Lab 5: Scapegoat Trees

Learning Outcomes

- Read and interpret a written description of an algorithm
- Correctly implement an algorithm from pseudocode
- Generate test cases for the implementation
- Design and execute benchmarks for an algorithm

Overview

In this lab, you will read a paper describing the scapegoat balanced binary search tree and implement the data structure based on that description.

Instructions

Read the provided paper and submit a report containing the following:

1. Read sections 1 – 4, 6, and 8 – 9 of the provided paper and answer the following questions:
    a. How do scapegoat trees compare with Red-Black, AVL, and splay trees? Why might you prefer to use or not use a scapegoat tree?
    b. What does it mean for a node to be weight balanced? What does it mean for a tree to be weight-balanced? Draw some examples and calculate their weight balances.
    c. What is the interpretation of the α parameter?
    d. What are the conditions for triggering a rebuild of a subtree (during inserts) or the entire tree (during deletes)?
2. Implement a scapegoat tree that supports insert, size, delete, and contains operations. The tree should additionally support a toList() operation that generates a list from an in-order traversal. Add a counter variable to keep track of the number of times the rebuild operation is performed. You do not need to implement the logarithmic space rebuild algorithm described in 6.1 and 6.2 – use the straightforward approach.
3. Write unit tests that involve the insert, remove, size, contains, and toList() operations.
4. Benchmark the insert, delete, and contains operations of your implementation on data sets of different sizes. Create tables and plots that include both run times and the number of times the rebuild operation was performed.
5. Analyze and interpret the benchmark results to determine if the run time of your implementation is consistent with the theoretical analysis.

Submission Instructions

Save your report as a PDF. Upload your report and source code files to Canvas.

Rubric

		Full Credit	Partial Credit	No Credit
Report writing and presentation quality	15%			
Reading response questions	10%	The decision rule is correct for all possible inputs that conform to the problem description	The described rule is correct for most inputs.	The decision rule is not correct for some common cases.
Implementation of the insert, size, contains, delete, and toList() operations	25%	Implementations are correct without consideration of time complexity.	Implementations are mostly correct without consideration of time complexity.	Implementations are fundamentally flawed.
Logic for triggering a rebuild on insert and rebuilding the subtree are correct	10%	Implementations are correct without consideration of time complexity.	Implementations are mostly correct without consideration of time complexity.	Implementations are fundamentally flawed.
Logic for triggering a rebuild on delete and rebuilding the entire tree are correct	10%	Implementations are correct without consideration of time complexity.	Implementations are mostly correct without consideration of time complexity.	Implementations are fundamentally flawed.
Test Cases	10%	Test cases consider a range of problem sizes and complexities and potential edge cases.	Limited number of test cases or only testing obvious or simple cases.	No test cases.
Benchmarks	10%	Benchmark experiments were set up and run correctly.	Benchmark experiments, implementations, or results are mostly correct.	Benchmark experiments, implementations, or results are flawed.
Implementation run time	10%	Empirical results support theoretical run times from paper.	Empirical results support theoretical run times from paper for most operations.	Implementation is slower than expected from theoretical run time.