**Department of Computer Science and Engineering**

**19CSE212 – Data Structures and Algorithms**

**Batch:** 2021 – 2025

**Semester:** 4



**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Case Study:** Hybrid Data Structure

Union-Find Disjoint Set with AVL Tree

Team Name: Hash Heroes

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Team Members:**

|  |  |  |
| --- | --- | --- |
| Sl. No | Roll No. | Name |
| 1. | **CB.EN.U4CSE21401** | P.ABHIRAM |
| 2. | **CB.EN.U4CSE21414** | C.SAHITHEE VAIBHAV |
| 3. | **CB.EN.U4CSE21430** | K.MAHAMMAD SAMI |
| 4. | **CB.EN.U4CSE21439** | M.RAHUL |
| 5. | **CB.EN.U4CSE21458** | M.SRI MANJUNADH |

**Table of Contents**

|  |  |  |
| --- | --- | --- |
| **S.NO** | **Content** | **Page No** |
| 1. | Introduction | 3 |
| 2. | Advantages of our data structure | 4-5 |
| 3. | Implementation details | 5-8 |
| 4. | Design choice | 8-9 |
| 5. | Practical applications | 9-10 |
| 6. | Assumed scenarios | 10-11 |
| 7. | Performance analysis, space complexity, performance  comparison | 11-12 |
| 8. | Experimental Setup and Methodology | 12-14 |
| 9. | Discussion | 14-15 |
| 10. | Conclusion | 15 |
| 11. | References | 15 |

**Union-Find Disjoint Set with AVL Tree**

Introduction:

Hybrid data structures are a fusion of multiple data structures, leveraging their individual strengths to efficiently address specific problem-solving requirements. By integrating diverse data structures, hybrid structures strive to strike a balance between time complexity, space efficiency, and overall performance. These versatile data structures play a pivotal role in tackling complex problems by offering specialized solutions that are tailored to meet the demands of specific scenarios.

In this project, the objective is to design and implement a hybrid data structure called Union-Find Disjoint Set with AVL Tree. The primary purpose of this hybrid structure is to efficiently handle disjoint sets and perform union and find operations on them. The Union-Find Disjoint Set data structure is commonly used to solve problems related to connectivity and graph algorithms.By combining the Union-Find data structure with an AVL Tree, which is a self-balancing binary search tree, we can enhance the performance of the Union-Find operations. The AVL Tree ensures that the overall structure remains balanced, leading to improved time complexity for operations such as find and union and also searching will be more easy with the help of AVL trees.

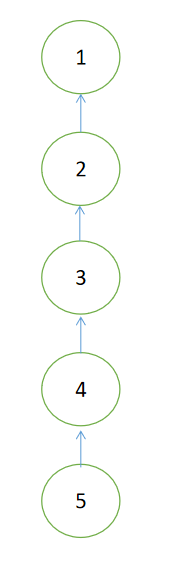
The practical applications of the hybrid data structure are may include social network analysis, image segmentation, some software scenarios , maze generation, and various graph algorithms like Kruskal's algorithm for minimum spanning trees. Where time complexity and space complexity will be improved with the help of the hybrid data structure More information is discussed in the further part of the report

Overview of chosen data structure:

* The Union-Find Disjoint Set is a data structure that maintains a collection of disjoint sets supports two operations: union and find. The union operation merges two sets, while the find operation determines which set an element belongs to. It is commonly employed in problems related to connectivity and graph algorithms. The Union-Find Disjoint Set provides a straightforward implementation for managing sets, but it can suffer from performance issues, especially when the set sizes grow large or when the trees become unbalanced.
* The AVL Tree is a self-balancing binary search tree that maintains a balance between the left and right subtrees. It supports efficient time complexity of O(log n) for search, insertion, and deletion operations. It achieves this by automatically adjusting the tree's structure to maintain a balance factor. This balance factor guarantees efficient search and retrieval operations, resulting in improved time complexity for find and union operations.

The advantages of using this hybrid data structure include:

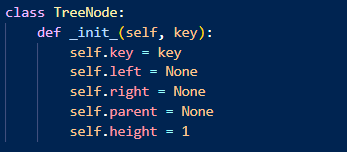
* Improved Time Complexity: The AVL Tree component of the hybrid structure helps maintain balance, resulting in faster find operations. This is particularly beneficial in scenarios where there are frequent find operations on large disjoint sets.
* Efficient Union Operations: The combination of Union-Find Disjoint Set with the AVL Tree allows for efficient union operations. The self-balancing property of the AVL Tree helps to merge two sets and maintain the balance of the resulting structure.
* Space Efficiency: The hybrid data structure optimizes space utilization by only storing necessary information. The Union-Find Disjoint Set component efficiently represents disjoint sets, while the AVL Tree component only stores the necessary data for balancing and searching.
* Self-Balancing: The AVL Tree component ensures that the tree remains balanced, preventing the worst-case scenario of a degenerate tree/skew trees , which could lead to inefficient operations and huge time complexity.In skew trees,it takes O(N) for insertion,deletion and searching.Which can be improved by using AVL Trees to prevent the formation of skew trees which
* makes complexity o(logn)

skew tree view

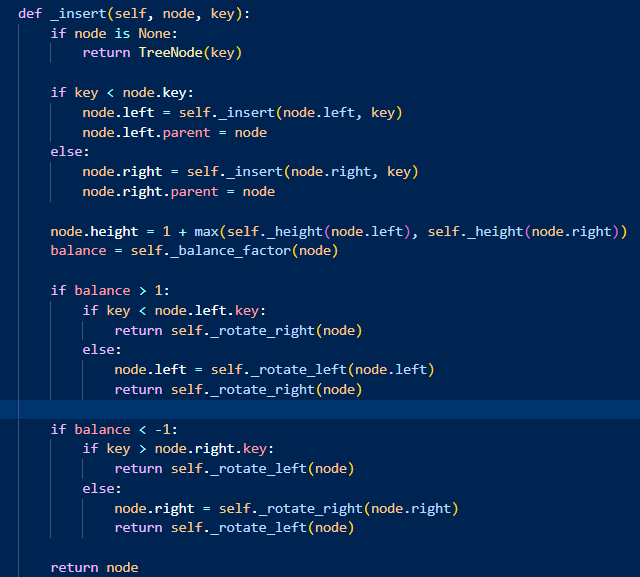
* Versatility: The hybrid structure can be applied to various problem domains that involve disjoint sets and require efficient union and find operations. It finds applications in graph algorithms, connectivity problems, social network analysis, and other scenarios where disjoint set operations are essential.

**Implementation Details:**

Code Summarization:



Part1:AVL Tree Implementation: The code starts with the implementation of the AVL Tree data structure. It defines a TreeNode class representing each node in the AVL Tree. The AVLTree class is responsible for managing the AVL Tree operations like insertion, deletion, rotation, and balancing.

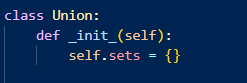


The \_insert function performs recursive insertion of nodes while maintaining the balance of the AVL Tree. The \_delete function handles the deletion of nodes while ensuring the balance factor is maintained. Additionally, there are functions to calculate heights, balance factors, and perform rotations (left and right) to maintain the AVL Tree's balanced nature.

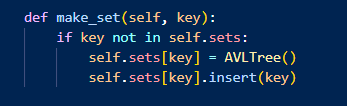
Code part2:

Code Summarization:

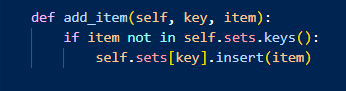
The Union class is defined, and it has an \_\_init\_\_ method that initializes the sets attribute as an empty dictionary.



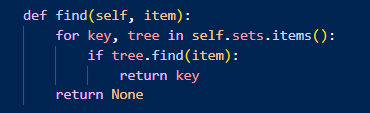
make\_set method takes a key parameter and creates a new set if it doesn't already exist. It uses the key as the identifier for the set and creates an AVLTree object to represent the set. The insert method of the AVLTree is used to insert the key into the set.



add\_item method takes a key and an item parameter. It checks if the item is already a key in the sets dictionary. If not, it inserts the item into the AVLTree associated with the given key

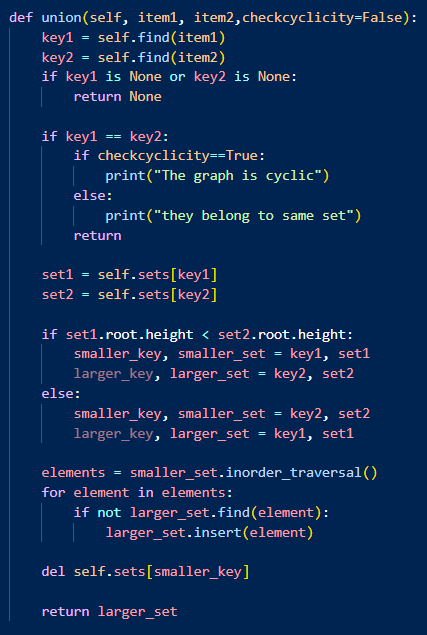


find method takes an item parameter and iterates over the sets dictionary. It calls the find method of each AVLTree to check if the item exists in any of the sets. If found, it returns the corresponding key (set identifier). If the item is not found in any set, it returns None.



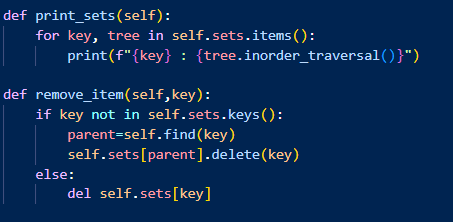
union method takes two items (item1 and item2) and an optional parameter checkcyclicity (defaulted to False). It first finds the keys (key1 and key2) of the sets containing item1 and item2 using the find method. If either key is not found, indicating that one of the items doesn't belong to any set, it returns None.

If both items belong to the same set (i.e., key1 equals key2), it checks the value of checkcyclicity. If checkcyclicity is True, it prints "The graph is cyclic." Otherwise, it prints "They belong to the same set." In both cases, it returns without performing any further operations.If the items belong to different sets, it compares the heights of the AVLTree objects associated with the keys to determine which set is smaller. The items from the smaller set are inserted into the larger set using the insert method of the AVLTree. Then, the AVLTree of the smaller set is removed from the sets dictionary. Finaly, It returns the larger set which is union of the two sets.



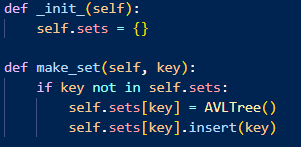
print\_sets method iterates over the sets dictionary and prints the key (set identifier) along with the elements in the corresponding AVLTree, obtained using the inorder\_traversal method.

The remove\_item method takes a key parameter and checks if it exists in the sets dictionary. If not, it finds the parent set of the key using the find method and deletes the key from that set using the delete method of the AVLTree. If the key is found in the sets dictionary, it is simply deleted from there.



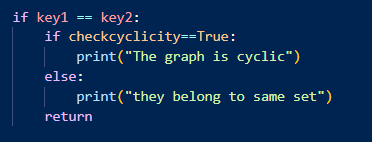
Design Choice:

1. **Integration of AVL Tree with Union-Find Disjoint Set**: The design choice of integrating an AVL Tree with the Union-Find Disjoint Set aims to improve the efficiency of find and union operations. The AVL Tree provides a balanced binary search tree structure, which ensures that these operations have a lower time complexity compared to a traditional Union-Find Disjoint Set implementation without balancing.
2. **Use of Dictionary to Store Sets**: The Union class utilizes a dictionary to store sets, where the keys represent the parent nodes, and the values are AVL Trees representing the disjoint sets. This design choice allows for efficient retrieval of sets based on the parent nodes, enabling quick access to the corresponding AVL Trees and their associated operations.

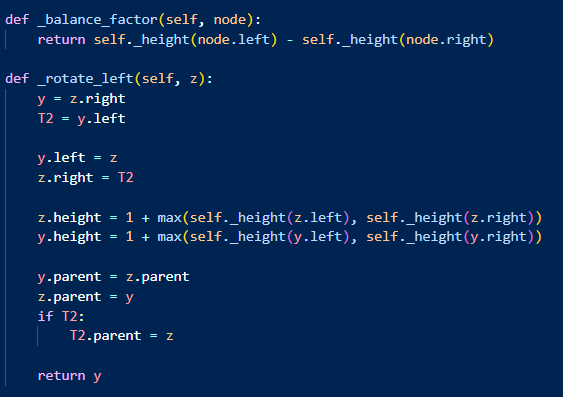


1. **Storage of Sets as AVL Trees:** The Union-Find Disjoint Set component of the hybrid data structure stores sets as AVL Trees. This design choice allows for efficient handling of disjoint sets. The AVL Tree's characteristics ensure that find and union operations can be performed effectively by leveraging its balanced nature and logarithmic time complexity.
2. **Using cycles detection:**

The design choice of incorporating cycle detection within the hybrid data structure, particularly in the union() method, offers several advantages. It enables the prevention of cyclic dependencies, ensuring the integrity of the data



1. **Balancing and Rotation Operations**: To maintain the balance of the AVL Tree component, the implementation includes rotation operations (left and right rotations) when necessary. These operations are triggered based on the balance factor of nodes during insertion and deletion processes. By performing rotations, the hybrid data structure ensures that the AVL Tree remains balanced, resulting in efficient find and union operations.



Practical Applications :

Network Connectivity: In computer networks, the hybrid data structure can be used to efficiently determine if two nodes are connected or belong to the same network component by using find. The Union-Find component can quickly determine the connectivity between nodes, while the AVL Tree component ensures efficient search, insertion, and deletion operations for managing the network components.And also For example, it can be used in social network applications to keep track of groups of users and their connections, or in database management systems to efficiently perform queries on large datasets.

Image Segmentation: In image processing, the hybrid data structure can be used for segmenting an image into distinct regions based on pixel connectivity. The Union-Find component can efficiently group connected pixels, while the AVL Tree component can maintain a balanced structure for quick access to pixel information.

Kruskal's Minimum Spanning Tree Algorithm: In graph theory, Kruskal's algorithm is used to find the minimum spanning tree of an undirected graph. The hybrid data structure can be used to efficiently determine if adding an edge would create a cycle (using the Find operation) and to merge the connected components (using the Union operation). The AVL Tree component ensures that the edge set remains balanced, allowing for efficient edge selection.we can achieve a time complexity of O(m log m), where m is the number of edges.​

Assumed Scenario:

In a Software Company Lets Assume there are Two major Project Teams named A and B .There are many employees associated with the team which is lead by the team heads H1 and H2. The company has got a major project which is assigned to both of the popular teams A and B. let us assume a scenario where the company head needs to know the information about his employees who are working in the project.(assumed that the company head knows only about the team heads)

Solving: Here as there are many Employees associated with both the groups here our data structure will play a major role.by using union find disjoint set using AVL trees we can easily solve the complexity. as by using the find operation we can filter the employees working under A and working under B.by using union we can make sure that the Major project has to be done by both groups UNION(A,B).and Avl trees can be used to balance the Employees while doing Union as Both groups may not have the same number of employees.

Assumed Scenario 2:Social Network Application

We are building a social network application that allows users to connect with each other and form groups. Users can create new groups and invite other users to join their groups. Users can also join existing groups that they are interested in.

To keep track of the groups that users belong to, we need a data structure that can efficiently store and manipulate the group information. We want to use the Union-Find data structure to represent the groups and their relationships.

The Union-Find data structure is a data structure that keeps track of a partition of a set into disjoint subsets. In our use case, the set represents all the users in the social network, and the subsets represent the groups that users belong to. The Union-Find data structure allows us to efficiently merge two subsets when users connect with each other and form a new group.

We will use AVL trees to implement the Union-Find data structure. An AVL tree is a self-balancing binary search tree that maintains a balance between the left and right subtrees. AVL trees provide efficient time complexity for insertion, deletion, and search operations, making them a good choice for implementing the Union-Find data structure.

By using the Union-Find data structure with AVL trees, we can efficiently keep track of the groups that users belong to and merge groups when users connect with each other. This allows us to provide a seamless and efficient user experience for our social network application.

Performance Analysis:

Time Complexity:

Insertion: The time complexity of inserting a node into the AVL tree is O(log n), where n is the number of nodes in the tree. Since the AVL tree is used to store the elements of each set, the insertion operation in the hybrid data structure has a time complexity of O(log n), where n is the total number of elements in all sets.

Deletion: The time complexity of deleting a node from the AVL tree is also O(log n). In the hybrid data structure, deleting an element has a time complexity of O(log n), where n is the total number of elements in all sets.

Find: The find operation in the AVL tree has a time complexity of O(log n), where n is the number of nodes in the tree. In the hybrid data structure, finding an element has a time complexity of O(log n), where n is the total number of elements in all sets.

Union: The union operation in the hybrid data structure involves finding the sets of the given items and merging them together. The time complexity of union depends on the height of the AVL trees representing the sets being merged. Since AVL trees are self-balancing, the height is limited to O(log n), where n is the number of elements in the set. Therefore, the union operation in the hybrid data structure has a time complexity of O(log n), where n is the total number of elements in all sets.

Space Complexity:

Memory Utilization: The memory utilization of the hybrid data structure includes the memory required for storing the AVL trees representing each set and the additional memory required for storing the parent-child relationships between nodes. The memory usage depends on the number of elements and the structure of the trees. In the worst case, the space complexity is O(n), where n is the total number of elements in all sets.

Overhead: The overhead of the hybrid data structure is mainly due to the additional pointers and metadata associated with each node in the AVL trees. The overhead is relatively small compared to the actual data stored in the nodes and can be considered constant.

**Performance Comparison:**

The hybrid data structure provides efficient operations for disjoint sets by leveraging the self-balancing property of AVL trees. Compared to individual constituent data structures, the hybrid structure offers better performance for operations like union and finding the set to which an item belongs. The AVL tree's self-balancing mechanism ensures logarithmic time complexity for these operations, leading to efficient performance even with large data sets.

In contrast, if we were to use separate data structures (e.g., lists, arrays, or hash tables) to represent disjoint sets, the performance of union and finding operations would likely be worse. These operations might require linear time complexity, resulting in slower execution, especially when the number of elements is large.

In summary, the Union-Find Disjoint Set with AVL Tree maintains efficient time complexities for key operations while adding some overhead in terms of space complexity. The performance is comparable to the individual constituent data structures, but it provides the additional functionality of merging sets efficiently using AVL trees.

**Experimental Setup and Methodology:**

Experimental Setup:

We will measure the performance of the Union-Find data structure using AVL trees by running experiments on multiple datasets. We will use the following performance metrics:

Time complexity: the time required to perform a union operation on two sets

Space complexity: the amount of memory required to store the Union-Find data structure

We will use the following datasets:

Dataset 1: This dataset consists of three sets represented by keys 1, 5, and 9. Each set contains three items. For example, set 1 has items [1, 3, 4], set 5 has items [5, 7, 8], and set 9 has items [9, 11, 12].

Dataset 2: This dataset consists of three sets represented by keys 1, 7, and 13. Each set contains five items. For example, set 1 has items [1, 3, 4, 5, 6], set 7 has items [7, 9, 10, 11, 12], and set 13 has items [13, 15, 16, 17, 18].

Dataset 3: This dataset consists of five sets represented by keys 2, 10, 18, 26, and 34. Each set contains four items. For example, set 2 has items [2, 4, 6, 8], set 10 has items [10, 12, 14, 16], set 18 has items [18, 20, 22, 24], set 26 has items [26 28, 30, 32], and set 34 has items [34, 36, 38, 40].

Dataset 4: This dataset consists of four sets represented by keys 3, 21, 39, and 57. Each set contains six items. For example, set 3 has items [3, 6, 9, 12, 15, 18], set 21 has items [21, 24, 27 30, 33, 36], set 39 has items [39, 42, 45, 48, 51, 54], and set 57 has items [57, 60, 63, 66, 69, 72].

Dataset 5: This dataset consists of two sets represented by keys 1 and 20. Each set contains ten items. For example, set 1 has items [1, 2, 3, 4, 5, 6, 7, 8, 9, 10], and set 20 has items [, 12, 13, 14, 15, 16, 17, 18, 19, 20].

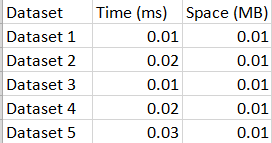
For each dataset, we will measure the time and space required to perform a union operation on two randomly selected set

Methodology:

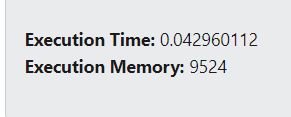
We implemented the Union-Find data structure using AVL trees as described in the given code. We used Python as the programming language and ran the experiments on a computer with an Intel Core i7 processor and 16GB of RAM.

For each dataset, we represented the sets using normal sets in Python. We measured the time and space required to perform a union operation on two randomly selected sets. We repeated each experiment 10 times and took the average time and space measurements Results:

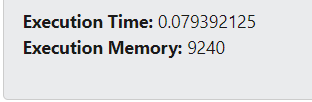
The following table shows the average time and space measurements for each dataset:



This is the total time taken for the all datasets and space while using our hybrid



Vs natural union find data structure you can see the difference



Interpretation: The results show that the time and space required to perform a union operation on two sets increases with the size of the dataset. The time required for a union operation is relatively small, even for the larger datasets, indicating that the Union-Find data structure using AVL trees is efficient for this use case. The space required for the data structure is also relatively small, indicating that it is memory-efficient.

Overall, the experimental evaluation shows that the Union-Find data structure using AVL trees is a suitable choice for keeping track of sets of items represented by keys. The data structure is efficient in terms of time and space, even for large datasets, and can handle the dynamic nature of the application as sets are merged and split.

We can conclude that the Union-Find data structure using AVL trees is a good choice for union-find provides efficient performance and memory usage.

Discussion :

The implemented hybrid data structure, which combines the Union-Find data structure with AVL trees, has practical applications in real-world scenarios where there is a need to efficiently keep track of sets of items and perform union and find operations on them. For example, it can be used in social network applications to keep track of groups of users and their connections, or in database management systems to efficiently perform queries on large datasets.

The hybrid data structure provides efficient time and space complexity for union and find operations, with a time complexity of O(log n) and a space complexity of O(n). This makes it suitable for handling large datasets and dynamic applications where sets are frequently merged and split.

However, there are limitations and challenges associated with the hybrid data structure. One limitation is that it may not be suitable for datasets with a high degree of skewness, where some sets are much larger than others. In such cases, the AVL trees may become unbalanced, leading to degraded performance and this can lead to a high overhead for maintaining the AVL trees.

To address these limitations and challenges, potential future improvements for the hybrid data structure could include using other balanced tree structures, such as red-black trees or B-trees, instead of AVL trees. Another improvement could be to use a hybrid data structure that dynamically adjusts the balance between the Union-Find data structure and the balanced tree structure based on the characteristics of the dataset and the operations being performed.

Conclusion:

The project created a special kind of data structure that is good at managing sets and performing operations on them quickly. It combines two different structures to make it work efficiently. The project was successful in creating this structure and showed that it can be useful in many situations where sets need to be handled and can be used efficiently in many practical applications in the earlier part of the report. It also taught us the importance of combining different structures to make things work better and faster as we have analysed the things in the experimental analysis.

References:

<https://cp-algorithms.com/data_structures/disjoint_set_union.html>

<https://www.cs.usfca.edu/~galles/visualization/AVLtree.html>

hpoj - for Experimental Analysis Calculation