, and function as a repository of knowledge where users may query the catalog for book information (search) more often than the library may be adding or removing books. Thus, if the library catalog remains relatively static, overall utility may be greater using Trie.

5 Conclusion

In this lab, we implemented a Binary Search Tree (BST) and a Trie for a library cataloging system and compared their operation speeds. The experiments showed that the BST performed better in terms of insertion and deletion operations. Therefore, we generally recommend using the Binary Search Tree data structure for the library's cataloging system to achieve efficient and fast book title management.