Dictionary Implementations

Jerry Fanelli

Bocconi University
Computer Science Algorithms

25/05/2021

Outline

Introduction

What is a Dictionary

A **dictionary** is a general purpose data structure consisting of a set of keys each of which gets associated to a single value.

Dictionaries can be implemented in various ways. The most common methods are through *hash tables*, *binary trees*, *linked lists* and *tries*. The choice of implementation depends on the specific use that is intended for the dictionary.

Hash Tables

A **hash table** is a data structure that stores data within a flat array and distributes the data according to the output of a hash function. Given a hash table with hash function h and n entries, the hash function is a function from a key K to a positive real number in a range $0 < h(x) < M$, $M \geq n$

The output of the hash function $h(x)$ usually generates a quasi-random number that can be used as an offset in the hash table. As a result, the offset of key K within the hash table will be measured as $h(K) \mod n$, and will result in an almost uniform distribution of key-value pairs.

A *hash collision* occurs when the output of the hash function of two different keys returns the same offset in the hash table
$K_1 \neq K_2, h(K_1) \mod n = h(K_2) \mod n$

There are mainly two methods to deal with *collisions*:

- **Separate Chaining**, where the entry in the table will point to the start of a linked list that can be searched in linear time O(n)
- **Open Addressing**, where all the elements are stored in the Hash Table and can be done in different ways, such as *linear probing*

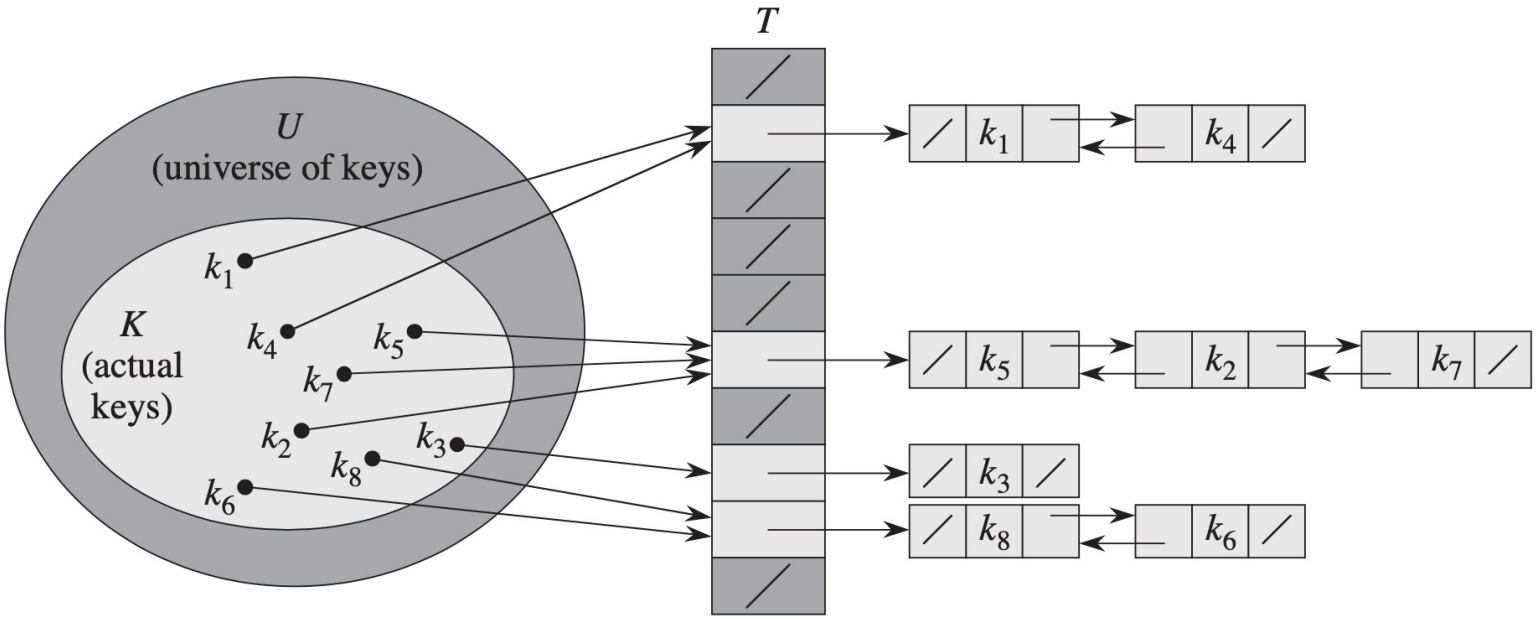

Figure 11.3 Collision resolution by chaining. Each hash-table slot $T[j]$ contains a linked list of all the keys whose hash value is j. For example, $h(k_1) = h(k_4)$ and $h(k_5) = h(k_7) = h(k_2)$. The linked list can be either singly or doubly linked; we show it as doubly linked because deletion is faster that way.

Figure 1: Hash Table - Chaining

Hash Tables - Time Complexity

Average

- Search: O(1)
- Insertion: O(1)
- Deletion: O(1)

Worst

- Search: O(n)
- Insertion: O(n)
- Deletion: O(n)

Binary Search Trees

What is a Red Black Tree

A **Binary Search Tree** is a binary tree where each node is arranged according to a specific order: *left descendants* $\leq n \leq$ *right descendants* for each node n.

This structure allows to efficiently maintain a dynamically changing dataset in sorted order while also enabling insertion and deletion at logarithmic time, O(log n).

It is possible to implement a dictionary with a binary search tree, by simply assigning to each node K, a value V. In this way if the tree is reasonably well balanced, we can guarantee search and insert operations in O(log n) time.

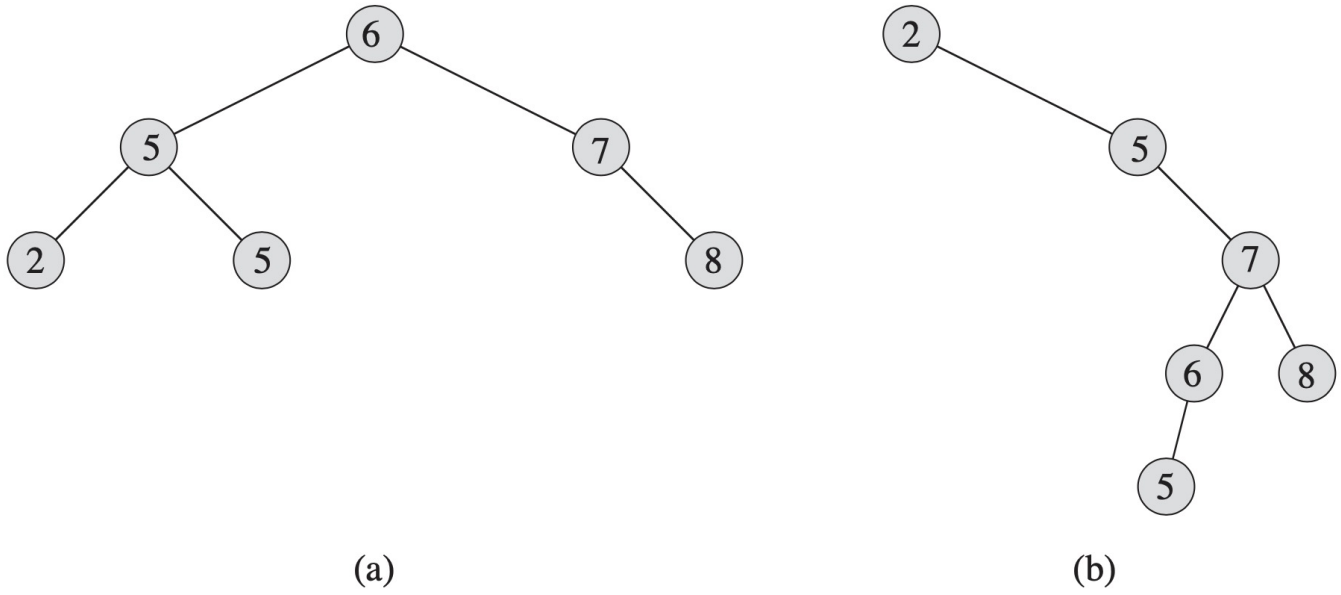

Figure 12.1 Binary search trees. For any node x, the keys in the left subtree of x are at most $x.key$, and the keys in the right subtree of x are at least $x.key$. Different binary search trees can represent the same set of values. The worst-case running time for most search-tree operations is proportional to the height of the tree. **(a)** A binary search tree on 6 nodes with height 2. **(b)** A less efficient binary search tree with height 4 that contains the same keys.

Figure 2: Binary Search Tree

Red Black Trees

A **Red Black Tree** is a type of self-balancing BST that adheres to a strict set of rules in order to maintain a logarithmic time complexity:

1. Each node must be either red or black;
2. The root of the tree must always be black;
3. The leaves, which are $NULL$ nodes, are considered black;
4. Two red nodes can never appear consecutively, one after another;
5. Every branch path must pass through the exact same number of black nodes;

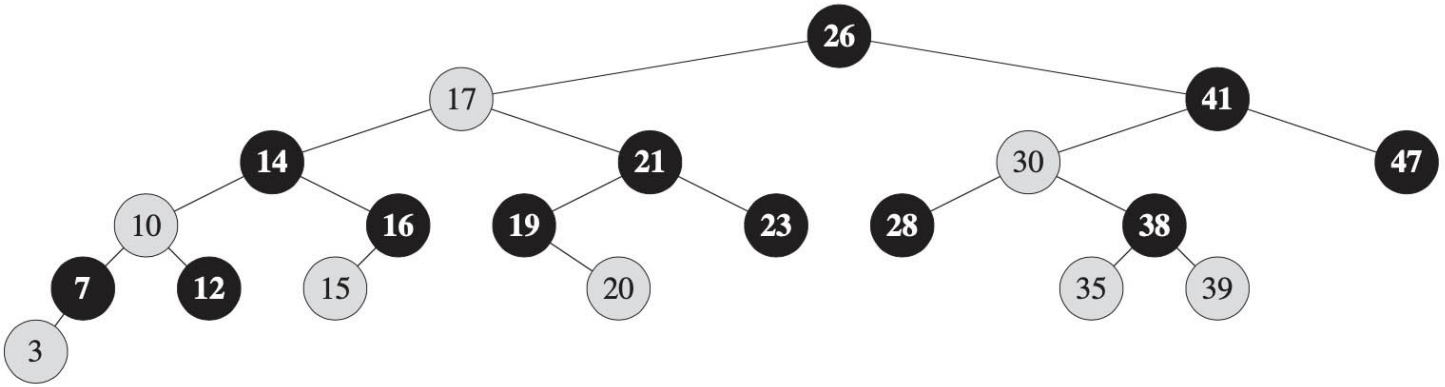

Figure 13.1 A red-black tree with black nodes darkened and red nodes shaded. Every node in a red-black tree is either red or black, the children of a red node are both black, and every simple path from a node to a descendant leaf contains the same number of black nodes.

Figure 3: Red Black Tree

Red Black Trees - Time Complexity

Average

- Search: O(log n)
- Insertion: O(log n)
- Deletion: O(log n)

Worst

- Search: O(log n)
- Insertion: O(log n)
- Deletion: O(log n)

Introduction

The analyses were conducted from two perspectives:

- 1 Performance evaluation with respect to the size of the hash tables
- 2 Performance evaluation on Insert and Get operations

The hash tables used for the analysis are distinguished with respect to their hash functions: (1) Naive, (2) Default, (3) MD5, (4) SHA256, (5) Jenkins

Instead for the binary search tree, an implementation of Red Black Tree has been used and adapted to be employed as a dictionary.

Hash Functions used:

- **Naive**: It computes the hash by summing the unicode code point for each char in the input string;
- **Default**: It uses the built-in hash functions from Python;
- **MD5**: It is a widely used hash function, mainly for cryptographic purposes even though it is now "broken";
- **SHA256**: It is a cryptographic hash function used in some of the most popular authentication and encryption protocols without any known collision ever found. However, it takes a very long time to compute the hash;
- **Jenkins**: the Jenkins hash and its variants require more computation, but they give a better distribution and thus will generally make for fewer collisions and a smaller required table size;

Size of hash tables

The size of the hash table is one of the tuning parameters, to be taken into account for performance analysis. Indeed, the size of the hash table has a direct impact on the number of hash collisions that will occur during the use of the dictionary.

There is no correct hash table size, but it tends to be adjusted based on the number of items we want to keep in the hash table.

I tested the Insert and Get operations, respectively for 10,000 and 1,000 keys.

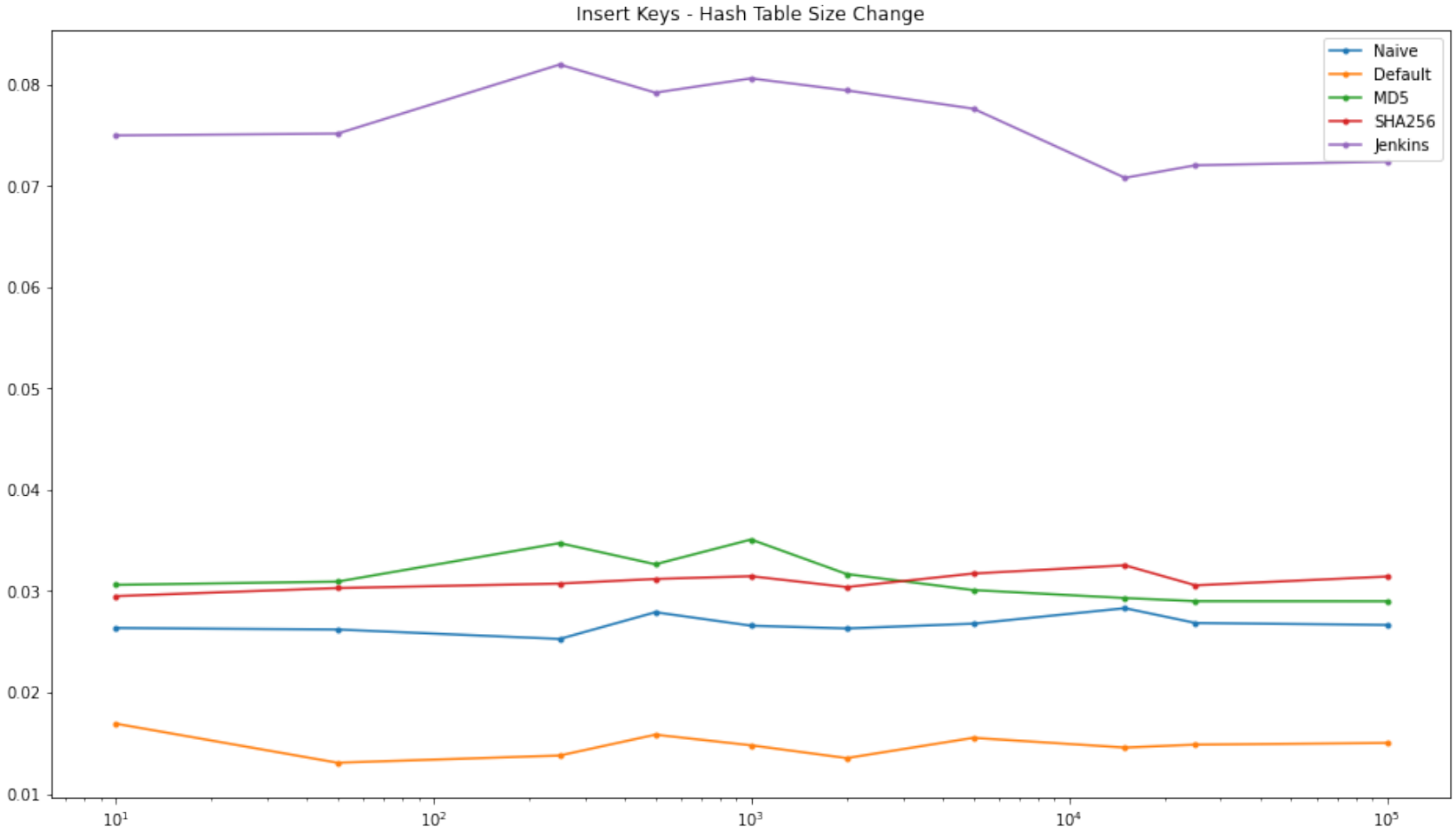

Figure 4: Size - Insert Operation

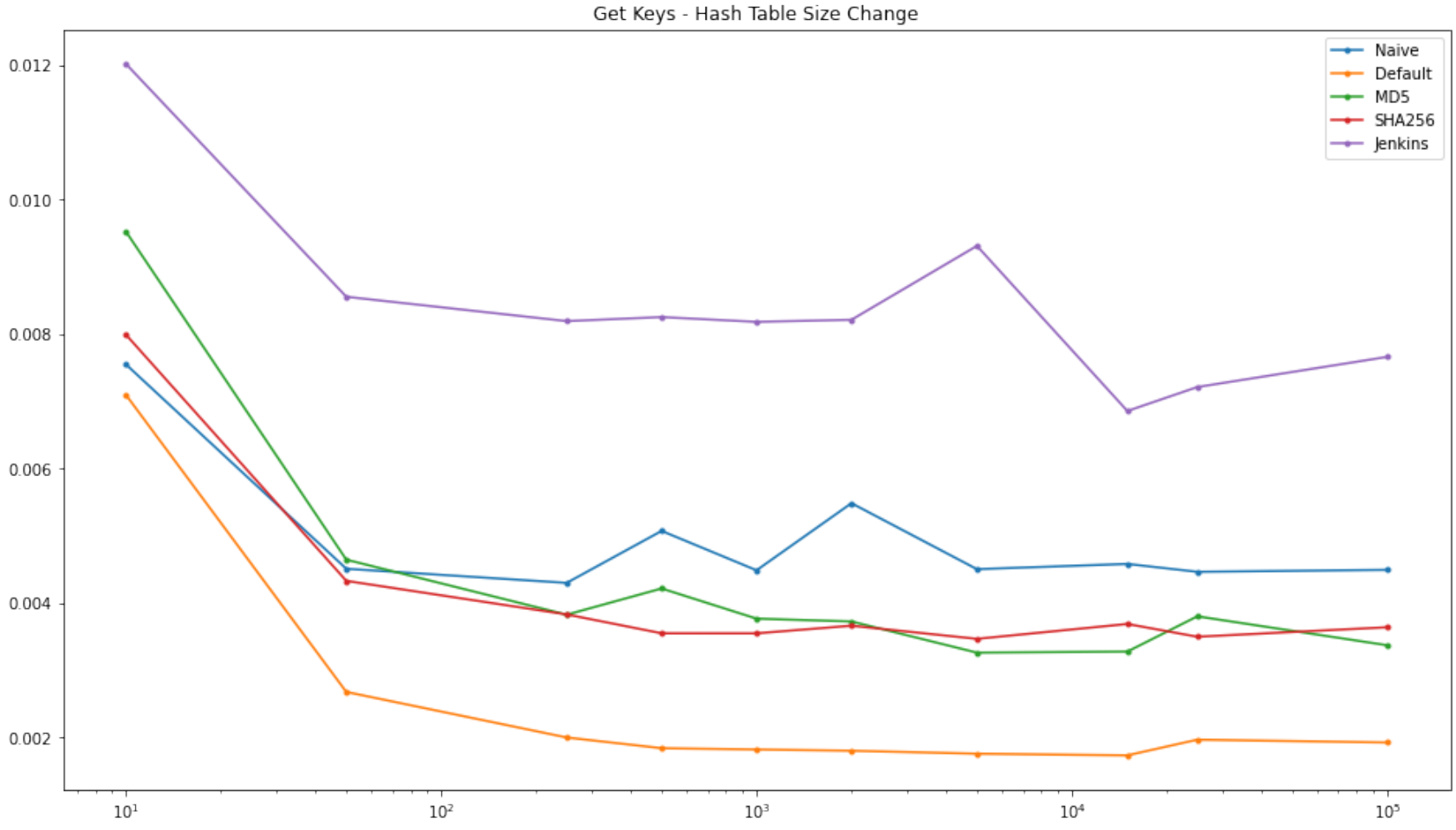

Figure 5: Size - Get Operation

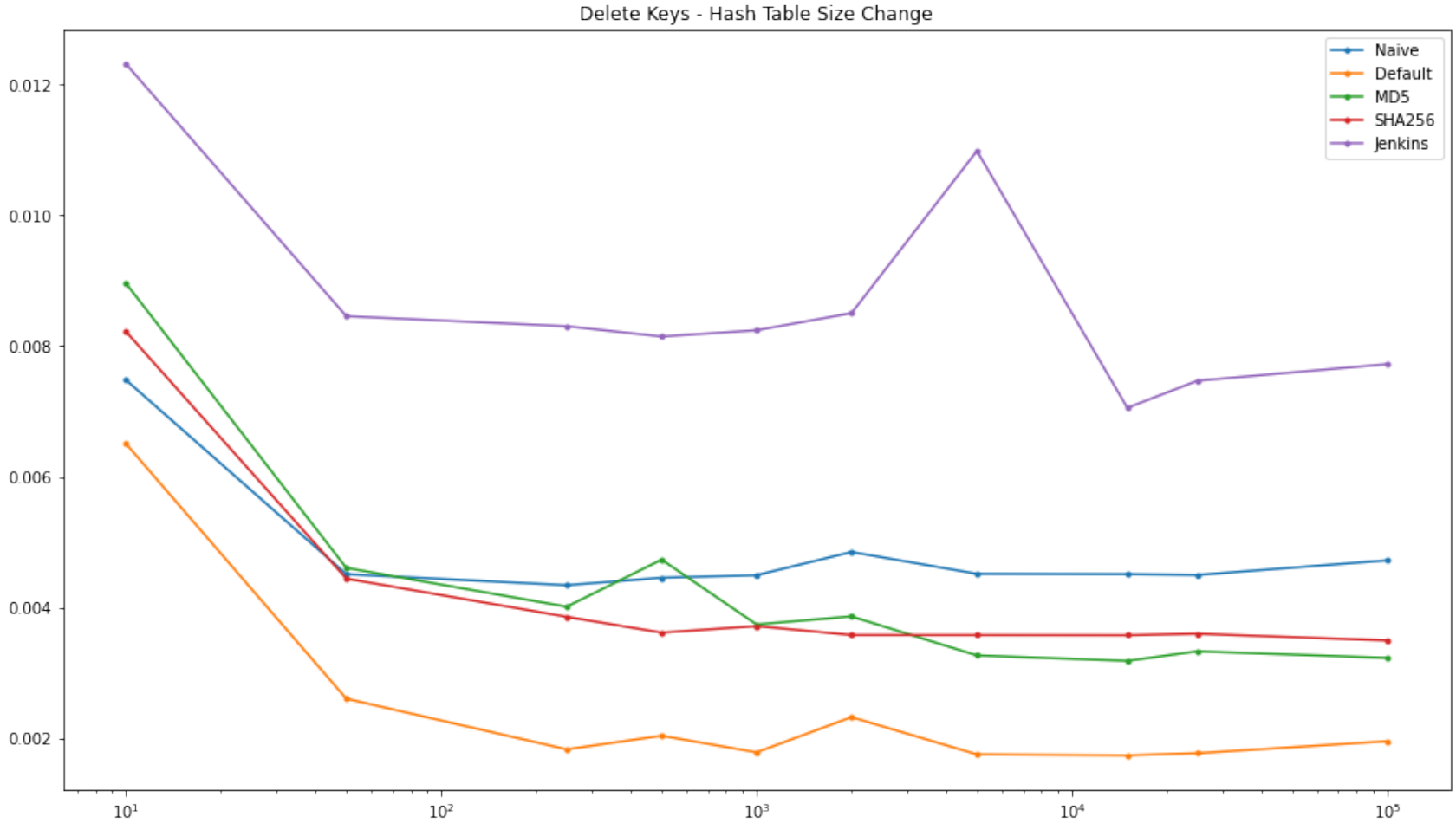

Figure 6: Size - Delete Operation

Number of Keys

The number of keys inserted and retrieved is the factor that actually determines the execution time. By varying the number of keys used, we can verify the differences between Hash Tables and Red Black Trees, and determine which hash function is more performant.

For the analysis I kept the size of the hash tables constant at 250, while the number of keys varies from 10 to $50,000$.

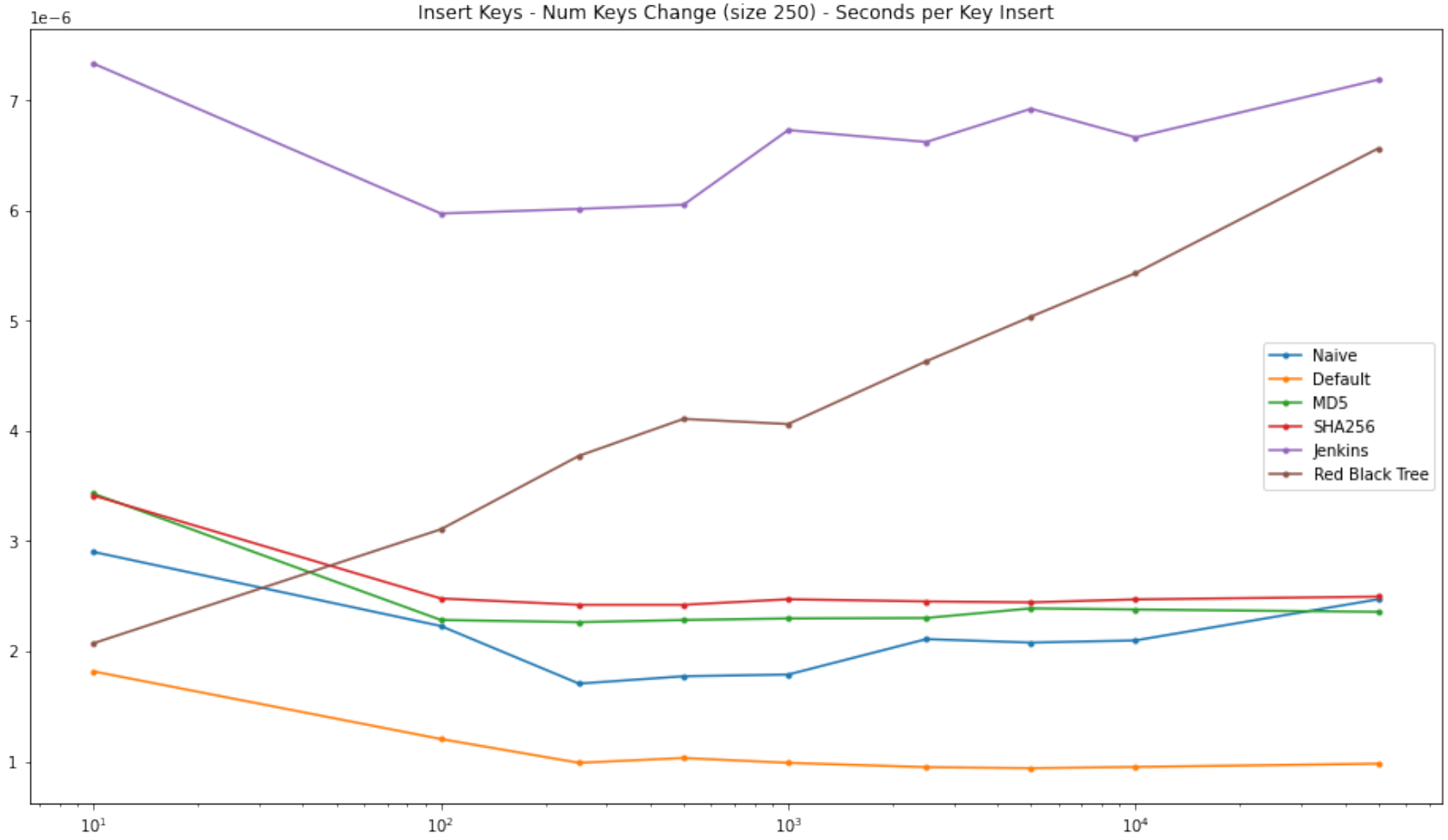

Figure 7: Keys - Insert Operation

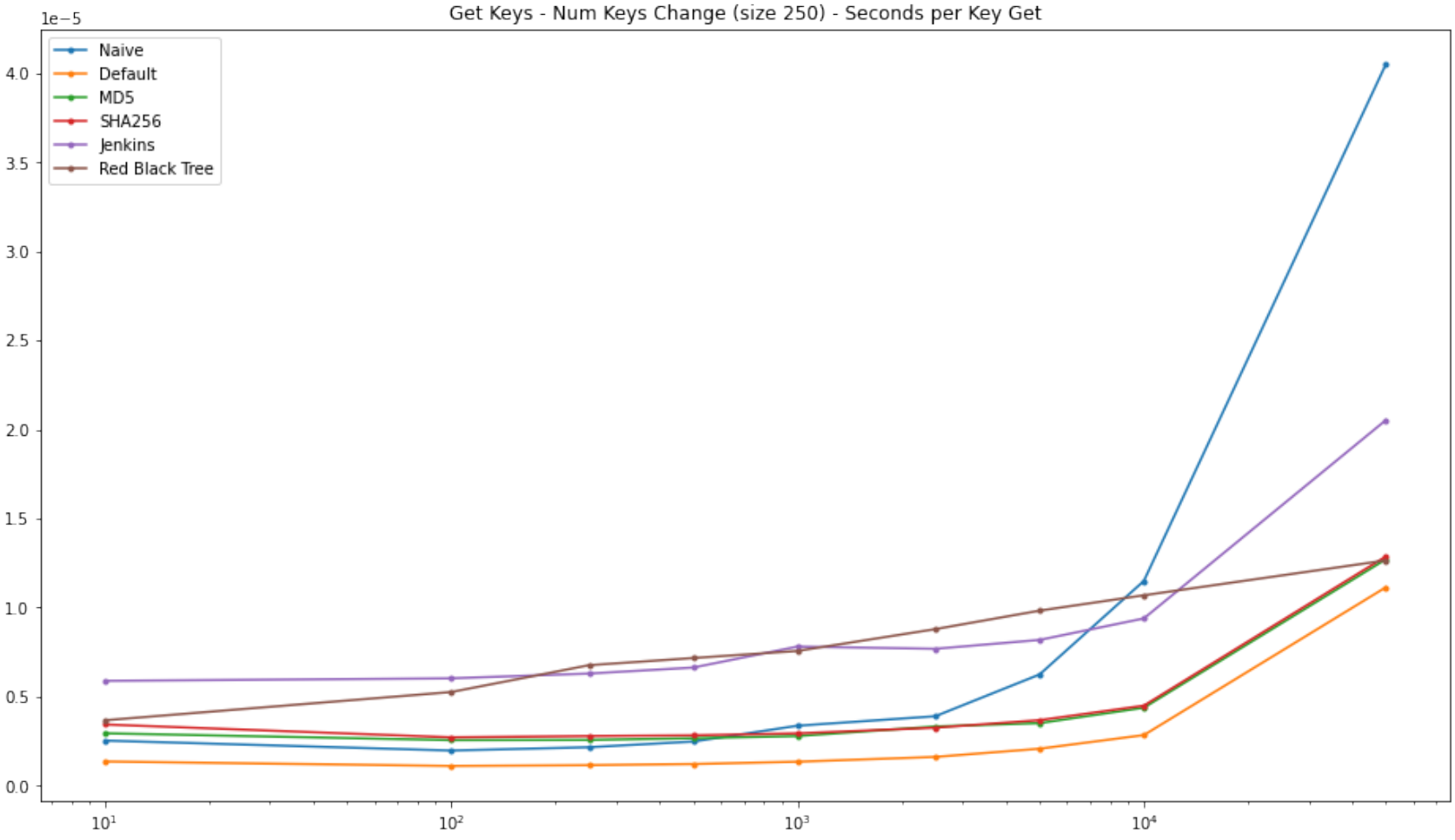

Figure 8: Keys - Get Operation

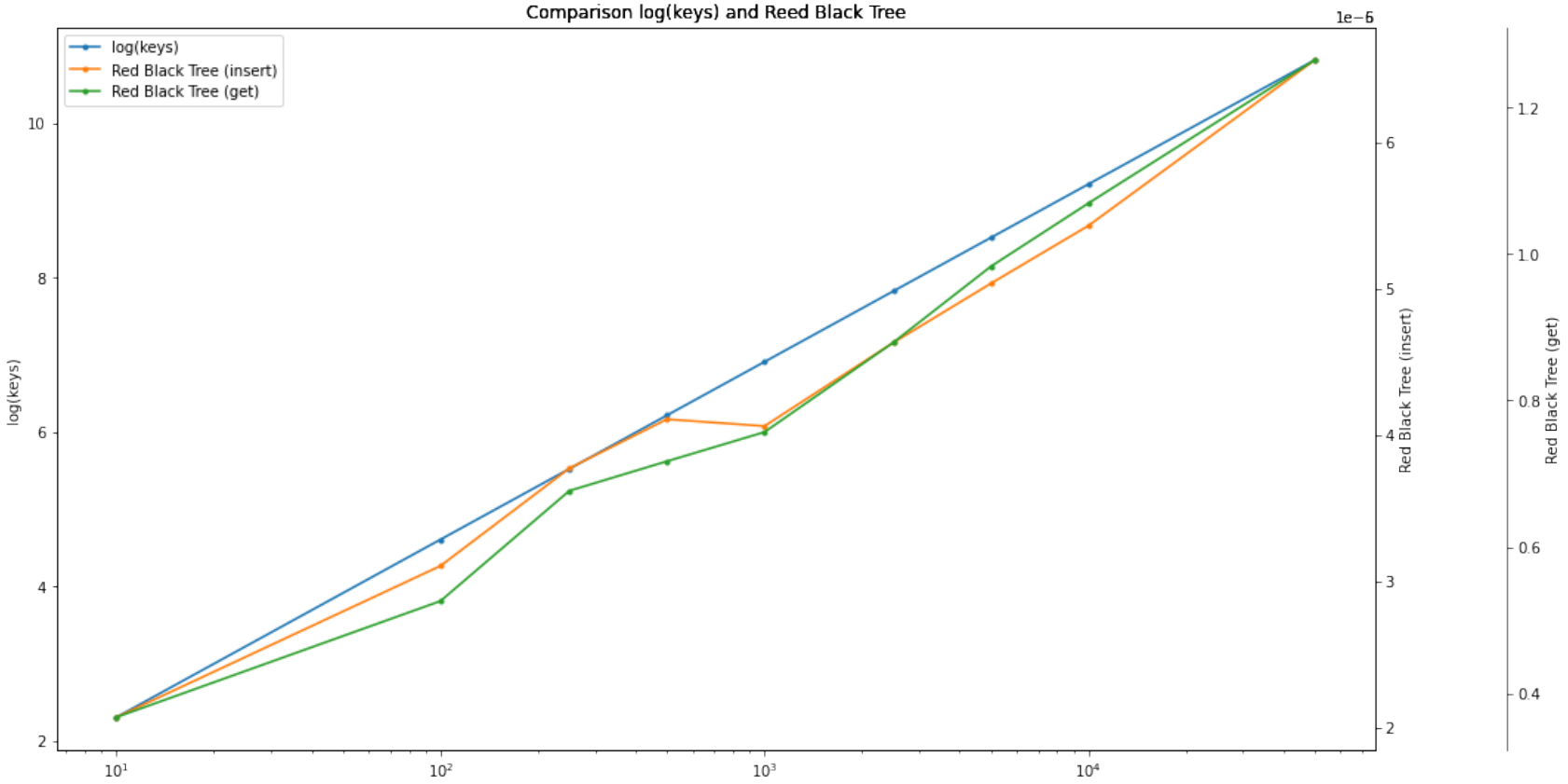

Figure 9: Comparision log(keys) and Red Black Tree

Findings

- The tests performed reflect the theoretical expectations in terms of time complexity;
- In order to maintain a desired degree in performance, it is necessary to adapt the implementation to the problem at hand;
- In the case of hash tables, the choice of hash function is one of the most important factors;
- To improve the performance of hash tables we could implement a dynamic table size based on the number of keys to be handled;

Further Work

- Implement **Open Addressing** and analyze performance;
- Introduce and estimate for **cache efficiency**;
- Use a **Nested Hash Table** to deal with collisions instead of a Python list;

References

- T.H. CORMEN, C.E. LEISERSON, R.L. RIVEST, et al., *Introduction to Algorithms*, MIT, 3rd edition.
- TOM VAN DIJK, *Analysing and Improving Hash Table Performance*, 2009.
- SAMUEL MARSH, *Dictionary*, 2015.

Appendix

- JERRY FANELLI, *Dictionaries Implementation. Github Repository*