CSE-3203

Double Hashing Performance Evaluation

S.M.Mehrabul Islam
SH-86
smmehrabul-2017614964@cs.du.ac.bd

Department of Computer Science & Engineering
University of Dhaka
August 27, 2022

Contents

1 Introduction

We've been asked to choose a preferred data-structure of our own choice and evaluate Double Hashing's performance with that.

So, we've implemented a Python [1] Application that will do so. We got two program that could be run:

- `dataset_preparation.py`

  Initially, run this script to generate the dataset which will be stored in the following path: `"./data/"`

- `main.py`

  This is the main script. Run this after dataset generation. So, this script will:

  - Load the dataset from `"./data/"`
  - Run different data-structures on them which are defined here: `"./data_structures/"`
  - Save their performance measures on separate files, here: `"./performance/"`
  - Finally, display the performance comparisons (descriptive, tabular & visual) based on those measures.

Github

- `https://github.com/smmehrab/double-hashing-performance-evaluation`

Run

(a) `git clone https://github.com/smmehrab/double-hashing-performance-evaluation.git`

(b) `python3 dataset_preparation.py`

(c) `python3 main.py`

2 Dataset

(a) Firstly, We generated 10000 unique random numbers from integer range. Later We've used it as the set of keys for the data structures.

(b) As the key set has already been generated randomly, the key set itself can be used as a random insertion sequence of 10000 numbers.

(c) Then, We generate a random search sequence of 3000 numbers. To do that We maintained the following probabilities:

- P(from the key set) = 0.7
- P(not from the key set) = 0.3
- P(repetition) = 0.2

(d) Lastly, from the insert sequence & search sequence, We've generated a insert-search sequence with following probabilities:

- P(insertion) = 0.7
- P(search) = 0.3

This insert-search sequence will later be used to evaluate the performance of the different data structures.

3 Double Hashing

Double Hashing [3, 4] is a collision resolving technique in Open Addressed Hashing [3, 5]. Double hashing uses the idea of applying a second hash function to key when a collision occurs.

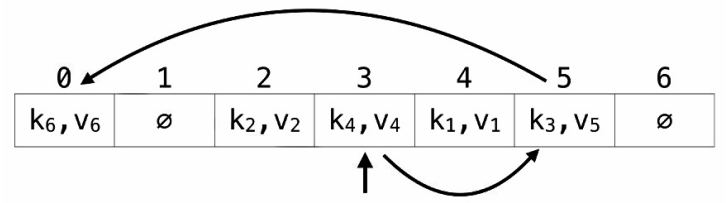


Figure 1: Double Hashing

Complexities:

- Time Complexity: O(n)
- Space Complexity: O(n)

The following implementation details must be mentioned:

(a) table size = (key sets size) * 1.2

(b) p = (maximum prime) less than (table size)

(c) double hash(k) = (h_1(k) - probe * h_2(k)) % table size

- h_1(k) = k % table size
- h_2(k) = p - (k % p)

As We'm using 120% of the cardinality of the key sets as the table size, it should be fine if 20% more data gets added to the table.

But if there's more than 20% of initial data gets added to the table, the table may overflow. And the table size will then be needed to increase for the excess amount of data.

4 Red Black Tree

As our preferred data structure to compare with Double Hashing, We've chosen Red Black Tree [6, 7].

Red Black Tree is a kind of self-balancing binary search tree where each node stores an extra bit representing "color" ("red" or "black") used to ensure that the tree remains balanced during insertions and deletions.

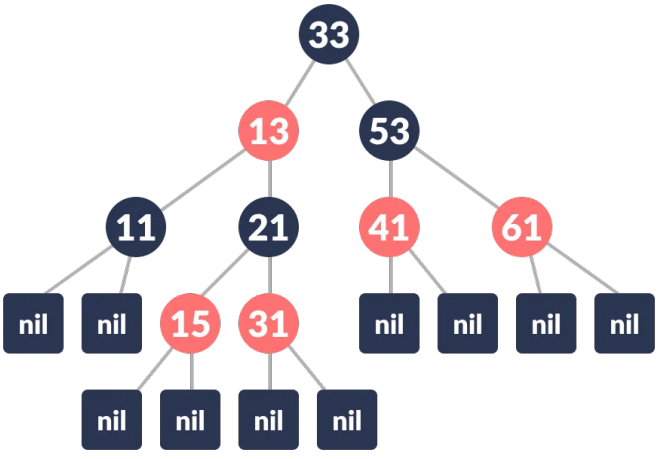


Figure 2: Red Black Tree

Complexities:

Operation	**Average**	**Worst**
Space	O(n)	O(n)
Search	O(log n)	O(log n)
Insert	O(log n)	O(log n)
Delete	O(log n)	O(log n)

The following implementation details must be mentioned:

(a) `Node`

- `value`
- `parent`
- `left`
- `right`
- `color`

(b) `RedBlackTree`

- `search_node(key)`
- `insert_node(key)`
- `print_tree()`

`And other internal methods.`

5 Performance Comparison

Here, the performance measures for the above two data structures are:

- Double Hashing: `number of probes`
- Red Black Tree: `number of inspection`

After running both Double Hashing & Red Black Tree on the same dataset, We compared the performance measures in 3 ways:

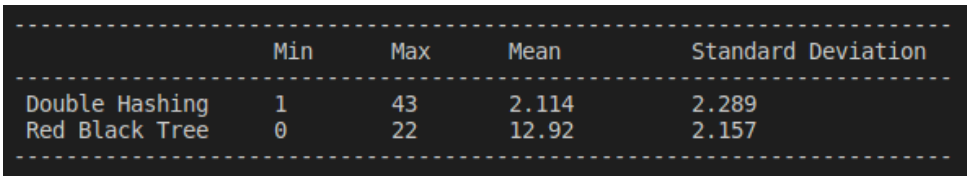
1. Descriptive Comparison
2. Tabular Comparison
3. Visual Comparison

5.1 Descriptive Comparison

The program generated descriptive comparison between Double Hashing & Red Black Tree based on the following statistical properties of the performance measurements:

- Min, Max, Mean & Standard Deviation

The part of the program output, that shows the descriptive comparison is attached below:



	Min	Max	Mean	Standard Deviation
Double Hashing	1	43	2.114	2.289
Red Black Tree	0	22	12.92	2.157

Figure 3: Descriptive Comparison

5.2 Tabular Comparison

During dataset preparation, we've generated the insert-search sequence randomly.

So, if we take the performance measures of the last 100 insert-search sequence for both of the data-structures, we should get a useful comparison between them based on some random insert-search sequence, while each of the data-structures is well-populated.

The part of the program output, that shows the tabular comparison is attached below:

SL	OP	Double Hashing	Red Black Tree
12901	INSERT	6	15
12902	INSERT	3	13
12903	INSERT	8	12
12904	INSERT	14	13
12905	INSERT	17	14
12906	INSERT	5	15
12907	INSERT	8	12
12908	INSERT	2	11
12909	INSERT	3	12
12910	INSERT	4	14
12911	INSERT	6	14
12912	INSERT	6	13
12913	INSERT	2	13
12914	INSERT	6	13
12915	INSERT	1	13
12916	INSERT	2	14
12917	INSERT	3	17
12918	INSERT	5	14
12919	INSERT	13	16
12920	INSERT	2	13
12921	INSERT	5	16
12922	INSERT	2	13
12923	INSERT	2	15
12924	INSERT	11	16

Figure 4: Tabular Comparison

5.3 Visual Comparison

Similarly, we visualized the performance measures of those data structures for the last 100 insert-search sequence, using a manually generated scatter plot.

- `Double Hashing Data Point: (*)`
- `Red Black Tree Data Point: (.)`

The part of the program output, that shows the visual comparison is attached below:

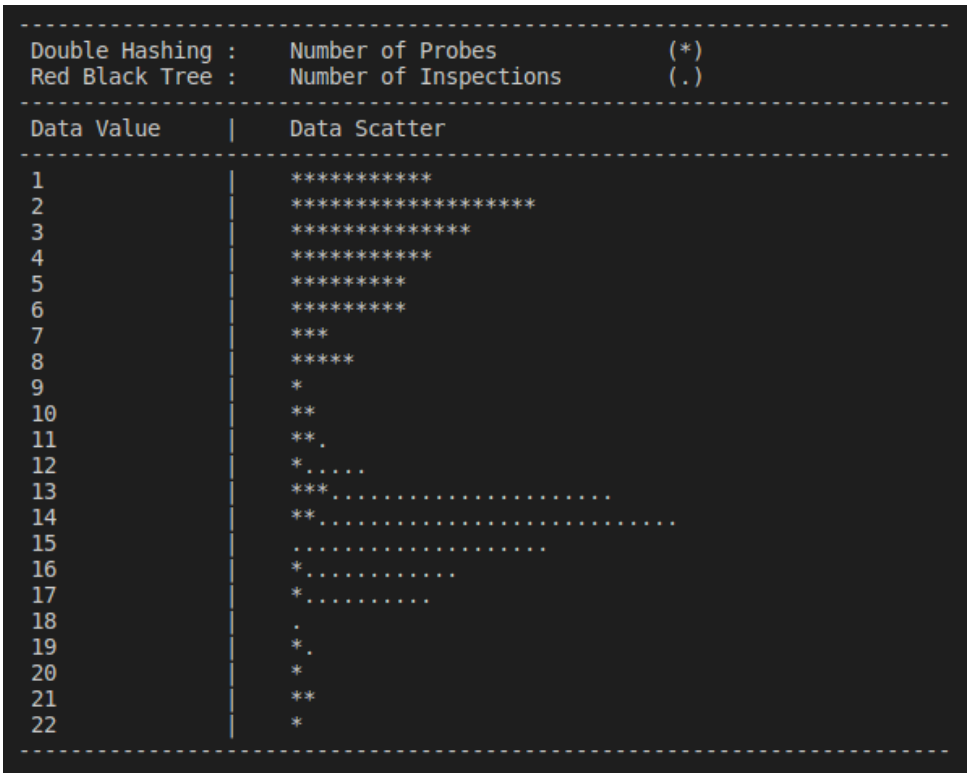


Figure 5: Visual Comparison

This simple plot generation code has been implemented by us, just using `Python` [1], without the use of any third-party library.

6 Discussion

Although in our performance evaluation, Double Hashing definitely outperformed Red Black Tree, there are scenarios where comparison can flip sides.

Some of those scenarios are mentioned below:

- Sorted Order Data Retrieval

  As Red Black Tree maintains the order of the data stored while Double Hashing does not, if we want to retrieve data in sorted order, the former should outperform the later one.

- Range Queries

  As range query requires to retrieve data from a certain range, thus emphasizing the order, this scenario should also give the Red Black Tree advantage over Double Hashing.

- Deletion Operation

  If we introduce delete operation on existing data, the Double Hashing data-structure will become more complex, which might affect the performance of the data-structure.

As we've only touched the surface researching about these data-structures, our quest for learning more about them on varieties of scenarios will continue.

References

[1] G. van Rossum, "Python." https://www.python.org/.

[2] smmehrab, "double-hashing-performance-evaluation." https://github.com/smmehrab/double-hashing-performance-evaluation.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009.

[4] Wikipedia, "Double Hashing — Wikipedia, the free encyclopedia." http://en.wikipedia.org/w/index.php?title=Double%20hashing&oldid=1106470694, 2022.

[5] Wikipedia, "Open Addressing — Wikipedia, the free encyclopedia." https://en.wikipedia.org/wiki/Open_addressing, 2022.

[6] Wikipedia, "Red Black Tree — Wikipedia, the free encyclopedia." http://en.wikipedia.org/w/index.php?title=Red%E2%80%93black%20tree&oldid=1100748306, 2022.

[7] Programiz, "Red Black Tree — Programiz." https://www.programiz.com/dsa/red-black-tree.