Empirical Analysis of Unbalanced and Balanced Tree Data Structures

Vivian Zhou

December 20th, 2023

Contents

1 Introduction

The primary objective of this analysis is to explore and understand the concept of binary tree structures in computer science, with a specific focus on balanced and unbalanced trees. The analysis aims to provide insights into the characteristics, advantages, and disadvantages of these two types of tree structures, specifically comparing the performance of insertion and delete operations. For each tree, three measures of performance were conducted: worst case, average case, and amortized complexity.

2 Background

2.1 Balanced Tree

A balanced tree is a type of tree data structure in which the height of the left and right subtrees of any node differs by at most one. The goal of maintaining balance is to ensure that the tree remains relatively shallow, leading to more efficient operations. Balanced trees are advantageous for maintaining logarithmic time complexity in operations like insertion, deletion, and search.

2.1.1 Key Characteristics:

- Logarithmic Height: Balanced trees maintain a logarithmic height, ensuring that operations like search, insertion, and deletion have a time complexity of O(log n), where n is the number of elements in the tree.

2.1.2 Time Complexities

- Insert Operation:
  - Worst Case: O(log n)
  - Average Case: O(log n)
  - Amortized Complexity: O(log n)
- Delete Operation:
  - Worst Case: O(log n)

    - Average Case: O(log n)
    - Amortized Complexity: O(log n)

- Delete Minimum Operation:
    - Worst Case: O(log n)
    - Average Case: O(log n)
    - Amortized Complexity: O(log n)

2.2 Unbalanced Tree

An unbalanced tree, on the other hand, does not adhere to strict height balance criteria. This lack of balance can result in certain branches of the tree being significantly longer than others. In unbalanced trees, the time complexity of operations may degrade to O(n) in the worst case, making them less efficient compared to balanced trees. Simple binary search trees (BSTs) are an example of unbalanced trees.

2.2.1 Key Characteristics:

- Variable Height: Unbalanced trees can have variable heights, and in the worst case, the height may be linear with the number of elements. This can lead to a time complexity of O(n) for certain operations.

2.2.2 Time Complexities

- Insert Operation:
    - Worst Case: O(n)
    - Average Case: O(n)
    - Amortized Complexity: O(n)
- Delete Operation:
    - Worst Case: O(n)
    - Average Case: O(n)

  - Amortized Complexity: O(n)

- Delete Minimum Operation:
  - Worst Case: O(n)
  - Average Case: O(n)
  - Amortized Complexity: O(n)

3 Experimental Setup

In order to efficiently present a comparison between the performances of balanced and unbalanced trees, the independent variable was chosen to be the tree size, ranging from 1-4096 nodes, increasing increments of powers of two. Because the expected time complexity is O(log(n)) for balanced trees and O(n) for unbalanced trees, having tree sizes of powers of two will effectively illustrate an exponential relationship in the balanced trees and a linear relationship in the unbalanced trees. The dependent variable, the performance metric, was chosen to be the number of comparisons made in each operation. A comparison would be defined as any operation required to perform the desired action, for example the comparison of two nodes. Using the number of comparisons as the performance metric ensures an accurate measure of performance with relatively low noise, compared to measuring, let's say the time spent in execution.

4 Code

4.1 Balanced Tree Insertion, Deletion, and Deletion of Minimum Node Algorithms (main method omitted)

```
#include <bits/stdc++.h>
#include <chrono>
#include <numeric>
using namespace std;

int numComparisons = 0;

struct Node {
    int data;
    Node* left;
    Node* right;
    Node(int value) {
        data = value;
        left = nullptr;
        right = nullptr;
    }
};

int getHeight(Node* node) {
    if (node == nullptr) {
        return 0;
    }

    numComparisons+=2;
    return max(getHeight(node->left), getHeight(node->right)) + 1;
}

void balanceTree(Node*& node) {
    if (node == nullptr) {
        return;
    }

    numComparisons+=2;
    int leftHeight = getHeight(node->left);
    int rightHeight = getHeight(node->right);

    if (abs(leftHeight - rightHeight) <= 1) {
        // Tree is already balanced
        return;
    }

    if (leftHeight > rightHeight) {
        // Left subtree is taller, perform right rotation
        if (getHeight(node->left->left) < getHeight(node->left->right)) {
            // Left-right case, perform left rotation on left child first
            Node* temp = node->left;
            node->left = temp->right;
            temp->right = node->left->left;
            node->left->left = temp;
        }

        // Perform right rotation on current node
        Node* temp = node;
        node = temp->left;
        temp->left = node->right;
        node->right = temp;
    } else {
        // Right subtree is taller, perform left rotation
        if (getHeight(node->right->right) < getHeight(node->right->left)) {
            // Right-left case, perform right rotation on right child first
            Node* temp = node->right;
            node->right = temp->left;
            temp->left = node->right->right;
            node->right->right = temp;
        }

        // Perform left rotation on current node
        Node* temp = node;
        node = temp->right;
        temp->right = node->left;
        node->left = temp;
    }

    // Recursively balance the left and right subtrees
    balanceTree(node->left);
    balanceTree(node->right);
}


void insert(Node*& root, int value) {
    if (root == nullptr) {
        root = new Node(value);
        return;
    }

    if (value < root->data) {
        numComparisons++;
        insert(root->left, value);
    } else {
        numComparisons++;
        insert(root->right, value);
    }

    balanceTree(root);
}

void deleteNode(Node*& root, int value) {
    if (root == nullptr) {
        return;
    }

    if (value < root->data) {
        deleteNode(root->left, value);
        numComparisons++;
    } else if (value > root->data) {
        deleteNode(root->right, value);
        numComparisons++;
    } else {
        // Node to be deleted found
        if (root->left == nullptr && root->right == nullptr) {
            numComparisons+=2;
            // Case 1: Node has no children
            delete root;
            root = nullptr;
        } else if (root->left == nullptr) {
            numComparisons+=3;
            // Case 2: Node has only right child
            Node* temp = root;
            root = root->right;
            numComparisons++;
            delete temp;
        } else if (root->right == nullptr) {
            numComparisons+= 4;
            // Case 3: Node has only left child
            Node* temp = root;
            root = root->left;
            numComparisons++;
            delete temp;
        } else {
            numComparisons+= 4;
            // Case 4: Node has both left and right children
            Node* successor = root->right;
            numComparisons++;
            while (successor->left != nullptr) {
                numComparisons++;
                successor = successor->left;
            }
            root->data = successor->data;
            deleteNode(root->right, successor->data);
        }
    }

    balanceTree(root);
}

void deleteMin(Node*& root) {
    if (root == nullptr) {
        numComparisons++;
        return;
    }

    if (root->left == nullptr) {
        numComparisons++;
        // Minimum value found at root
        Node* temp = root;
        root = root->right;
        delete temp;
    } else {
        numComparisons++;
        deleteMin(root->left);
    }

    balanceTree(root);
}
```

4.2 Unbalanced Tree Insertion, Deletion, and Deletion of Minimum Node Algorithms (main method omitted)

```
#include <bits/stdc++.h>
#include <chrono>
#include <numeric>
using namespace std;

int numComparisons = 0;

struct Node {
    int data;
    Node* left;
    Node* right;

    Node(int value) {
        data = value;
        left = nullptr;
        right = nullptr;
    }
};

void insert(Node*& root, int value) {
    if (root == nullptr) {
        root = new Node(value);
        return;
    }

    if (value < root->data) {
        numComparisons++;
        insert(root->left, value);
    } else {
        numComparisons++;
        insert(root->right, value);
    }
}

void deleteNode(Node*& root, int value) {
    if (root == nullptr) {
        return;
    }

    if (value < root->data) {
        deleteNode(root->left, value);
        numComparisons++;
    } else if (value > root->data) {
        deleteNode(root->right, value);
        numComparisons++;
    } else {
        // Node to be deleted found
        if (root->left == nullptr && root->right == nullptr) {
            numComparisons+=2;
            // Case 1: Node has no children
            delete root;
            root = nullptr;
        } else if (root->left == nullptr) {
            numComparisons+=3;
            // Case 2: Node has only right child
            Node* temp = root;
            root = root->right;
            numComparisons++;
            delete temp;
        } else if (root->right == nullptr) {
            numComparisons+= 4;
            // Case 3: Node has only left child
            Node* temp = root;
            root = root->left;
            numComparisons++;
            delete temp;
        } else {
            numComparisons+= 4;
            // Case 4: Node has both left and right children
            Node* successor = root->right;
            numComparisons++;
            while (successor->left != nullptr) {
                numComparisons++;
                successor = successor->left;
            }
            root->data = successor->data;
            deleteNode(root->right, successor->data);
        }
    }
}
```

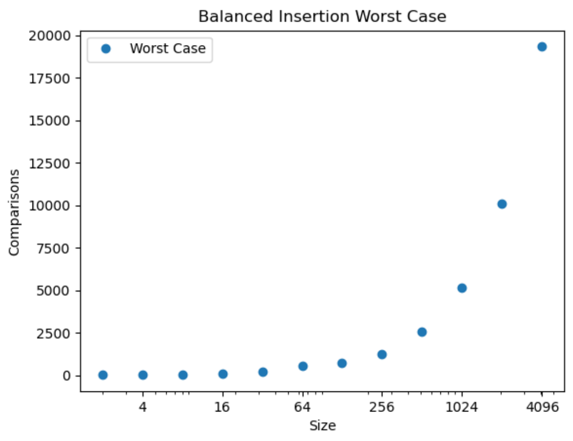
```
void deleteMin(Node*& root) {
    if (root == nullptr) {
        numComparisons++;
        return;
    }

    if (root->left == nullptr) {
        numComparisons++;
        // Minimum value found at root
        Node* temp = root;
        root = root->right;
        delete temp;
    } else {
        numComparisons++;
        deleteMin(root->left);
    }
}
```

4.3 Auxiliary Code (plotting graphs)

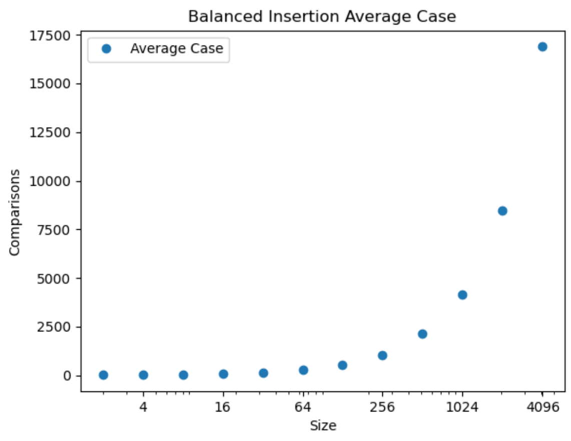
```
import matplotlib.pyplot as plt
import matplotlib.ticker as ticker
import pandas as pd

# Read the CSV file
data = pd.read_csv("/Users/vivyw/cs/data.csv", delimiter=',')

# Extract the columns
size = data['Size']
worst = data['Worst']
average = data['Average']
amortized = data['Amortized']

# Plot the data as dots
plt.plot(size, worst, 'o', label='Worst Case')

# Add labels and title
plt.xlabel('Size')
plt.ylabel('Comparisons')
plt.title('Unbalanced Deletion of Minimum Node Worst Case')
plt.xscale('log')
plt.gca().xaxis.set_major_locator(ticker.LogLocator(base=2))
plt.gca().xaxis.set_major_formatter(ticker.FuncFormatter(lambda y, _: '{:g}'.format(y)))

# Add legend
```

```
plt.legend()

# Show the plot
plt.show()

plt.clf()

# Plot the data as dots
plt.plot(size, average, 'o', label='Average Case')

# Add labels and title
plt.xlabel('Size')
plt.ylabel('Comparisons')
plt.title('Unbalanced Deletion of Minimum Node Average Case')
plt.xscale('log')
plt.gca().xaxis.set_major_locator(ticker.LogLocator(base=2))
plt.gca().xaxis.set_major_formatter(ticker.FuncFormatter(lambda y, _: '{:g}'.format(y)))

# Add legend
plt.legend()

# Show the plot
plt.show()

plt.clf()

# Plot the data as dots
plt.plot(size, amortized, 'o', label='Amortized Complexity')

# Add labels and title
plt.xlabel('Size')
plt.ylabel('Comparisons')
plt.title('Unbalanced Deletion of Minimum Node Amortized Complexity')
plt.xscale('log')
plt.gca().xaxis.set_major_locator(ticker.LogLocator(base=2))
plt.gca().xaxis.set_major_formatter(ticker.FuncFormatter(lambda y, _: '{:g}'.format(y)))

# Add legend
plt.legend()

# Show the plot
plt.show()
```

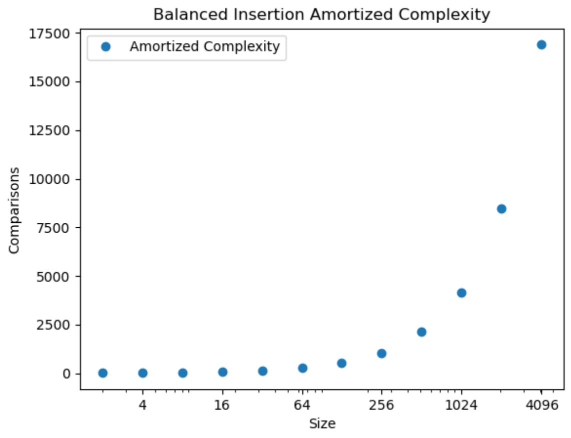
5 Results



(a) Worst Case

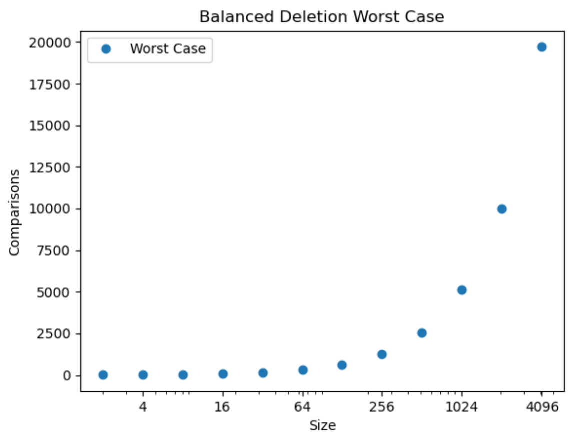


(b) Average Case

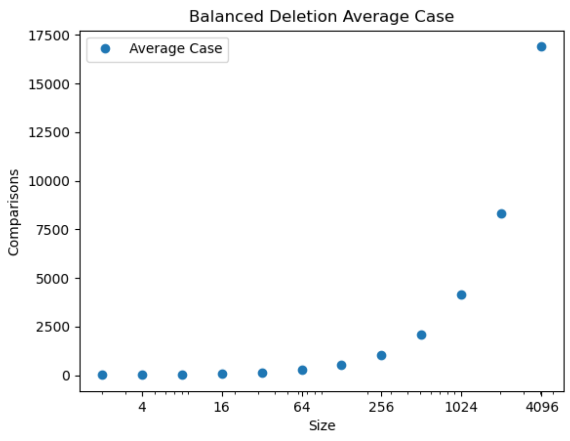


(c) Amortized Complexity

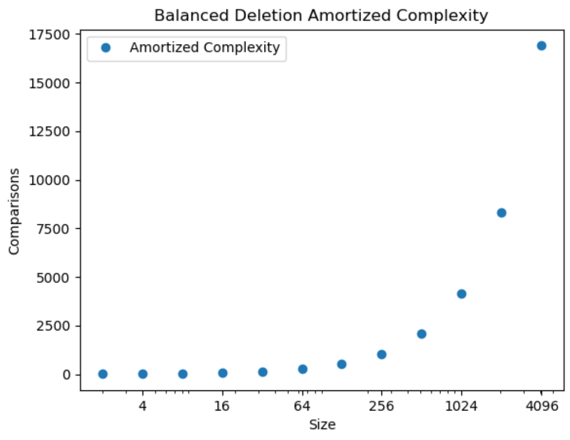
Figure 1: Balanced Tree Insertion Performance



(a) Worst Case

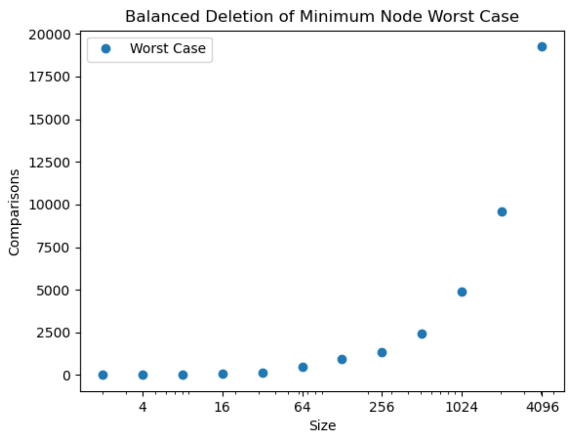


(b) Average Case

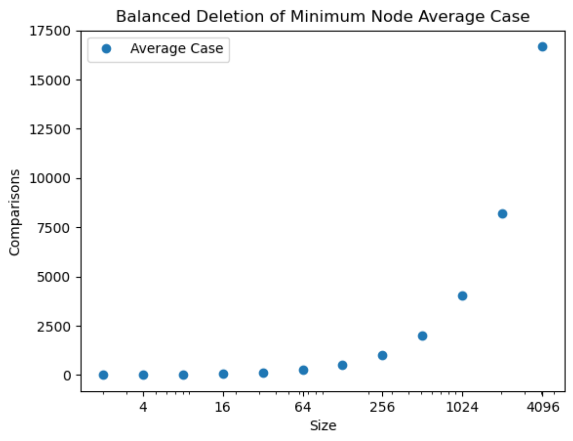


(c) Amortized Complexity

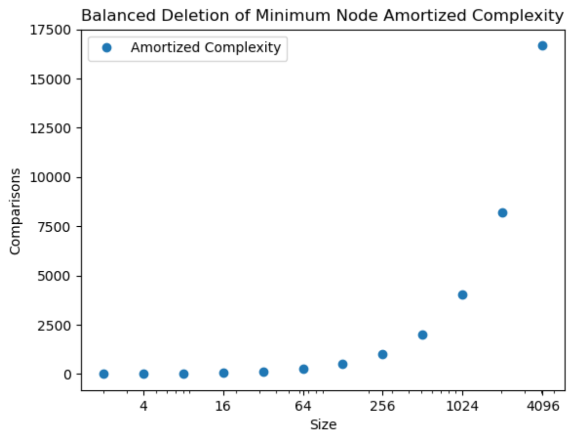
Figure 2: Balanced Tree Deletion Performance



(a) Worst Case

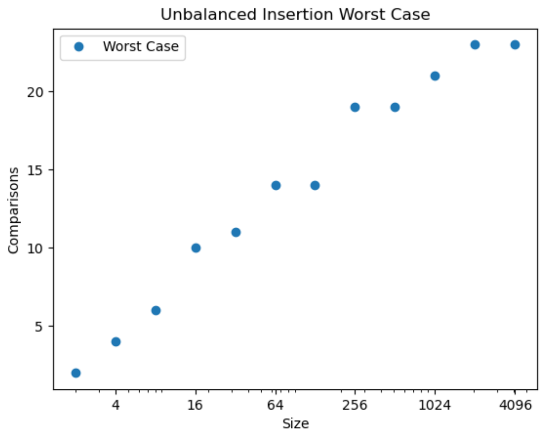


(b) Average Case

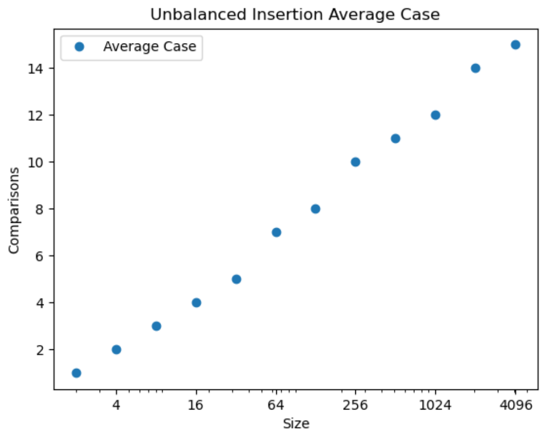


(c) Amortized Complexity

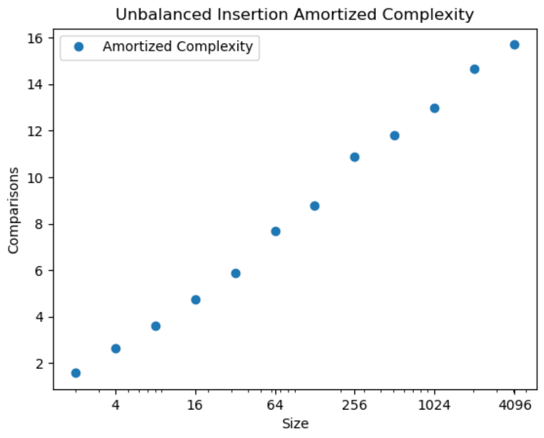
Figure 3: Balanced Tree Deletion of Minimum Node Performance



(a) Worst Case

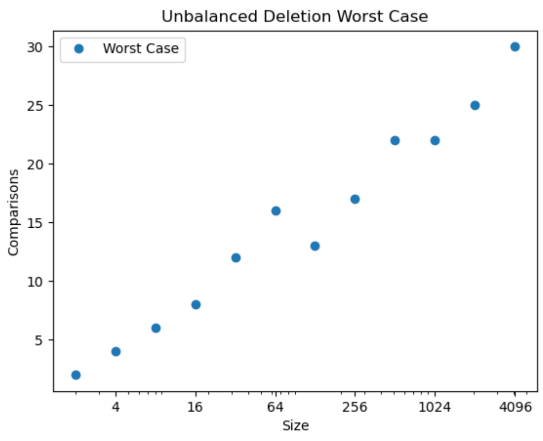


(b) Average Case

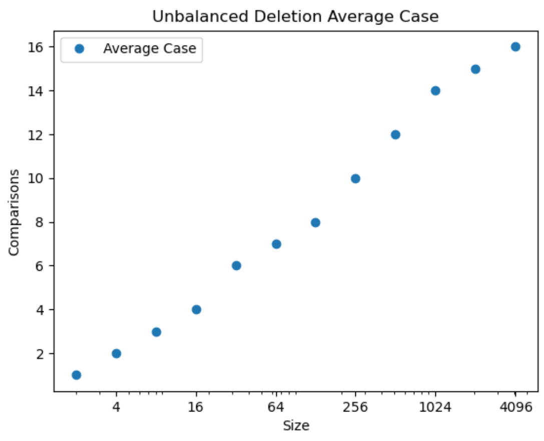


(c) Amortized Complexity

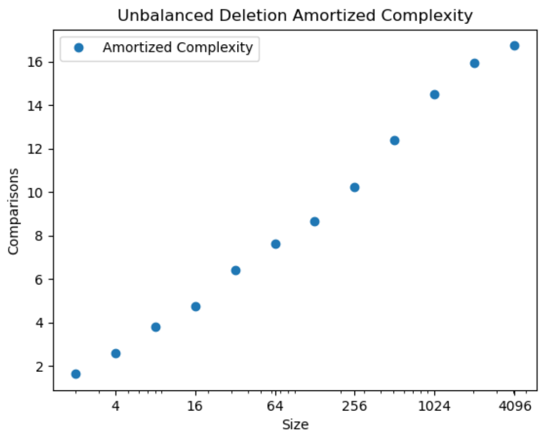
Figure 4: Unbalanced Tree Insertion Performance



(a) Worst Case

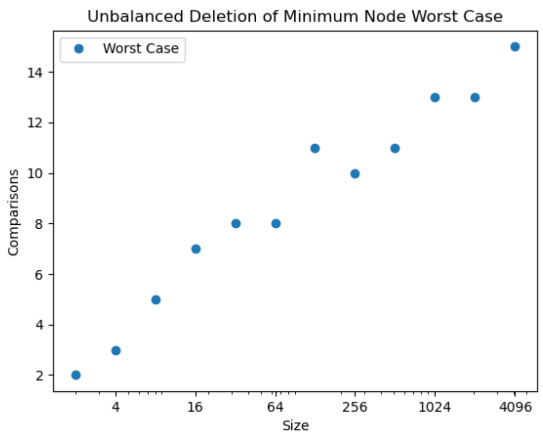


(b) Average Case

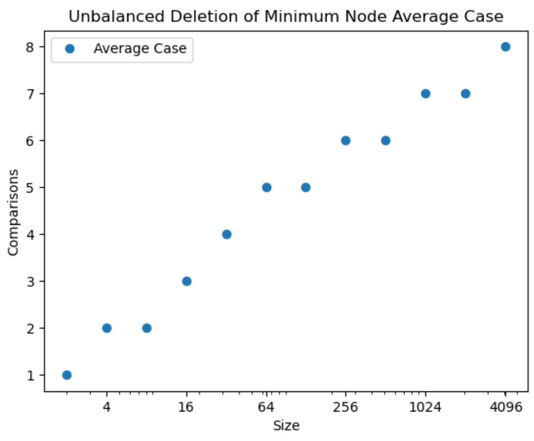


(c) Amortized Complexity

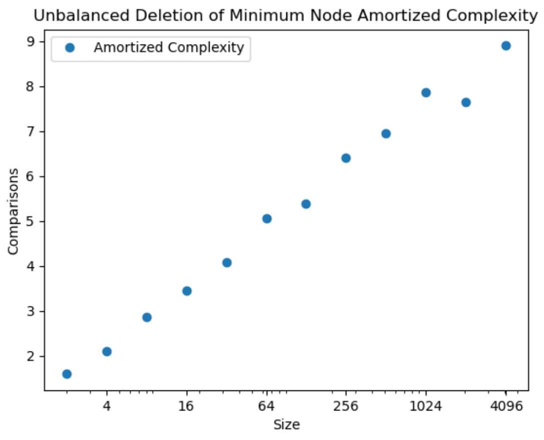
Figure 5: Unbalanced Tree Deletion Performance



(a) Worst Case



(b) Average Case



(c) Amortized Complexity

Figure 6: Unbalanced Tree Deletion of Minimum Node Performance