COMP47500 – Advanced Data Structures in Java

Assignment No: 2

Date: 18/03/24

Title: Leveraging AVL Tree to manage Contact details in Contact Management System (CMS)

Code Link : https://github.com/COMP47500-Adv-Data-Structures-in-Java/Assignment_2

Execute: Main.java inside ie/ucd/csnl/comp47500

Assignment Type of Submission:			
Group	Yes/No	**List all of group members' details:**	**% Contribution** **Assignment Workload**
	Yes	Student Name : Prakash Jha Student ID : 2320077	25%
		Student Name: Satish Gaikwad Student ID: 23204142	25%
		Student Name: Chungman Lee Student ID: 23205535	25%
		Student Name: Abhijith Sathyendran Student ID: 23200124	25%

1. **Problem Domain Description:**

This Contact Management System(CMS) is aiming to create, search, amend, and remove contacts effectively, especially for large parties who need to deal with large contacts of the members.

AVL tree data structure has been chosen as a data structure for this problem. There are some reasons why we chose this data structure for this problem. First of all, the AVL tree's time complexity for searching is O(log n). This makes the complete performance efficient and quick, especially in big data. Secondly, AVL trees keep the data balanced after insertion and deletion, so increase the efficiency of management. Furthermore, AVL trees have a stable performance despite the huge number of contacts.

Key Features:

- Contact Information Management:
  Contact information is managed through the ContactImpl class. This class contains ID, name, phone number, e-mail address, photo URI, additional email address, and favorite.

- Dynamic Contact Information Search:
  The user can search the contact information by providing the name the user wants to know. SearchContact class returns a list of all the contacts that correspond to the name by using AVL tree.

- Contact Information Insertion and Deletion from the File:
  Main class provides methods that read contacts from the file, and insert and delete it from the tree so that the system can deal with the large data insertion and deletion.

- Performance Measurement and Tree Information(size, height, balance)

2. Theoretical Foundations of the Data Structure(s) utilised

AVL trees are self-balancing binary search trees. This guarantees a balance between the left and right subtrees of each node. This ensures that the tree has a logarithmic height in the number of nodes which make it efficient to search, insert, and delete the data.

Concepts of AVL Tree

1. Binary Search Tree(BST)

- Specific type of BST.
- Each node has a value greater than all its left subtree's nodes and less than all its right subtree's nodes.
- Efficient search by traversing tree based on comparisons with node value

2. Balance Factor:

- Specific type of BST.
- Each node has a value greater than all its left subtree's nodes and less than all its right subtree's nodes.
- Efficient search by traversing tree based on comparisons with node value

2. Balance Factor:

- The balance factor of a node is the difference between the heights of its left and right subtrees.
- A perfectly balanced tree has a balance factor of 0 for every node.
- AVL trees define a stricter balance constraint: the balance factor of any node must be within -1 and +1 (inclusive).

3. Rotations:

- AVL trees maintain their balance through rotations.
- Rotations are local operations performed on a subtree to adjust the heights of its nodes and restore the balance factor.
- There are four basic rotation types in AVL trees:
    - Left Rotation: Performed when the right subtree is heavier.
    - Right Rotation: Performed when the left subtree is heavier.
    - Left-Right Rotation: Combination of a left rotation on the child and a right rotation on the parent.
    - Right-Left Rotation: Combination of a right rotation on the child and a left rotation on the parent.
- We used Left and Right rotations and used these both for balancing by case.

4. Search Time Complexity:

Due to the guaranteed balance, AVL trees have a search time complexity of O(log n) in the average and worst cases, where n is the number of nodes. This is faster than a BST's average case of O(log n) and worst case of O(n).

5. Insertion and Deletion Time Complexity:

Insertion and deletion in AVL trees have a time complexity of O(log n) on both average and worst case, due to the efficient balancing mechanism. This is slower than a perfectly balanced BST (O(1)), but ensures consistent performance regardless of the order of insertions or deletions.

6. The operations available in AVLTree:

- insert(T data): Insert data in the tree sent as a parameter in the tree and perform rotations that maintain the balance of the tree.
- delete(T data): Search and delete an element in the tree sent as a parameter. For the balance, rotations can be performed after the deletion.
- contains(T data): Return if the AVL tree contains a certain data.
- getHeight(): Return the height of the AVL tree. The height information is stored in the node.
- inOrderTraversal(): Perform in-order traversal of the tree and return a list containing all elements in sorted order.
- clear(): Remove all the nodes from the tree. Possibly require traversing all nodes.
- size(): Return the number of the nods. The size information is stored in the tree.
- isBalanced(): Check if the tree is balanced. It’s always true because of the automatic rotation.

Time complexity table for AVL Tree operations:

insert(T data)	O(log n)
delete(T data)	O(log n)
contains(T data)	O(log n)
getHeight()	O(1)
inOrderTraversal()	O(n)
clear()	O(n)
size ()	O(1)
isBalanced()	O(1)

3. Analysis/Design (UML Diagram(s))

Sequence Diagram

The flow is following:

1. User starts adding contacts
2. Main class starts a loop to parse the text file lines and instantiate contact objects
3. Each contact inserts in the tree and processes insert/rebalancing automatically.
4. Main class search using SearchContact class to find a contact that the user typed.
5. SearchContact performs a tree traversal to find matching contacts that will be returned to the user.

Class Diagram

Main: Reference of AVL tree, methods for insertion/deletion form the file

AVLTreeImpl: Methods for dealing with root nodes and trees, and for checking the status of the tree such as height, root, and balance. Implementation of the AVLTree interface.

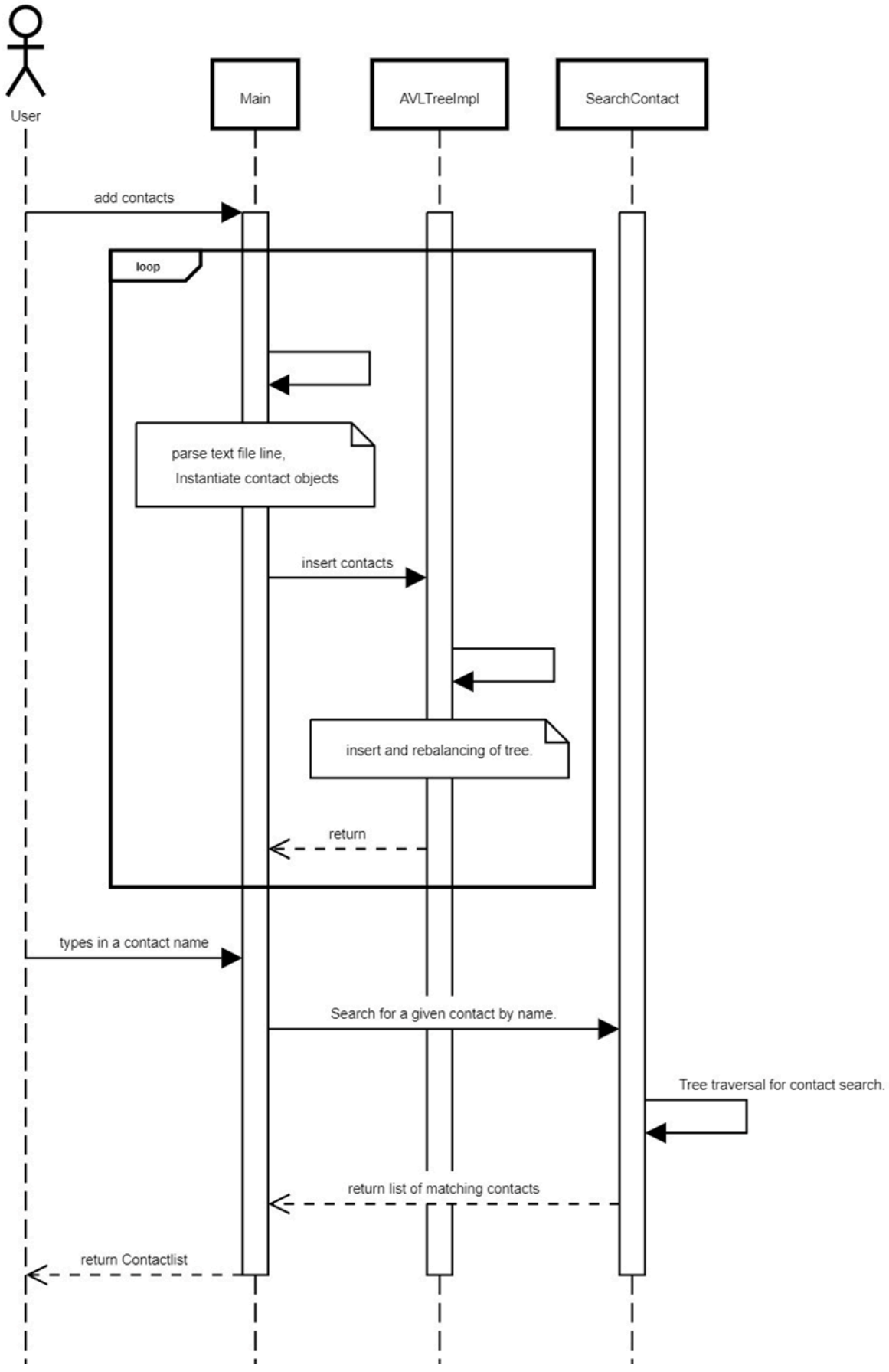
Node: Reference of the left and right child nodes, height of the node. Component of AVLTreeImpl<Contact>

Contact: Definitions of the contacts.

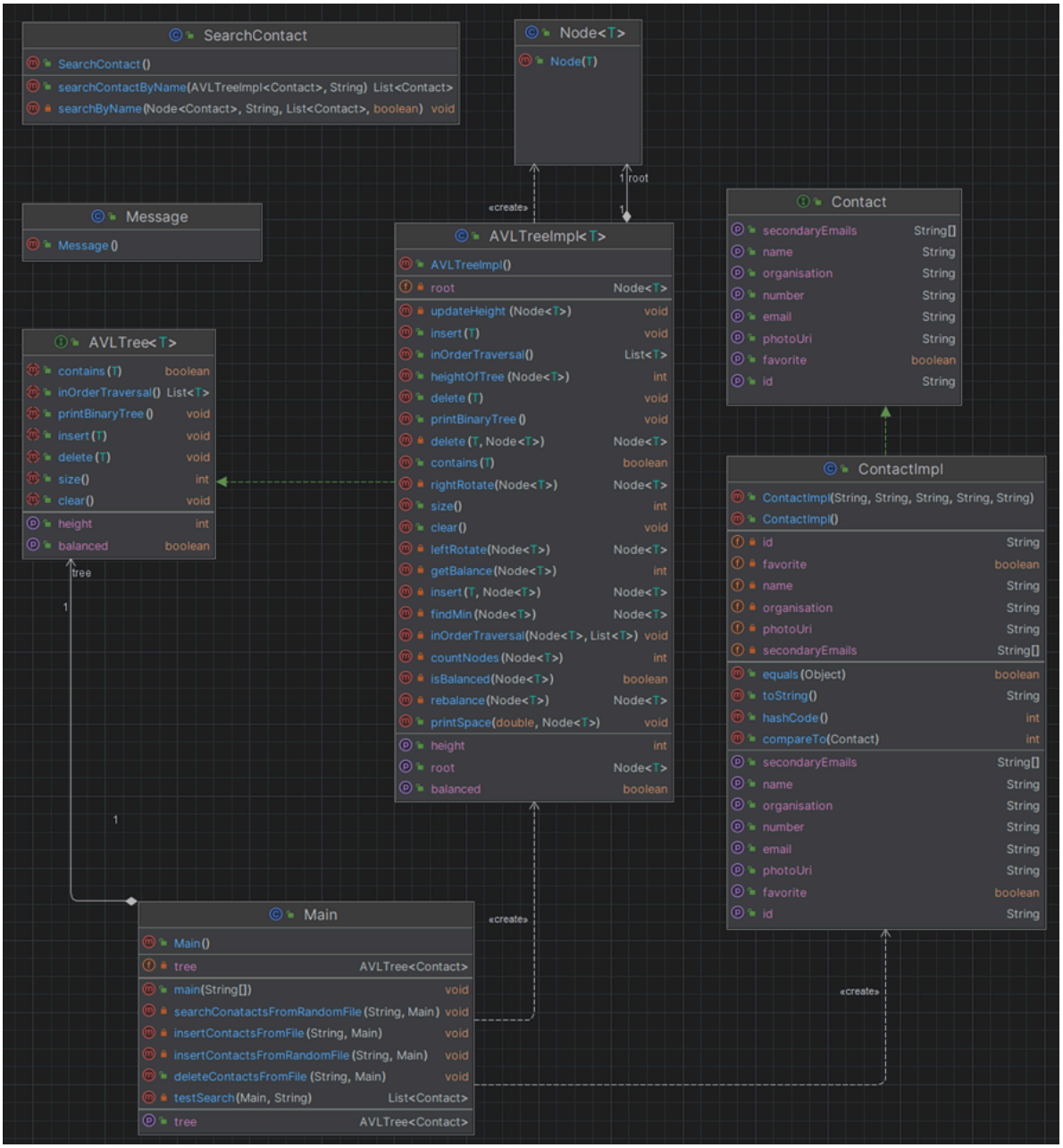
ContactImpl: implantation of the Contact interface. Provide equals(Object), hashCode(), toString(), and compareTo methods.

SearchContact: perform search in the AVL tree(serachContactByName)

Contacts app



Class Diagram



4. Code Implementation

GitHub (link): https://github.com/COMP47500-Adv-Data-Structures-in-Java/Assignment_2

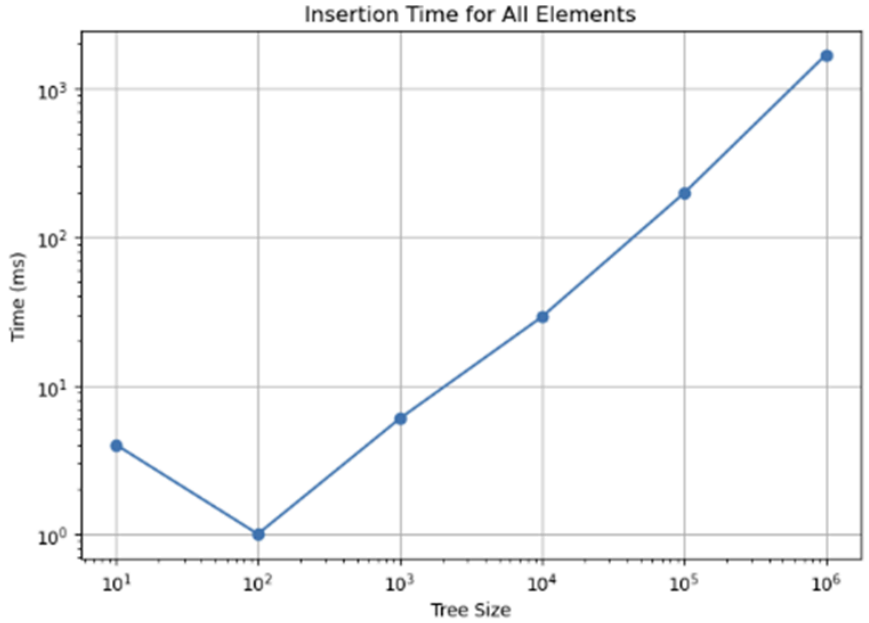
5. Set of Experiments run and results

Table: Time for each operation:

As seen from the below table, height and insertion time, increase noticeably based on tree size, but for insertion, deletion, and search for the next 25 elements, there is not a huge difference noticed as the input size is only 25 and if we consider the operation of insertion, search and delete it is O(log(n))

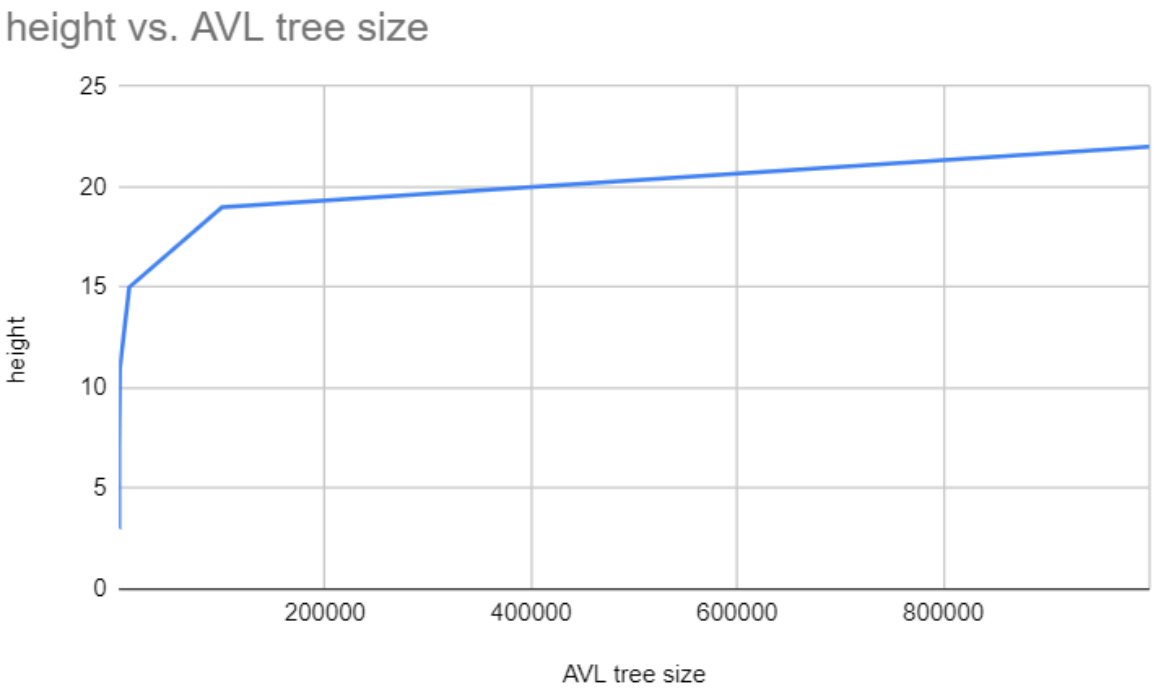
Tree Size	Height	Insertion Time (all elements)	Insertion Time (25 elements)	Search Time (25 elements)	Delete Time (25 elements)
10	3	4ms	2ms	16ms	0ms
100	7	1ms	0ms	1ms	0ms
1000	11	6ms	0ms	0ms	0ms
10000	15	29ms	1ms	0ms	0ms
100000	19	197ms	0ms	1ms	0ms
1000000	22	1687ms	0ms	0ms	0ms

Graph 1:



comment: This graph for total insertion time shows a general increase in time takes as the tree size grows.

Graph 2:



comment: This graph shows the AVL tree size vs height, as we can see it shows a logarithmic trend

6. Video of the Implementation running

Recording link: https://drive.google.com/file/d/1PWOMv9bTlBU-GTQzvFO7HwLKpXg6_yhY/view?usp=sharing

Comments: uploaded video recording in google drive.