Empirical evaluation of ordered dictionary with ALGator

Jani Suban
89222015@student.upr.si
Faculty of Mathematics, Natural Sciences
and Information Technologies,
University of Primorska
Glagoljaška 8
SI-6000 Koper, Slovenia

ABSTRACT

This paper presents an empirical evaluation of different implementations of an abstract data structure, the ordered dictionary. The implemented data structures are Binary Search Tree, AVL Tree, Red-Black Tree, Zip Tree, Skip List and 2-3 Tree.

From the result of testing the data structures, both in with strictly increasing, which is the worst case scenario for the data structure, and random updates and query operations, it is shown that the time complexity is the same as expected.

KEYWORDS

binary search trees, AVL tree, red-black tree, 2-3 tree, zip tree, skip list, ALGator

1 INTRODUCTION

There are various approaches to implementing the abstract data structure dictionary. But all of them try to minimise the time complexity of the operations. The first approach is to use hash tables. Hash tables have an expected time complexity $O(1)$, but at the cost of losing the order of the data.

If the goal of the data structure is to preserve the order of the data, the appropriate implementation of a dictionary is a kind of link list where all elements to the left of element x are less than x and all elements to the right of x are greater than x. The best possible time complexity for implementing a dictionary is $O(\log n)$[1]. This can be achieved with Balanced Binary Search Trees (or BBST).

This paper presents an empirical comparison between different implementations of ordered dictionaries such as Balanced Binary Search Trees, a 2-3 Tree, a Binary Search Trees and a Skip List. All data structures were tested with an open source test suite ALGator [5].

2 ORDERED DICTIONARY

A dictionary is an abstract data structure that has three operations: *find* an element in the structure, *insert* an element into the structure, and *delete* an element from the structure. In addition to these three operations, the ordered dictionary has two additional operations to go through all the elements in order. The two additional operations are *next element*, which returns the next element in order, and *has next*, which returns whether a next element exists or not. In this article we will focus on the operations find, delete and insert.

There are many different ways to implement an ordered dictionary that also have a time complexity of $O(\log n)$. This paper compares the performance of the following data structures with respect to real words: Binary Search Tree, AVL Tree, Red-Black Tree, 2-3 Tree, Skip List and Zip Tree. All of the above data structures, with the exception of the Binary Search Tree, have a time complexity of $O(\log n)$, as can be seen in the Table 1. In the rest of the chapter, the most important ideas behind the implemented and tested data structures are presented and described.

[1]$\log n$ stands for $\log_2 n$ unless otherwise specified

Table 1: Time complexity of insert, delete and find for all implemented data structures

Operations	Binary Search Tree	AVL Tree	Red-Black Tree	2-3 Tree	Zip Tree	Skip List
Insert	$O(n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
Delete	$O(n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
Find	$O(n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$

2.1 Binary Search Tree

The binary search tree (or BST) is the simplest of all the data structures presented in this paper. The idea behind it is to split the data into two parts. The left part (left subtree) stores all nodes whose value is less than that of the current node, and the right part (right subtree) stores all nodes whose value is greater than that of the current node[9].

The time complexity of insertion, deletion and search operations in a binary search tree is $O(n)$. This is because the tree becomes a linked list if the data is inserted into the tree in the worst possible way, e.g. in a strictly increasing way[9]. However, if the data stored in the tree was inserted randomly, then the time complexity of all three operations is $O(\log n)$ with a high probability[7].

2.2 AVL Tree

The AVL tree is the first balanced binary search tree presented in this paper. AVL trees achieve balance by limiting the height difference between the left and right subtree to a maximum of 1, $|L.height - R.height| \leq 1$. The tree can only become unbalanced during updates. That for when the tree is updated, it checks if it is still balanced, and if not, the nodes are rotated so that the tree is balanced again [1].

The time complexity of insertion, deletion and find operation in an AVL tree is $O(\log n)$. This is because in the worst case it takes for the find operation and always for update operations to reach the leaves. Since the height difference of the subtrees is at most 1, this means that all levels except the two lowest ones are full, so that the height of the tree is at most $\log n + 1 = O(\log n)$. [1]

2.3 Red-Black Tree

The red-black tree is another balanced binary search tree where the leaf nodes are always *NILL* . The balance is achieved by colouring the nodes red or black. The colouring is done with the following roles:

(1) each node is either black or red,
(2) all leaf nodes are black,
(3) red nodes have no red children,
(4) each path from the root to the leaf has the same number of black nodes,
(5) the root is always black.

The balance is achieved by requiring that the black height (number of black nodes on the path from root to leaf) is the same for each leaf. This means that the lowest leaf is at most 2 times lower than the highest. This means for $n \geq 2^b$ (b is the black height) that for $b = O(\log n)$ and also height $h \leq 2b = O(\log n)$[2].

The time complexity of insertion, deletion and find operation in a Red-Black tree is $O(\log n)$. This is because, in the worst case, for the find operation and always for update operations is needed to reach the level before the leaves, and the height of the tree is $O(\log n)$[2].

2.4 2-3 Tree

The 2-3 tree is the only tree presented in this paper that is not a binary search tree, but a B-tree, where $B = 3$[3]. The 2-3 tree has two types of nodes: a 2 node, which has two children (left and middle) and one key, and a 3 node, which has three children (left, middle and right) and 2 keys. All leaf nodes are on the same level. The insertion in the 2-3 trees and also in the B trees is in the leaf node. If the node overflows, it has 3 keys, the node is split and the middle key is inserted into the parent node. If the parent node does not exist, the middle key becomes the new root node. Deletion is done by exchanging the value to be deleted with the minimum value in the right subtree (the middle one if the left key is deleted, or the right one if the right key is deleted). If the leaf node becomes empty, the tree must be corrected by rearranging the parent node so that it becomes a two-node, or by splitting and rotating a sibling tree [4].

This method of insertion and deletion guarantees that the leaves are always on the same level, that for the tree is balanced. Since the tree is balanced, all three operations have the time complexity of $O(\log n)$ [4].

2.5 Skip List

Skip list is the only implementation of the dictionary that is not a tree, that will be examine in this paper. Skip list is a probabilistic data structure. Probability is used to define the number of pointers to the next nodes in the list. A pointer at height x always points to the next node that has at least x pointers and skips all nodes with fewer pointers. The number of pointers is calculated with the Algorithm 1. The probability that a node has one pointer is $1/2$, two is $1/4$, ... The value max in the line 3 of the Algorithm 1 is the maximum number of pointers a node can have, and it its $\log_{1/p} n \log n$, because $p = 1/2$ [6].

Because of node skipping, the expected time complexity is $O(\log n)$ with high probability for all three operations. In the worst case, if the skipping in the skip list is small, the time complexity of all three operations is the same as for the normal link list, $O(n)$ [6].

```
   Output: Number of pointer to the next node
1  i = 1 ;
2  k = PRNG.boolean;
3  while k ∧ i< max do
4  |   i = i + 1;
5  |   k = PRNG.boolean;
6  end
7  return i
```

Algorithm 1: Algorithm for calculating the height of the node

2.6 Zip Tree

The zip tree is a probabilistic data structure. It uses the idea of the skip list to balance the tree. So each node in the tree also stores its rank value. The rank of the node is assigned in the same way as the number of pointers a node has in the skip list, that for Algorithm 1 can be used for assigning the rank of the node. A new element is always inserted as a leaf of the tree, as in all binary search trees. The next step is to correct for the rank of the node. On the way from the leaf to the root, the newly added node becomes the root of the subtree if its rank is greater than the rank of the root of the subtree. If both nodes have the same rank, the node with the smallest value becomes the new root. Deleting an element from the tree is done by finding the correct node, removing the pointer to the node and then combining the children of the node to create a new subtree. The operation of combining subtrees is cold zipping [8].

Given the ranks of the nodes, the expected time complexity is $O(\log n)$ with high probability. In the worst case, if all nodes have the same rank and the data is inserted, let say strictly increasing, the time complexity is $O(n)$ [8].

3 TESTING

The test method used in this paper is to measure the duration for performing all operations. Each test can have a different number of operations. The number of operations for test i can take the following forms:

$$\begin{aligned} N_i &= (n_I, n_F, n_D), \\ N_i &= (n_I, n_D) = (n_I, 0, n_D), \\ N_i &= (n_I, n_F) = (n_I, n_F, 0), \\ N_i &= (n_I) = (n_I, 0, 0), \end{aligned} \tag{1}$$

where n_I is the number of insert operations performed in the test, n_F is the number of find operations performed in the test and n_D is the number of delete operations performed in the test.

Similarly, each test has its own set of timers, one for each operation. The time taken to execute all the operations of test i can take

the following forms:

$$\begin{aligned} \tau_i &= (t_I, t_F, t_D) \\ \tau_i &= (t_I, t_D) = (t_I, 0, t_D) \\ \tau_i &= (t_I, t_F) = (t_I, t_F, 0) \\ \tau_i &= (t_I) = (t_I, 0, 0) \end{aligned} \tag{2}$$

where t_I represents the time taken for all n_I insert operations, t_F represents the time taken for all n_F search operations and t_D represents the time taken for all n_D delete operations. To get only the total time of a particular operation from the timer τ_i, call $\tau_i(X)$, where X is the operation for which you need the time. For example, the timer for deletion is obtained as follows: $\tau_i(D)t_D$.

As can be seen in the Equation 1 and the Equation 2, each test must always contain the insert operations, otherwise the delete and find operations cannot be performed. The tests in this paper always perform two operations, either insert and find or insert and delete. The tests are divided into two parts. In the first part of the test, all values are inserted into the data structure. In the second part of the test, either the find operation or the delete operation is performed.

3.1 ALGator

ALGator is used for testing the implemented ordered dictionaries. ALGator is a test programme introduced in the paper [5]. ALGator allows the user to implement, test and evaluate algorithms and data structures with a single tool. At the moment, only the Java programming language is supported for the implementation of the algorithms and data structures.

In ALGator, the implementation of tested algorithms and data structures is done with the help of an abstract class in which the entire test logic and the structure of the implemented algorithms or data structures are defined. Although the test logic is implemented in the abstract class, the tests to be performed are written in a separate file. The evaluation of the test results is done by allowing the user to visualise the test results in a way that best represents the results.

3.2 Testing Sequences

In this paper, two scenarios are tested. The first is the worst case, or in this case the strictly increasing sequence presented in the section 3.2.1. The second is the expected scenario or in this case the random sequence presented in the section 3.2.2.

3.2.1 Strictly Increasing Sequence. This test is to show how the data structures behave when the data is stored in the worst possible way. One way to store the data in this way is to have a strictly increasing sequence of data. The sequence is strictly increasing if for each $i < j$

$$x_i, x_j \in \mathbb{N} \wedge x_i < x_j, \tag{3}$$

where x_i and x_j are the elements stored in the data structure.

3.2.2 Random Sequence. This test is designed to show how data structures behave when data is stored in a random manner. Data stored in this way is not stored optimally, but it represents all possible ways of storing it. In this case, for each $i < j$

$$x_i, x_j \in \mathbb{N} \wedge x_i \neq x_j \tag{4}$$

where x_i and x_j are the elements stored in the data structure.

4 EVALUATION

The tests were performed on a computer with an AMD Ryzen 7 2700, with 8 cores and 16 threads, 16 GB RAM and Fedora 36 with Linux kernel 6.2.14-100. The ALGator version was 0.985, created in a Docker container running Ubuntu with Java 11.0.18. The implementation of Binary Search Tree, AVL tree[2], Red-Black Tree[3], 2-3 Tree[4], Skip list[5] and Zip tree was done in Java [6].

In order to eliminate the disturbance caused by background operation as much as possible, all tests were carried out 5 times. The tests can be divided into two groups: random insertions, deletions and find operations and strictly increasing insertions, deletions and find operations.

All tests use integers in the range $[0, 100000]$ as input for the operation. The test is performed for a different number of elements stored in the data structure. The number of elements in the data structure goes from 50 to 400 elements in steps of 50.

The small size of the test set is due to the Java stuck size while testing with ALGator. The reason for the Stuck Overflow error is the recursive implementation of the data structures.

4.1 Random Tests

The random tests were implemented using the Java library `java.util.Random`. The library was used to generate the pseudo-random numbers that were used as input for the insert, find and delete operations. At the beginning of each part of the test, the seed was set to 0.

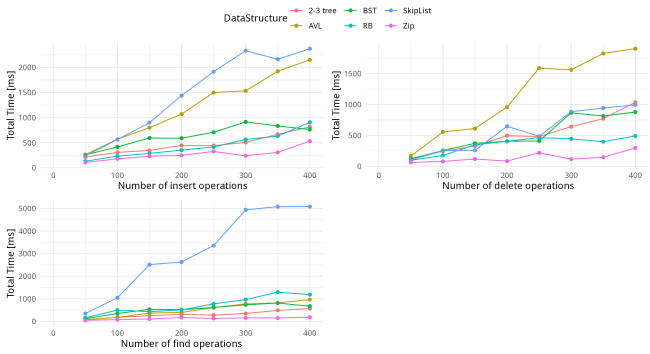

Figure 1: The time taken to perform all random insert, find and delete operations. The time is measured in milliseconds.

The results of the testing can be seen in the Figure 1. In the top left graph it can be seen the change in total time for a given number of insertions. In the top right graph it can be seen the change in total time for performing a certain number of deletions. In the bottom left graph it can be seen the change in total time for

[2]Modification of the https://www.javatpoint.com/avl-tree-program-in-java
[3]Modification of the https://algorithmtutor.com/Data-Structures/Tree/Red-Black-Trees/
[4]Modification of the https://github.com/SValentyn/2-3-tree
[5]Modification of the https://www.baeldung.com/cs/skip-lists#bd-how-to-insert-into-a-skip-list
[6]Link to the implementation https://github.com/GioGiou/BinarySearchTree

performing a certain number of find operations. Each point in the graph represents an average of all 5 runs of the same test.

From all three graphs in the Figure1, it is evident that time $O(n \log n)$. This means that the time for all operations $O(\log n)$. This result confirms that all implemented data structures have $O(\log n)$ time complexity for insert, delete and find operations when the data is randomly inserted into the structure. The high execution time for the skip list, seen mainly in bottom left graph, is due to the probability with which the skip list is balanced, but the time for all operations is still $O(\log n)$.

4.2 Strictly Increasing Tests

The strictly increasing test is the implementation of the test method presented in the section 3.2.1. Sequential numbers were chosen over a more generic strictly increasing numbers because they are easier to implement. The result is expected to be the same as for the strictly increasing sequence, since a sequence of consecutive numbers is also a strictly increasing sequence.

The strictly increasing test consists of inserting, finding and deleting n consecutive numbers. This was implemented with a counter that starts with the value 0 and increases after each operation. The numbers that are inserted into the data structure are therefore integers from 0 to $n-1$. For the deletion, the counter was initialised with the value $n-1$ and after each deletion, the counter was decreased. If the counter for the deletion was not implemented in this way, the deletion is performed in a time of $O(1)$, because the deleted element is stored in the root of the tree each time.

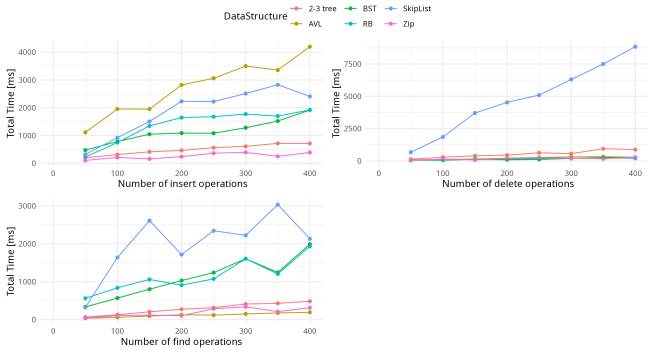

Figure 2: The time taken to perform all strictly increasing insert, find and delete operations. The time is measured in milliseconds.

The results of the testing can be seen in the Figure 2. In the top left graph it can be seen the change in total time for a given number of insertions. In the top right graph it can be seen the change in total time for performing a certain number of deletions. In the bottom left graph it can be seen the change in total time for performing a certain number of find operations. Each point in the graph represents an average of all 5 runs of the same test.

From all three graphs in the Figure 2, it is evident that time is rising $O(n \log n)$. This means that the time for all operations is $O(\log n)$. The only exception is the binary search tree, where the time increases faster, $O(n^2)$, which means that the time complexity of all tree operations is $O(n)$. This result confirms that all implemented data structures have a time complexity of $O(\log n)$ for insert, delete and find operations, except for the binary search tree, which has a time complexity of $O(n)$.

The reason why the time for inserting into the binary search tree is smaller than the time of the AVL tree and the Red Black tree is due to the rebalacing of the AVL and Red Black trees. For a larger test set, the time of the binary search tree will exceed the time of the AVL and Red-Black tree. The reason that the skip list with a time complexity of $O(\log n)$ performs worst is the lack of skipping, which can be seen especially in the top right graph in the Figure 2.

5 CONCLUSIONS AND FUTURE WORKS

This paper was presented the empirical evaluation of time complexity for different implementations of ordered dictionaries. More specifically, balanced trees (AVL, Red-Black, Zip, 2-3 tree), Skip list and binary search tree. All data structures were tested with the test suit ALGator [5]. Although the size of the test set was small, it can be seen from Figure 1 and Figure 2 that the time complexity of all data structures is $O(\log n)$, except for the binary search tree with strictly increasing insertion, deletion and find, where the time complexity is $O(n)$, as expected from the theoretically proven time complexity.

Our aim is to improve the outcome of this work in the future by increasing the size of the test sets. By doing so, we hope to obtain more accurate results and to better identify the differences in time complexity of all data structures. Furthermore, we plan to generalise the idea of Zip trees from working with binary search trees to working with k-ary search trees.

REFERENCES

[1] Georgii Maksimovich Adelson-Velskii and Evgenii Mikhailovich Landis. 1962. An algorithm for organization of information. In *Doklady Akademii Nauk*, Vol. 146. Russian Academy of Sciences, 263–266.

[2] Rudolf Bayer. 1972. Symmetric binary B-Trees: Data structure and maintenance algorithms. *Acta Informatica* 1, 4 (1972), 290–306. https://doi.org/10.1007/bf00289509

[3] R. Bayer and E. McCreight. 1970. Organization and Maintenance of Large Ordered Indices. In *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control* (Houston, Texas) *(SIGFIDET '70)*. Association for Computing Machinery, New York, NY, USA, 107–141. https://doi.org/10.1145/1734663.1734671

[4] Thomas H. Cormen, Charles Eric Leiserson, Ronald Linn Rivest, and Clifford Seth Stein. 2009. *Introduction to Algorithms* (third ed.). MIT Press. https://mitpress.mit.edu/books/introduction-algorithms-third-edition

[5] Tomaz Dobravec. 2019. Implementation and Evaluation of Algorithms with ALGator. *Informatica (Slovenia)* 43 (2019).

[6] William Pugh. 1990. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Commun. ACM* 33, 6 (jun 1990), 668–676. https://doi.org/10.1145/78973.78977

[7] Bruce Reed. 2003. The Height of a Random Binary Search Tree. *J. ACM* 50, 3 (may 2003), 306–332. https://doi.org/10.1145/765568.765571

[8] Robert E. Tarjan, Caleb Levy, and Stephen Timmel. 2021. Zip Trees. *ACM Transactions on Algorithms* 17, 4 (oct 2021), 1–12. https://doi.org/10.1145/3476830

[9] P. F. Windley. 1960. Trees, Forests and Rearranging. *Comput. J.* 3, 2 (01 1960), 84–88. https://doi.org/10.1093/comjnl/3.2.84 arXiv:https://academic.oup.com/comjnl/article-pdf/3/2/84/1358330/030084.pdf