Linnéuniversitetet
Kalmar Växjö

Report

Assignment 2

1DV516

Supervisor: Morgan Ericsson
Semester: Autumn 2023

Author1: Katarina Simakina
Email es225hi@student.lnu.se

Author2: Zejian Wang
Email zw222bb@student.lnu.se

Table of Contents

2

Task 5

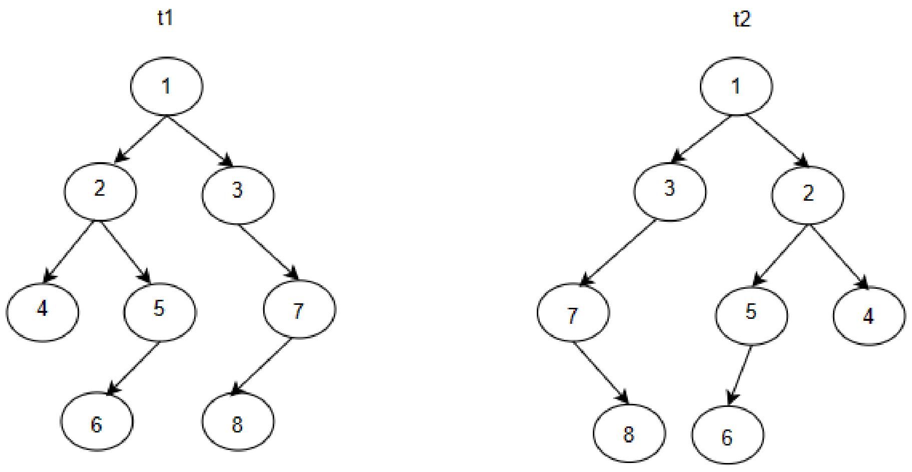
The problem 5 is an implementation of a method to check whether two binary trees are isomorphic. Two binary trees are considered isomorphic if one of them can be transformed from the other by swapping left and right children of some nodes. The analysis of this algorithm is below:

Two Binary trees (t1, t2) are two objects containing nodes of which the value of each node we set is integer in this case. They are as the passing parameters in the method(isIsomorphic(t1, t2) to check if they are isomorphic. Two trees' roots should be obtained firstly and if their roots are null, which means both are empty trees, they are isomorphic, the method returns true. This operation takes constant time, the time complexity is O(1).

If one tree is empty and the other is not, which means one tree's root is null but the other is not, two trees are not isomorphic, the method returns false. This operation also takes constant time, the time complexity is O(1).

If the value of both trees' root are the same, four trees are temporarily formed. Two subtrees(t1_left, t1_right) are formed by one tree's root's left child and right child as their roots, the other two subtrees(t2_left, t2_right) are formed by the other tree's root's left child and right child as their roots. One subtree from t1 and one subtree from t2 will be as the passing parameters to make a recursive call. For example, isIsomorphic(t1_left, t2_left), it checks the left subtree from t1 against the left subtree of t2, three other recursive calls also should be made -- isIsomorphic(t1_left, t2_right), isIsomorphic(t1_right, t2_left), isIsomorphic(t1_right, t2_right).

After all the recuisive calls are made, whether the two trees are isomorphic can be determined by satisfying the following two conditions: If t1_left and t2_left are isomorphic and t1_right and t2_right are isomorphic as well, it means that the both trees are the same, If this condition is true, it implies that the trees are isomorphic. If t1_left and t2_right are isomorphic and t1_right and t2_left are isomorphic as well, it means that two trees are isomorphic, considering the swap of left and right children. If one of the two conditions above is met, there is an isomorphic relationship between the two trees.



Two trees are manually built in main function in the code. We can take these as an example to illustrate how this algorithm works, the roots of both trees are the same, then down to the left sub tree and right sub tree, we can find that it meets one of the two conditions that the value of root of left sub tree rooted 1 is the same as the value of root of right sub tree rooted 1and the value of root of right sub tree rooted 1 is the same as the value of root of left sub tree rooted 1, but because both node 2 and 3 on both trees have children, we repeat the process to continue to check whether their children met the condition. After check all the nodes down to the bottom level of node 6 and 8 in both trees, it satisfies the conditions, then we can determine that these two trees are isomorphic.

We can draw from the above that this algorithm explores all nodes from the root down to bottom level of the tree, hence the number of recursive calls is proportional to the number of nodes(n) in the tree, which means the time complexity is O(n), the other operations we discussed at the beginning are constant, so the time complexity is O(1), which does not contribute the overall complexity. In conclusion, the time complexity of this algorithm is O(n), which means that the algorithm's performance grows linearly with the number of nodes in the input trees.

Task 6

In this task, we investigate the behavior of AVL (Adelson-Velsky and Landis) and Binary Search Trees (BST) through a series of experiments involving random insertions and deletions nodes and randomly search nodes.

The goal is to observe and compare the structural changes in these tree data structures as elements are added and removed randomly.

A total of 10 runs were conducted for each data structure to ensure consistency and reliability of the results.

To investigate AVL and BST we have done the following:

For run number 1,2,3,4,5:
-Random numbers in the range from 0 to 100 000 were generated using random.nextInt(0, 100 000) for both insertion and deletion operations.
-A loop was employed to perform 100 000 insertions and deletions in each run for both AVL and BST trees.
-After each insertion and deletion operation, the height and node count of the trees were recorded.
-The time taken for both insertion and deletion operations was measured using System.currentTimeMillis(), and these times were recorded.
-Similarly, the time taken for 100 000 search operations was measured and recorded.

For run number 6,7,8,9,10 some parameters have been changed to make the trees deeper:
--Random numbers in the range from 0 to 1 000 000 were generated using random.nextInt(0, 1 000 000) for both insertion and deletion operations.
-A loop was employed to perform 1000 000 insertions and deletions in each run for both AVL and BST trees.
-After each insertion and deletion operation, the height and node count of the trees were recorded.
-The time taken for both insertion and deletion operations was measured using System.currentTimeMillis(), and these times were recorded.
-Similarly, the time taken for 1 000 000 search operations was measured and recorded.

Table 1

Run Number	Metric	AVL Tree	Binary Search Tree
1	Nodes (After Insert)	63325	63290
	Height (After Insert)	18	41
	Nodes (After Delete)	23360	23352
	Height (After Delete)	16	38
	Time for insert and delete operation	90 milliseconds	55 milliseconds
	Time for search operation	20 milliseconds	32 milliseconds
2	Nodes (After Insert)	63277	63427
	Height (After Insert)	18	40
	Nodes (After Delete)	23169	23280
	Height (After Delete)	16	35
	Time for insert and delete operation	72 milliseconds	48 milliseconds
	Time for search operation	20 milliseconds	30 milliseconds
3	Nodes (After Insert)	63197	63098
	Height (After Insert)	18	39
	Nodes (After Delete)	23159	23181
	Height (After Delete)	17	34
	Time for insert and delete operation	55 milliseconds	52 milliseconds
	Time for search operation	15 milliseconds	19 milliseconds
4	Nodes (After Insert)	63304	63233
	Height (After Insert)	18	38
	Nodes (After Delete)	23313	23201
	Height (After Delete)	16	35
	Time for insert and delete operation	49 milliseconds	35 milliseconds
	Time for search operation	15 milliseconds	20 milliseconds
5	Nodes (After Insert)	63312	63159
	Height (After Insert)	18	38
	Nodes (After Delete)	23278	232332
	Height (After Delete)	16	32
	Time for insert and delete operation	50 milliseconds	38 milliseconds
	Time for search operation	12 milliseconds	20 milliseconds
6	Nodes (After Insert)	631858	631790
	Height (After Insert)	22	45
	Nodes (After Delete)	232323	232389
	Height (After Delete)	21	42
	Time for insert and delete operation	970 milliseconds	610 milliseconds
	Time for search operation	250 milliseconds	275 milliseconds
7	Nodes (After Insert)	632162	631351

	Height (After Insert)	22	49
	Nodes (After Delete)	232602	232462
	Height (After Delete)	20	45
	Time for insert and delete operation	915 milliseconds	626 milliseconds
	Time for search operation	270 milliseconds	359 milliseconds
8	Nodes (After Insert)	631916	631933
	Height (After Insert)	23	47
	Nodes (After Delete)	232508	232648
	Height (After Delete)	21	42
	Time for insert and delete operation	1095 milliseconds	764 milliseconds
	Time for search operation	296 milliseconds	362 milliseconds
9	Nodes (After Insert)	632284	632416
	Height (After Insert)	22	47
	Nodes (After Delete)	232116	232805
	Height (After Delete)	21	43
	Time for insert and delete operation	888 milliseconds	748 milliseconds
	Time for search operation	392 milliseconds	400 milliseconds
10	Nodes (After Insert)	632029	632492
	Height (After Insert)	22	48
	Nodes (After Delete)	233098	232595
	Height (After Delete)	20	45
	Time for insert and delete operation	1054 milliseconds	855 milliseconds
	Time for search operation	301 milliseconds	440 milliseconds

Conclusions based on the results from the Table 1:

1. In AVL trees, insertion and deletion operations take more time compared to the same operations in BST tree. This is primarily because AVL trees are self-balancing and after inserting or deleting a node, the tree must be rebalanced through rotations (single or double rotations).

In the case of an unbalanced BST tree (where no efforts are made to maintain balance), insertion and deletion operations tend to be faster than in AVL trees. This is because no rebalancing is performed and nodes are simply added or removed without considering the balance of the tree. As a result, the tree may become skewed and inefficient for search operations, but insertions and deletions are relatively straightforward and quicker.

2. Based on the results we see that search operation in AVL tree goes quickly and takes less time than in unbalanced BST.

The reason is that AVL trees are self-balancing binary search trees, which means they maintain a balanced structure by design. This balance ensures that the height of the tree remains logarithmic in relation to the number of nodes. As a result, search operations in AVL trees have a time complexity of O(log n). This logarithmic time complexity is significantly faster than linear time complexity (O(n)) seen in unbalanced trees.

3. Comparing height of the AVL tree and Binary Search Tree we see definitely difference. For example, in Run1: height of AVL tree is 16, height of BST is 38. The primary reason for this difference is the self-balancing property of AVL trees, which ensures that the tree remains balanced after every insertion or deletion operation. In contrast, BSTs degenerate into unbalanced structures depending on the order of insertions and deletions.

Analysis of AVL Tree Height.

Table 2

Number Run	Number of nodes after deleting	AVL height based on calculations	AVL height-based on the output
1	23 360	20	16
2	23169	20	16
3	23159	20	17
4	23313	20	16
5	23278	20	16
6	232323	24	21
7	232602	24	20
8	232508	24	21
9	232116	24	21
10	233098	24	20

To perform an analysis of the tree height, we utilize the formula *1.44*log(N + 2) − 1.328*. This formula is commonly employed to estimate the height of an AVL tree based on the number of nodes (N) in the tree. As well this formula provides a theoretical upper bound for the height.

For a more accurate and meaningful analysis, we have chosen to evaluate the tree's height after the deletion operation. This approach ensures that our analysis takes into account the effects of both node insertion and deletion on the tree's structure.

Run 1: we observed that the number of nodes remaining after the delete operation was 23,360. To calculate the estimated height using the formula 1.44*log(N + 2) − 1.328:

$\log(23\,360 + 2) \approx \log(23\,362) \approx 14.512$

Multiply the result by 1.44:

$1.44 * 14.512 \approx 20.89$

Subtract 1.328:

20.89- 1.328 ≈ 19.56

So, 1.44 * log(23 362 + 2) - 1.328 ≈ 19.56 ≈ 20

Run 2:

Calculate the natural logarithm (base e) of (23 169 + 2):

log(23 169+ 2) ≈ log(23171) ≈ 14.50

Multiply the result by 1.44:

1.44 * 14.50 ≈ 20.88

Subtract 1.328:

20.88 - 1.328 ≈ 19.552

So, 1.44 * log(23171) - 1.328 ≈ 19.552≈ 20

Run 3:

We have done the same actions for the calculation which were described above in Run1 and Run2.

1.44 * log(23 159 + 2) - 1.328 ≈ 19.552 ≈ 20

Run 4:

1.44 * log(23 313 + 2) - 1.328 ≈ 19.566 ≈ 20

Run 5:

1.44 * log(23 278 + 2) - 1.328≈ 19.562 ≈ 20

Run 6:

1.44 * log(232323 + 2) - 1.328≈ 24.341 ≈ 24

Run 7:

1.44 * log(232602 + 2) - 1.328 ≈ 24.344 ≈24

Run 8:

1.44 * log(232508 + 2) - 1.328 ≈ 24.342 ≈ 24

Run 9:

1.44 * log(232116+ 2) - 1.328 ≈ 24.338 ≈ 24

Run 10:

1.44 * log(233098+ 2) - 1.328≈ 24.347 ≈ 24

Analys of the results from the Table 2.

Our results from the calculations and practice results from output (after running) code slightly differentiate. For example, in Run 1, the calculated height is 20, while the observed height is 16. This difference is expected. Firstly, the formula provides an upper bound estimate, the actual height observed in practice may vary. Secondly, the height of AVL trees can be influenced by various factors such as the order in which nodes are inserted and deleted, specific dataset.

Calculating the minimum number of nodes (S(h)) in an AVL tree of height h.

Table 3.

Number Run (Table1)	Height	Minimum number of the nodes	Number of the Nodes in the AVL tree (after deleting)
1	16	4182	23 360
3	17	6767	23159
7	20	28656	232602
9	21	46367	232116

To calculate this parameter we have used the following formula: $S(h) = S(h - 1) + S(h - 2) + 1$. It provides initial values for S(0) and S(1).

Explanations for our calculations using the height of the tree.

Height is 16.
$S(0) = 1$
$S(1) = 2$
$S(2) = S(1) + S(0) + 1 = 2 + 1 + 1 = 4$
$S(3) = S(2) + S(1) + 1 = 4 + 2 + 1 = 7$
$S(4) = S(3) + S(2) + 1 = 7 + 4 + 1 = 12$
$S(5) = S(4) + S(3) + 1 = 12 + 7 + 1 = 20$
$S(6) = S(5) + S(4) + 1 = 20 + 12 + 1 = 33$
$S(7) = S(6) + S(5) + 1 = 33 + 20 + 1 = 54$
$S(8) = S(7) + S(6) + 1 = 54 + 33 + 1 = 88$
$S(9) = S(8) + S(7) + 1 = 88 + 54 + 1 = 143$
$S(10) = S(9) + S(8) + 1 = 143 + 88 + 1 = 232$
$S(11) = S(10) + S(9) + 1 = 232 + 143 + 1 = 376$
$S(12) = S(11) + S(10) + 1 = 376 + 232 + 1 = 609$
$S(13) = S(12) + S(11) + 1 = 609 + 376 + 1 = 986$
$S(14) = S(13) + S(12) + 1 = 986 + 609 + 1 = 1597$
$S(15) = S(14) + S(13) + 1 = 1597 + 986 + 1 = 2584$
$S(16) = S(15) + S(14) + 1 = 2584 + 1597 + 1 = 4182$

Height is 17.
S(0) = 1
S(1) = 2
S(2) = S(1) + S(0) + 1 = 2 + 1 + 1 = 4
S(3) = S(2) + S(1) + 1 = 4 + 2 + 1 = 7
S(4) = S(3) + S(2) + 1 = 7 + 4 + 1 = 12
S(5) = S(4) + S(3) + 1 = 12 + 7 + 1 = 20
S(6) = S(5) + S(4) + 1 = 20 + 12 + 1 = 33
S(7) = S(6) + S(5) + 1 = 33 + 20 + 1 = 54
S(8) = S(7) + S(6) + 1 = 54 + 33 + 1 = 88
S(9) = S(8) + S(7) + 1 = 88 + 54 + 1 = 143
S(10) = S(9) + S(8) + 1 = 143 + 88 + 1 = 232
S(11) = S(10) + S(9) + 1 = 232 + 143 + 1 = 376
S(12) = S(11) + S(10) + 1 = 376 + 232 + 1 = 609
S(13) = S(12) + S(11) + 1 = 609 + 376 + 1 = 986
S(14) = S(13) + S(12) + 1 = 986 + 609 + 1 = 1597
S(15) = S(14) + S(13) + 1 = 1597 + 986 + 1 = 2584
S(16) = S(15) + S(14) + 1 = 2584 + 1597 + 1 = 4182
S(17) = S(16) + S(15) + 1 = 4182 + 2584 + 1 = 6767

Calculations for the height 20 and 20 were done in the same way which were described above for the height 16 and 17.

We get difference between our calculation of the minimum of the number nodes and practical output when we run the program. For example, minimum number of the nodes for the AVL tree for the height 16 is 4 182, in practice from our output when we run the program, we got 23 362.

The reasons why we got the difference in the results:

Firstly, it is randomness. In our program, we insert and delete nodes randomly. The final structure of the AVL tree will depend on the specific random sequence generated. Since AVL trees strive to maintain balance, the order of insertions and deletions can significantly impact the tree's structure. Theoretically calculated values assume a specific structure, while our program introduces randomness.

Secondly, it is initial conditions. The theoretical calculations for S(0) and S(1) assume specific initial conditions. However, our program doesn't guarantee that the tree starts with these exact conditions. The initial state of the tree, influenced by the random insertion and deletion process, can lead to variations.

Thirdly, it is algorithm implementation. Differences in the implementation details of AVL tree operations, such as balancing and rotations, can affect the tree's structure and, consequently, the number of nodes. Minor variations in implementation can lead to different results.

In addition, it is data size. In our code inserts and deletes nodes in the range from 0 to 100 000 and from 0 to 1 000 000, which is a relatively small dataset compared to the theoretical model, which considers the tree's behavior with larger datasets. AVL trees tend to demonstrate their balancing benefits more effectively with larger datasets.

Time Complexity.

For balanced in our case we consider AVL tree time complexity for all operation (insert, delete and search) is O(log n). The O(log n) notation tells you that as the number of nodes increases, the time for operations should grow logarithmically, which is generally a good performance characteristic for data structures.

The time complexity is usually described in terms of big O notation, which provides an upper bound on the time complexity as a function of the input size (in this case, the number of nodes).

Time complexity AVL Tree.

Table 4

Number of Run (Table 1)	Operation	Number of nodes	Time complexity AVL Tree O(log n)
1	Delete nodes	23360	O(log 23360) ≈ O(14)
	Insert Nodes	63325	O(log 63325) ≈ O(16)
2	Delete nodes	23169	O(log 23360) ≈ O(14)
	Insert Nodes	63277	O(log 63327) ≈ O(16)
6	Delete nodes	232323	O(log 232323) ≈ O(17)
	Insert Nodes	631858	O(log 631858) ≈ O(19)
7	Delete nodes	232602	O(log 232602) ≈ O(18)
	Insert Nodes	632162	O(log 632162) ≈ O(20)

In accordance our calculations we conclude that the time complexity for AVL tree operations, including insertions and deletions, exhibits a logarithmic growth pattern relative to the number of nodes (n) in the tree. Across multiple runs (Runs 1, 2, 6, and 7), the time complexity for AVL tree operations remains approximately logarithmic. This consistency demonstrates the stability and reliability of AVL trees in maintaining balanced structures.

<u>For BST which is unbalanced the time complexity for all operations is O (n).</u>

<u>Time complexity BST Tree</u>

Table 5

Number of Run (Table 1)	Height (after delete)	Time for insert/delete operation (ms)	Time for search operation (ms)
2	35	48	30
1	38	55	32
6	42	610	275
7	45	626	359

The data which is provided in Table 5 clearly demonstrates an unbalanced Binary Search Tree (BST), where the height is not constrained, the worst-case time complexity for all operations (insertion, deletion, and search) can indeed become O(n).

The height of the tree increases from 35 to 45, indicating increasing imbalance.

In the first two runs, where the height after deletion is 38 and 35 respectively, the time for insert/delete operations and search operations are relatively high compared to a balanced BST. This indicates that the time complexity is approaching O(n).

In the run 6 and 7 where the height after deletion is even higher (42 and 45), the time for operations becomes significantly higher, reinforcing the O(n) time complexity.

In an unbalanced BST, the tree's structure does **not** guarantee logarithmic height, so as the height of the tree increases with more insertions and deletions, the time complexity grows linearly with the number of nodes (O(n)).

<u>In case BST (unbalanced) is an average tree which has height O (√n).</u>

Table 6

Number of Run (Table 1)	Operation	Number of nodes after the operation	height O (√n)
1	Delete nodes	23360	O((√23360) ≈ O(152)
	Insert Nodes	63325	O(√ 63325) ≈ O(251)
2	Delete nodes	23169	O(√ 23169) ≈ O(152)
	Insert Nodes	63277	O(√ 63327) ≈ O(251)
6	Delete nodes	232323	O(√ 232323) ≈ O(481)

	Insert Nodes	631858	O(√ 631858) ≈ O(794)
7	Delete nodes	232602	O(√ 232602) ≈ O(482)
	Insert Nodes	632162	O(√ 632162) ≈ O(795)

In the table 6 we calculated the estimated heights of the unbalanced Binary Search Tree (BST) for various runs based on the number of nodes using the formula O(√n). This is a reasonable way to estimate the average height of an unbalanced BST when insertions and deletions are performed randomly.

It is worth noting that O(√n) represents an upper bound on the average height of the unbalanced BST, which means the actual height may vary but is not expected to grow significantly faster than the square root of the number of nodes.

Our calculations show that the height increases as the number of nodes grows and the growth is roughly proportional to the square root of the number of nodes.