Advanced Data Structures

COSC 520 Assignment 2

Done by: Emon Sarker, emon331@student.ubc.ca

In this assignment we explore advanced data structures and benchmark their performance when it comes to data insertion, deletion, and search. The three data structures I have chosen are Red-Black Tree, B-Tree, and Doubly Linked XOR List. Each data structure has been implemented and tested with unit tests. Scripts to record and plot their runtime performance for upto a million records has also been written.

1 Introduction to the Data Structures

1.1 Red-Black Tree

Red-Black Trees, introduced in Trees (2004), is a type of binary search tree that uses a set of rules to ensure that the tree remains balanced. Having a balanced tree ensures that operations for insertion, deletion, and search should always have a logarithmic time complexity.

In a Red Black Tree, each node has an additional attribute which is its color- red or black. These colors allows certain rules to be enforced:

1. Root Nodes: The root node is always black
2. Leaf Nodes: The leaf nodes are either black or null
3. Red Nodes: Red nodes cannot have other red nodes as children
4. Black Nodes: Every path from a node to the leaves should have the same number of black nodes

These simple rules help ensure that the longest path is only twice as long as the shortest path, which helps keep the tree balanced.

1.2 B-Tree

B-Trees (not Binary Trees), introduced in Bender et al. (2015), is another self-balancing tree that is designed to be optimized for accessing data stored on a disk. This optimization makes them ideal for building databases. In a B-Tree, each node can hold multiple keys and have thus have multiple children. Thus making this a multi-way tree that is often shallow as well. The shallow nature is an advantage, as when reading from secondary memory each node being read is a slow operation, therefore being able to fetch more data at a time speeds up the overall process.

1.3 Doubly Linked XOR List

Doubly linked XOR list, introduced in Sinha (2005), is a variant of the doubly linked list which is more memory efficient as it only stores one pointer per node. Unlike a doubly-linked list where

each node has two pointers, one that points to the predecessor and the other pointing towards the successor, the XOR linked list stores the bit-wise XOR of the addresses of the nodes that come before and after it. This is possible due to the XOR logic that has the following properties: commutative, associative, identity element, and self-inverse. As the start and the end of the linked list do not have a real predecessor node or a successor node respectively, they are given imaginary nodes with an address of all 0 bits. The data structure works by performing a XOR operation on the previous and the current addresses to find the next address in the linked list.

2 Justification of Data Structure Selection

2.1 Red-Black Tree

Binary trees are a popular data structure for their ease of use and their high performance. Their accessibility is why I like them personally. However, one downside of binary trees is that if they are not balanced they start losing what makes them a good data-structure. Red-Black trees is a simple iteration over the classic binary tree that helps keep the tree balanced. While there are other methods that can also do so, Red-Black trees feel more intuitive and thus easy to work with. While there is a constant overhead with the additional complexity during implementing insertions and deletions, its great for looking up data stored in the tree. So for processes or applications that are read-heavy its a good choice.

2.2 B-Tree

B-Trees are a prime example of a data structure engineered for real-world constraints. B-Trees aim to save time when reading large amounts of data from the secondary storage, which is a very important process needed to build any functional system. Its practicality and usefulness is the reason I like it as its a good tool to have in your toolbox when designing a program that needs to interact with slower memory.

2.3 Doubly Linked XOR List

A linked list is another classic data structure that is used in a lot of programs, therefore knowing a version of it that is an improvement is a really useful tool to have in your toolbox. I personally believe the future of computing is going to require us to be more efficient with resources so we can do more with less. Therefore, knowing tricks that help save memory is exciting.

3 Time and Space Complexity Analysis

3.1 Red-Black Trees

A Red-Black Tree is a self-balancing binary search tree. The time complexities for insertion, deletion, and search operations are:

- **Insertion:** $O(\log n)$
- **Deletion:** $O(\log n)$

- **Search:** $O(\log n)$

where n is the number of nodes in the tree. Since the tree remains balanced, the height of the tree is at most $O(\log n)$, leading to logarithmic complexity for these operations.

The space complexity is $O(n)$, as each node requires a fixed amount of space.

3.2 B-Trees

A B-Tree is a self-balancing tree data structure commonly used in database indexing and file systems. It has a branching factor b, meaning each internal node can have between $\lceil b/2 \rceil$ and b children.

The time complexities for operations are:

- **Insertion:** $O(\log_b n)$
- **Deletion:** $O(\log_b n)$
- **Search:** $O(\log_b n)$

where n is the number of keys stored in the tree, and b is the branching factor. Since the height of the tree is at most $O(\log_b n)$, all fundamental operations run in logarithmic time with respect to the base b.

The space complexity is $O(n)$, as the tree stores n keys along with additional pointers and metadata.

3.3 Doubly Linked XOR Tree

A Doubly Linked XOR Tree is a space-efficient variation of a doubly linked list where each node stores an XOR of its previous and next pointers instead of separate pointers.

The time complexities for operations are:

- **Insertion:** $O(1)$ (if inserting at a known position, otherwise $O(n)$ for traversal)
- **Deletion:** $O(1)$ (if node is known, otherwise $O(n)$ for search)
- **Search:** $O(n)$ (since traversal requires reconstructing pointers from the XOR values)

The space complexity is $O(n)$ since each node stores a single XOR pointer, reducing space usage compared to a traditional doubly linked list, but still proportional to the number of elements.

4 Runtime Performance

4.1 Experiment setup

A benchmark is designed to evaluate the runtime complexity of the three data structures by conducting incremental performance testing across dataset sizes ranging from 100,000 to 1,000,000

elements, with increments of 100,000. At each dataset size, a new set of random integers within the range of 1 to 1,000,000 is generated using a fixed seed of 42 to ensure reproducibility.

During the insertion phase, all elements of the dataset are inserted into an initially empty data structure. The search phase measures the time required to locate each element within the populated structure. Finally, in the deletion phase, all elements are removed from the data structure, with balanced deletion strategies applied to tree-based structures by processing elements in sorted order.

4.2 Performance Analysis

The recorded times for insertion, deletion, and search are found in Table 1, Table 2, and Table 3 respectively. The results are visualized through separate performance graphs corresponding to insertion, deletion, and search operations. This facilitates a comparative analysis of how each data structure scales as dataset size increases, providing insights into their relative efficiency and operational characteristics under varying workloads.

The observed performance trends largely align with theoretical expectations for Red-Black trees, exhibiting logarithmic time complexity for insertion, deletion, and search operations. However, the B-tree's unexpectedly high search times and the consistently superior performance of the XOR Linked List warrant further analysis. Specifically, the B-tree's search implementation and the XOR Linked List's traversal mechanism require closer examination to ensure the observed results accurately reflect the underlying data structure characteristics. It's suspect that Python's just in time compilation might have some side-effects when it comes to the implementation.

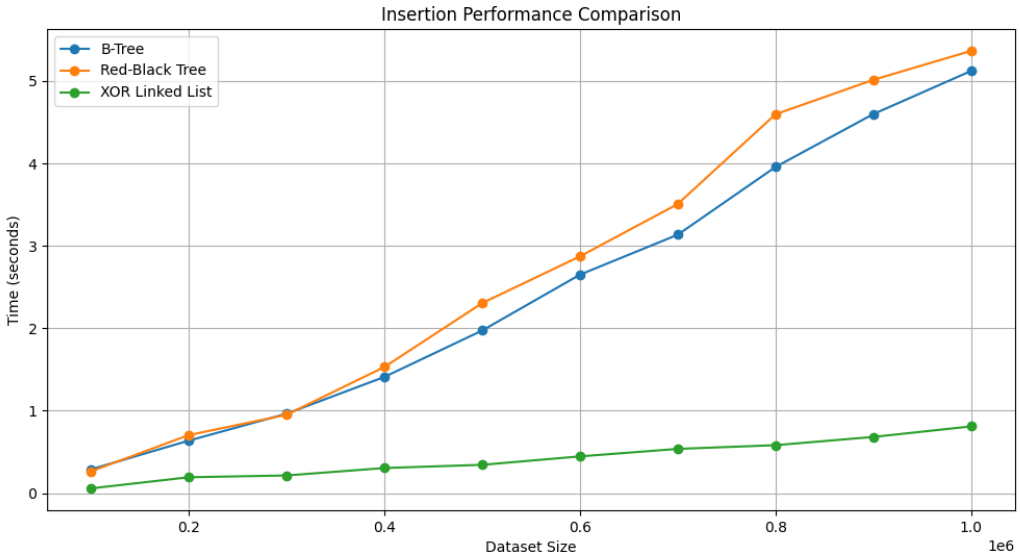

Figure 1: This graph compares the insertion performance of B-trees, Red-Black trees, and XOR Linked Lists. It's clear that XOR Linked Lists have significantly faster insertion times than both B-trees and Red-Black trees across all dataset sizes. While the difference between B-trees and Red-Black trees is less pronounced, B-trees appear to have slightly faster insertion times, especially as the dataset size increases.

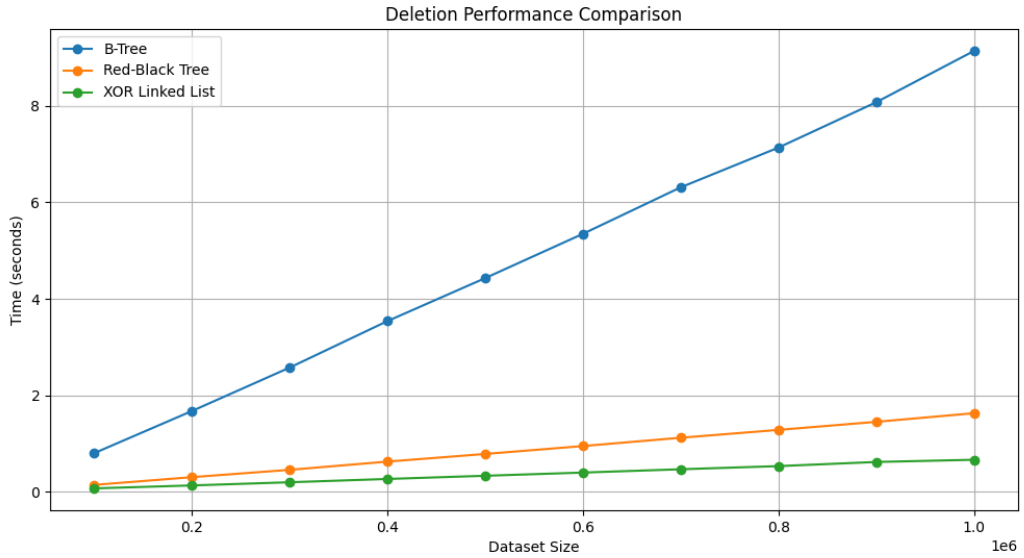

Figure 2: This graph compares the deletion performance of B-trees, Red-Black trees, and XOR Linked Lists as the dataset size increases. The B-tree exhibits the highest deletion times, showing a steep linear increase with dataset size, indicating it becomes significantly slower as the data grows. In contrast, both Red-Black trees and XOR Linked Lists demonstrate much lower and more gradual increases in deletion time, with XOR Linked Lists consistently showing the fastest deletion performance among the three.

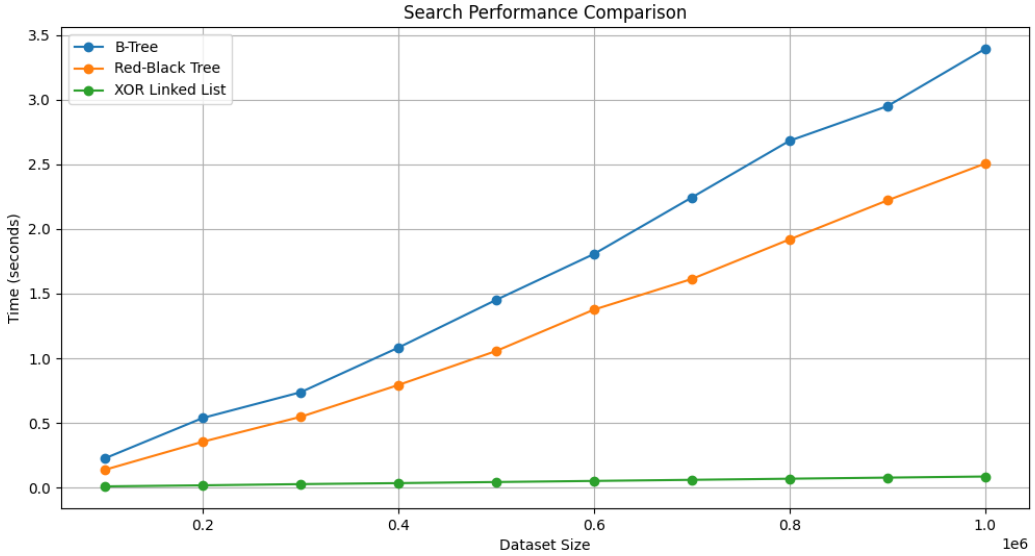

Figure 3: This graph compares the search performance of B-trees, Red-Black trees, and XOR Linked Lists. B-trees have the longest search times, increasing linearly with dataset size. Red-Black trees perform better, but XOR Linked Lists demonstrate superior search efficiency with consistently low search times.

Table 1: Performance Results for Insertion

Data Structure	Dataset Size	Time (seconds)
Red-Black Tree	100000	0.265688
	200000	0.704594
	300000	0.951161
	400000	1.531159
	500000	2.305613
	600000	2.873103
	700000	3.507963
	800000	4.596933
	900000	5.014337
	1000000	5.365769
B-Tree	100000	0.289015
	200000	0.637497
	300000	0.964171
	400000	1.410319
	500000	1.973182
	600000	2.650803
	700000	3.135479
	800000	3.957369
	900000	4.598844
	1000000	5.122318
XOR Linked List	100000	0.057819
	200000	0.192773
	300000	0.214682
	400000	0.304786
	500000	0.343680
	600000	0.445846
	700000	0.536821
	800000	0.581984
	900000	0.681874
	1000000	0.809906

Table 2: Performance Results for Deletion

Data Structure	Dataset Size	Time (seconds)
Red-Black Tree	100000	0.145242
	200000	0.305518
	300000	0.457761
	400000	0.627908
	500000	0.786508
	600000	0.950187
	700000	1.122888
	800000	1.285052
	900000	1.451969
	1000000	1.631094
B-Tree	100000	0.802244
	200000	1.675964
	300000	2.576155
	400000	3.539562
	500000	4.430009
	600000	5.347889
	700000	6.311380
	800000	7.134863
	900000	8.074982
	1000000	9.138150
XOR Linked List	100000	0.073350
	200000	0.134144
	300000	0.200190
	400000	0.268406
	500000	0.334626
	600000	0.400788
	700000	0.469466
	800000	0.534666
	900000	0.620466
	1000000	0.666056

Table 3: Performance Results for Search

Data Structure	Dataset Size	Time (seconds)
Red-Black Tree	100000	0.137979
	200000	0.354757
	300000	0.547099
	400000	0.793359
	500000	1.056535
	600000	1.376298
	700000	1.613119
	800000	1.919380
	900000	2.221114
	1000000	2.505838
B-Tree	100000	0.226691
	200000	0.538596
	300000	0.737313
	400000	1.082450
	500000	1.451895
	600000	1.806758
	700000	2.243750
	800000	2.682449
	900000	2.950051
	1000000	3.393286
XOR Linked List	100000	0.009010
	200000	0.017129
	300000	0.026466
	400000	0.034165
	500000	0.042660
	600000	0.051116
	700000	0.059645
	800000	0.068310
	900000	0.076767
	1000000	0.085205

5 Dataset

5.1 Dataset Generation

The dataset generation system is implemented through the `DatasetGenerator` class. The system creates standardized numerical datasets for performance testing of the data structures. The generation process is automated through the `generate_datasets.py` script.

5.2 Dataset Structure

Each dataset consists of randomly generated integers with the following characteristics:

- **Value Range:** 1 to 1,000,000 (configurable through `min_value` and `max_value` parameters)
- **Format:** JSON arrays containing the generated integers
- **Storage:** Datasets are stored in the `datasets` directory with descriptive filenames indicating their size

5.3 Generating a Dataset

The dataset generation is controlled by the `generate_standard_datasets()` function, which:

- Initializes a `DatasetGenerator` instance
- Uses a fixed random seed (42) for reproducibility
- Generates datasets for each size category
- Saves each dataset as a JSON file in the `datasets` directory

The `DatasetGenerator` class provides several key methods:

- `generate_dataset(size, seed)`: Creates a new dataset of specified size
- `save_dataset(dataset, name)`: Saves a dataset to a JSON file
- `load_dataset(name)`: Loads a dataset from a JSON file
- `list_datasets()`: Returns names of all available datasets

All dataset generation uses a fixed random seed (42) to ensure:

- Reproducible test results
- Consistent performance comparisons
- Reliable benchmarking across different runs

Link to dataset: *https://github.com/emondsarker/advanced-data-structures/tree/main/datasets*

6 Github Repository

https://github.com/emondsarker/advanced-data-structures

Code convention:

This section outlines the coding conventions followed in the implementation, ensuring consistency, readability, and maintainability.

6.1 Naming Conventions

- **Private Methods:** Underscore prefix (e.g., `_fix_insert`, `_transplant`).
- **Public Interface:** Descriptive verbs (e.g., `insert`, `delete`, `read`).
- **Variables:** Self-documenting names (e.g., `node`, `parent`, `color`).
- **Constants:** Uppercase with underscores.
- **Classes:** PascalCase (e.g., `RedBlackTree`, `Node`).

6.2 Code Organization

6.2.1 Class Structure

- Constructor and initialization.
- Public interface methods.
- Private helper methods.
- Utility functions.

6.2.2 Method Ordering

- Core operations first (e.g., `insert`, `read`, `delete`).
- Supporting operations following.
- Internal utilities last.

6.3 Style Consistency

- **Indentation:** 4 spaces.
- **Line Length:** PEP 8 compliance.
- **Method Separation:** Double line breaks.
- **Comment Alignment:** Inline with code.
- **Parameter Alignment:** Vertical alignment in multi-line calls.

References

Bender, M. A., Farach-Colton, M., Jannen, W., Johnson, R., Kuszmaul, B. C., Porter, D. E., Yuan, J., and Zhan, Y. (2015). An introduction to b-trees and write-optimization. *login; magazine*, 40(5).

Sinha, P. (2005). A memory-efficient doubly linked list. *Linux Journal*, 2005(129):10.

Trees, R.-B. (2004). Red-black trees. *Red*, 6(7):4.