# GRAPH WITH TRIE BASED-INDEXING

# Implemented using combining GRAPH AND TRIE

GROUP NAME : **HEAPSTERS**

GROUP MEMBERS :

|  |  |
| --- | --- |
| NAME | ROLL NUMBER |
| Aswin M | CB.EN.U4CSE21408 |
| Neeli Uday Kumar | CB.EN.U4CSE21441 |
| Nila Bhaskar | CB.EN.U4CSE21442 |
| Rithish Abinav N | CB.EN.U4CSE21448 |
| Abhignya | CB.EN.U4CSE21463 |

INTRODUCTION:

Graph :

A Graph is a non-linear data structure consisting of vertices and edges. The vertices are sometimes also referred to as nodes and the edges are lines or arcs that connect any two nodes in the graph. More formally a Graph is composed of a set of vertices( V ) and a set of edges( E ). The graph is denoted by G(E, V).  
For implementing this hybrid data structure we are going to use directed graphs.

Trie-based Indexing Data Structure :

This project focuses on designing and implementing a hybrid data structure called "Graph with Trie-Based Indexing."

The main advantages of trie-based indexing include:

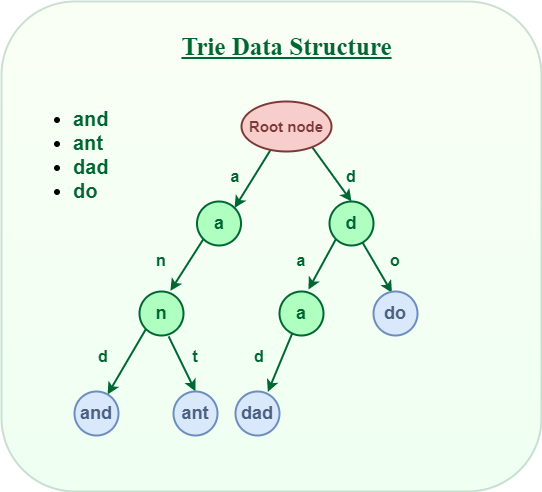
->Fast Prefix Matching: Efficient prefix retrieval

->Space Efficiency: Compact representation

->Quick Search and Retrieval: Fast data access

->Dynamic Insertion and Deletion: Dynamic operations support

->Alphabet-Based Indexing: Customizable indexing



Graph with Trie-based Indexing:

The objective of our project is twofold: first, to design and implement the Graph with Trie-Based Indexing data structure, and second, to explore its practical applications and analyze its time and space complexity.

We will develop the Graph with Trie-Based Indexing data structure, implement essential algorithms, evaluate performance, and demonstrate its applicability in domains like natural language processing, network analysis, and recommendation systems.

OVERVIEW OF THE HYBRID DATA STRUCTURE:

By combining the graph and trie concepts, a hybrid data structure can capture both the hierarchical structure of a trie and the relationship network of a graph. The main idea is to represent the hierarchical structure using trie nodes and use graph edges to represent relationships between different trie nodes.​

Here's a high-level overview of how a graph-trie hybrid works:​

1.Each node in the trie represents a common prefix or a full word, depending on the use case. The root node represents an empty prefix or the starting point.​

​2.The children of each trie node correspond to the possible extensions of the prefix represented by the parent node. These children can be organized in a variety of ways, such as a list, an array, or a map.

3. In addition to the trie structure, the hybrid data structure maintains graph edges between trie nodes to represent relationships. These edges can be directed or undirected, depending on the nature of the relationships.​

4. The graph edges connect trie nodes based on specific criteria or rules. For example, in a social network context, the edges might represent friendships between users, where each trie node corresponds to a user and the edges connect friends.​

5. With the hybrid structure in place, you can perform various operations efficiently. For instance, you can navigate the trie structure to search for specific prefixes or words, while also leveraging the graph edges to traverse related trie nodes.

here are certain advantages of using this hybrid data structure for solving specific problems efficiently. ​

1. Hierarchical Organization: Tries inherently provide a hierarchical organization of data based on shared prefixes. This is particularly useful when dealing with structured data that exhibits a hierarchical pattern. The hybrid data structure leverages this strength, allowing for efficient storage and retrieval of hierarchical information. This can be beneficial in applications like file systems, dictionary lookups, or organizing data with nested categories.​

2. Efficient Searching: Tries are known for their efficient prefix-based searching. By incorporating trie-like structures into the hybrid data structure, it becomes well-suited for problems that involve searching for words or patterns efficiently. The hierarchical nature of the trie allows for fast retrieval of words with shared prefixes, while the graph connections enable traversal across related trie nodes.​

3. Relationship Representation: Graphs excel at representing relationships between entities. By incorporating graph edges into the hybrid data structure, it becomes possible to capture and utilize relationships between trie nodes. This is valuable in scenarios where relationships play a significant role in the problem domain. For instance, in social networks, the hybrid structure can efficiently represent friendships, connections, or interactions between users.

4. Flexibility and Adaptability: The hybrid data structure offers flexibility and adaptability in representing and solving various problems. It can handle a wide range of data types, including strings, numbers, or custom objects, making it versatile for different applications. The structure can be tailored to specific problem requirements, allowing for customizations such as directed or undirected edges, weighted edges, or additional properties associated with trie nodes or edges.​

5. Improved Performance: By combining the strengths of tries and graphs, the hybrid data structure can provide improved performance for specific problem domains. It can significantly reduce the search space and traversal complexity compared to traditional graph-based approaches. This can result in faster operations, reduced memory usage, and improved overall efficiency for tasks like autocomplete, recommendation systems, network analysis, and more.​

6. Domain-specific optimizations: The hybrid data structure can be optimized further based on specific problem domains. For example, various techniques like caching, pruning, indexing, or compression can be applied to enhance the performance and reduce memory requirements. These optimizations can be tailored to the unique characteristics and requirements of the problem at hand.​​

​​IMPLEMENTATION DETAILS:

The hybrid data structure Graph with Trie based indexing is implemented using the adjasent list, where the main list can be maximum of size 27and all the child lists can be maximum of size 26.​

For every node in the main list has a variable is\_end which is a booleanf data type and stores True I the character/letter stored in that node is end of any string/word and stores False if it is not. ​

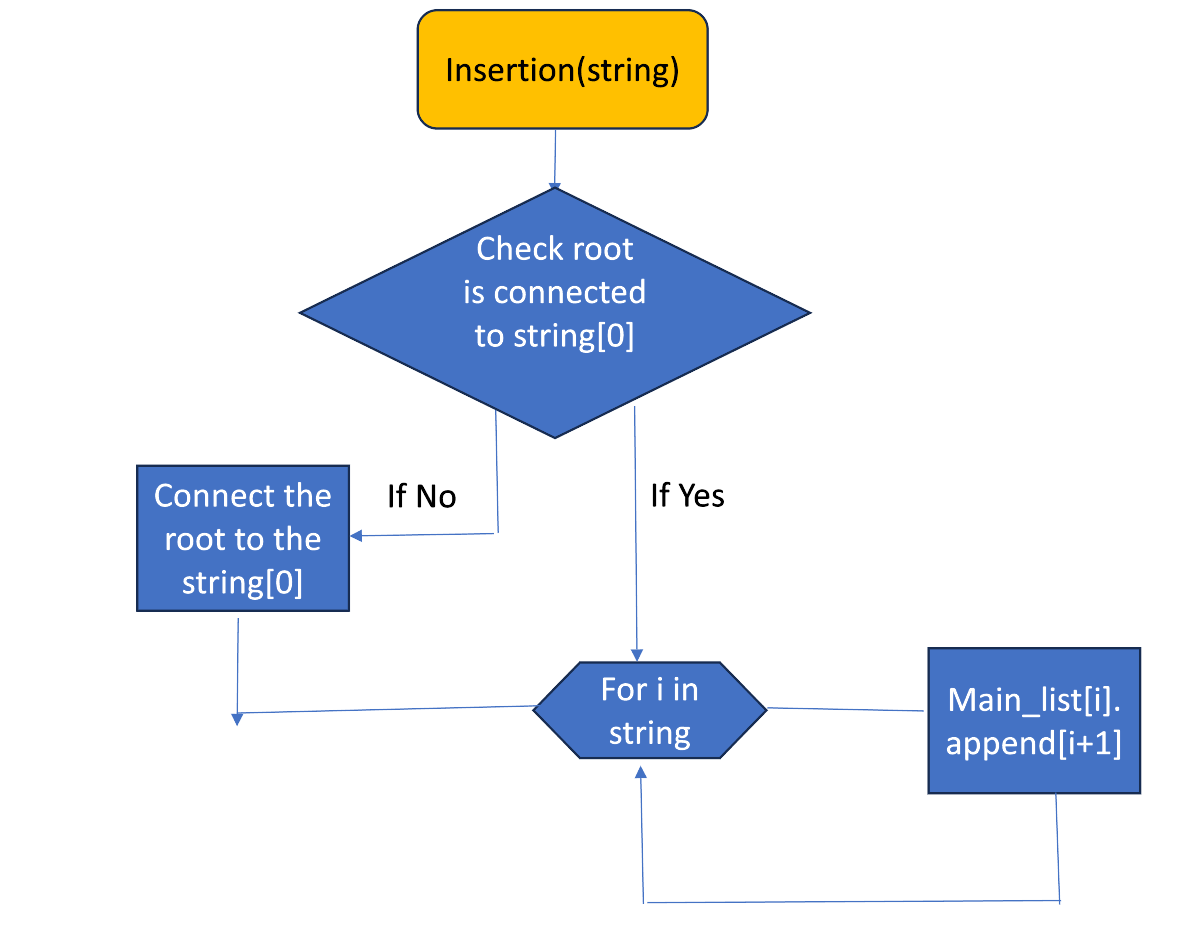
We are going to use the directed graph for implementation.​

This hybrid data structure is mainly used for storing and retriving the strings ,so the operations we are going to perform in this data structure are inserting a string and dearching for the string.​

We can even perform the deletion but it is not efficient in some cases.​

Insertion:

​Flowchart:



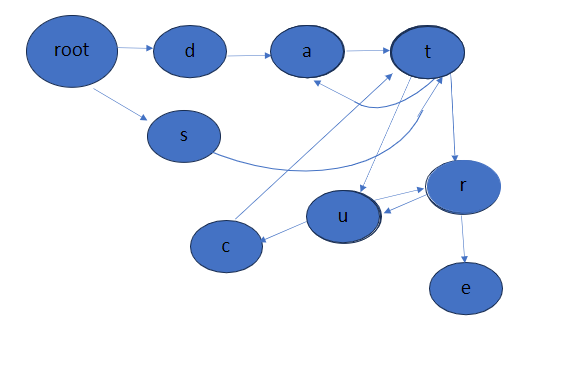
For inserting a string we are going to follow a process where we first will check is the first letter of the string is connected to the root node or not and if not we will connect the root to the first letter.​

For programming the above case we are going to check is the first letter is present in the first node of the main list which is allocated to store the list of nodes that are connected to the root, if not found then we are going to add that letter in the child list of the root node.​

Now we have to connect each adjacent letter in the string one by one that is for the letter (n)th from the initial letter to the final letter in the string is to be connected to the (n+1)th letter. For programming this we are going to insert the (n+1)th letter into the child list of the Nth node in the main list.​

​Ex:

Let’s insert strings data and structure.The graph is going to look like this

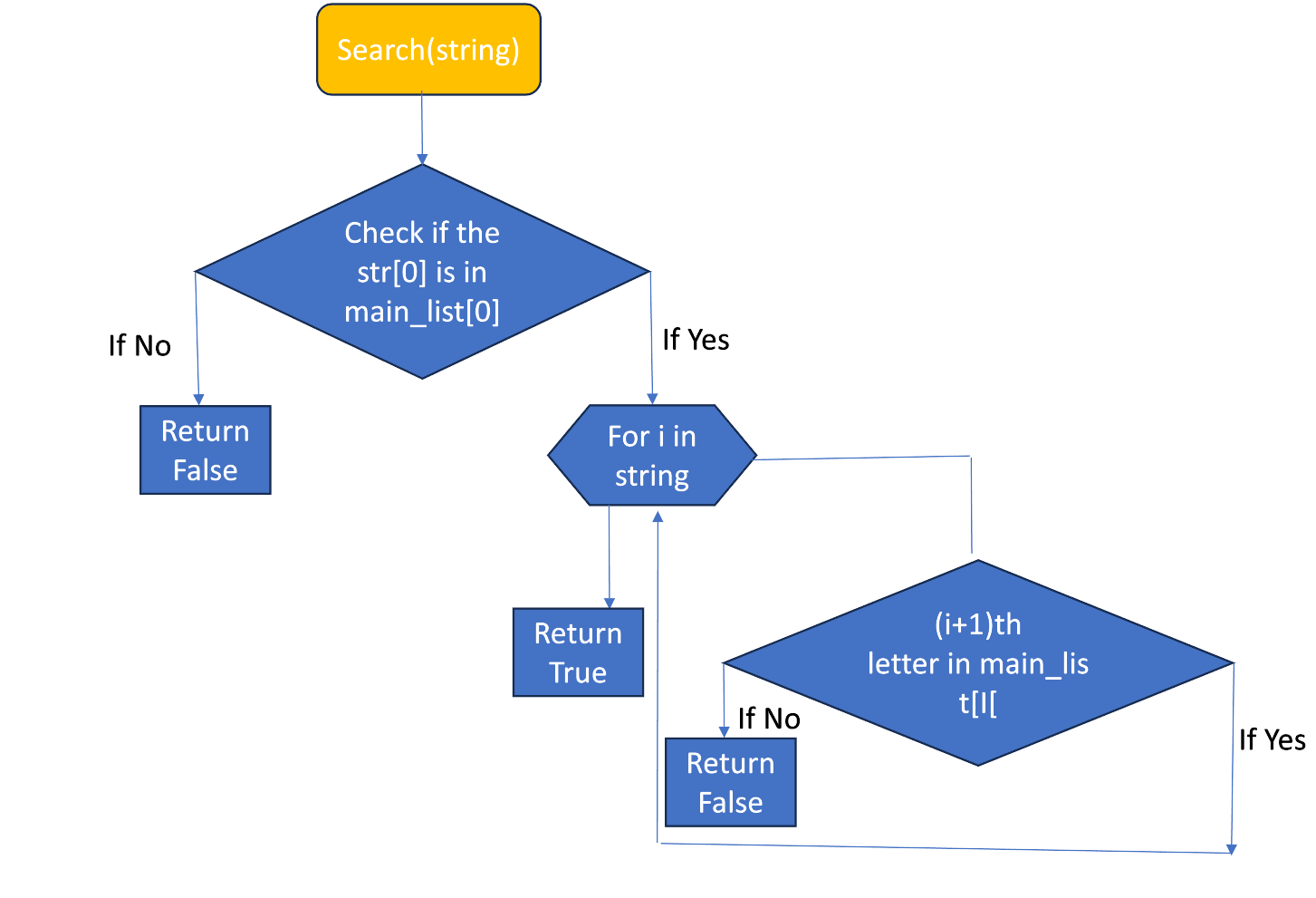


Searching:

To perform the search we follow a process where we check whether the first letter of the string is connected to root node or not if not connected this function is directly going to return false .If the node is connected to the root then for each nth letter from the initial letter of the string to ending letter of the string we are going to check is the (n+1)th letter is connected or not if not connected then this function is going to return false.​

For programming the above case first we are going to check is the first letter of the string is stored in the child list of the root node of the main list, if no we are directly going to return False. If the first letter Is stored then for each letter n in the string we are going to check if the (n+1)th letter is stored in the child list of nth node of the main list, if yes for every letter then we are going to return True, if for any letter the condition fails then we will return False.

The flow chart of search operation is:



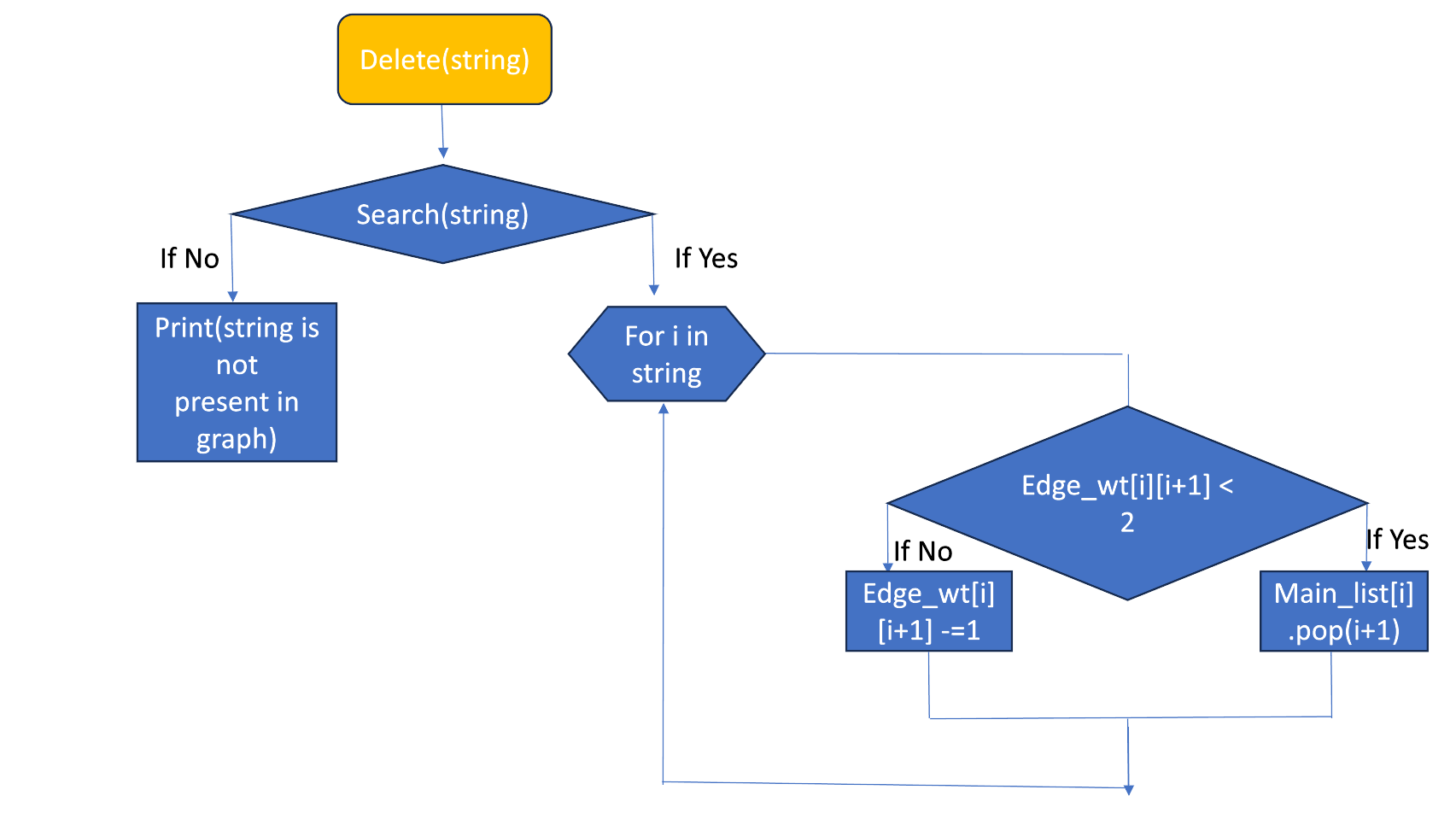
Deletion:

For performing deletion without losing or affecting the other strings/words we are going to increase the edge weight for every connection while inserting the string ​while a new edge is added. ​

Now to delete a string from this hybrid data structure we are going to first dearch if the string is present in the graph or not if not return the string is not present in the graph ,if Yes then for every connection of the letters in the string if the edge weight is 1 then we are going to remove that edge/connection and if edge weight is grater than 1 then we are going to decrease the edge weight of that edge by 1.​

For programming the above case with the help of the search function first, we are going ind if the string is present in the graph or not, then using the list which stores the ededge weight of every edge, with this list for every edge of the string to be deleted if he edge value is greater than 1 then we are going to decrease it by 1 if the edge value is 1 then we are going to removr that letter from that adjacent child list.

Flowchart of this operation is:



Link: https://github.com/uday1003/Team\_HEAPSTERS

PRACTICAL APPLICATIONS:

The Graph with Trie-Based Indexing hybrid data structure offers a versatile solution that can be effectively applied in various practical scenarios. By combining the strengths of graphs and trie data structures, it enables efficient operations and provides valuable insights in the following applications:

Natural Language Processing (NLP):

The hybrid structure finds significant application in NLP tasks such as text analysis, information retrieval, and language modeling. By representing textual data as a graph, with nodes representing words and edges indicating relationships (e.g., co-occurrence, semantic similarity), the structure facilitates efficient search, clustering, and recommendation algorithms. The trie-based indexing enables fast prefix searches and efficient autocomplete functionalities, aiding tasks like keyword matching, spell checking, and text prediction.

Social Network Analysis:

Social networks can be modeled as graphs, where nodes represent individuals, and edges denote connections or relationships between them. The hybrid data structure facilitates the analysis of large-scale social networks by efficiently storing and querying the graph structure. The trie-based indexing allows for quick access to user profiles, efficient searching for individuals based on their attributes (e.g., location, interests), and identification of relevant communities or influential individuals.

Recommendation Systems:

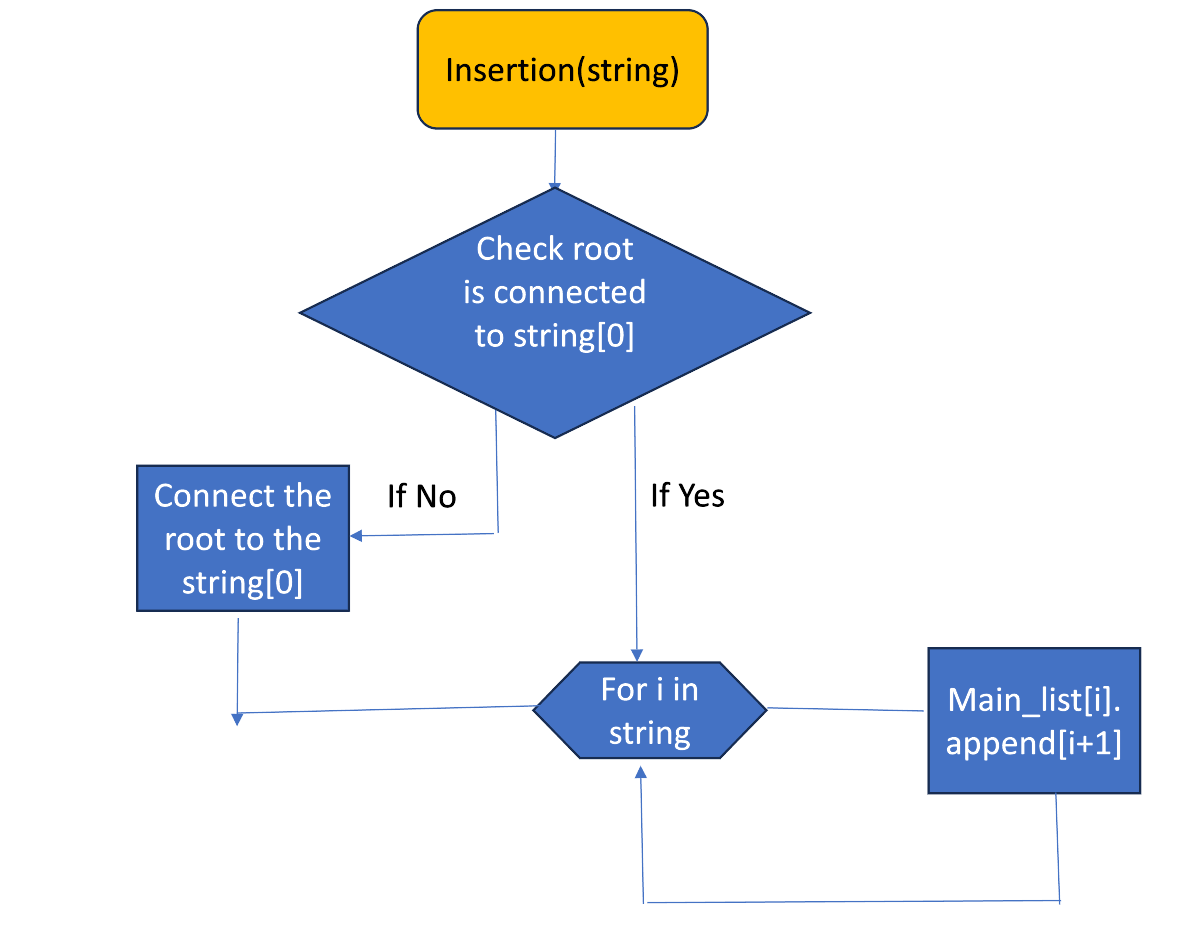
Graphs with trie-based indexing are instrumental in building recommendation systems that suggest items based on user preferences and behaviors. The hybrid structure enables the representation of user-item interactions as a graph, where nodes represent users and items, and edges indicate interactions or ratings. By leveraging the trie-based indexing, the system can efficiently retrieve similar users or items, perform personalized recommendations, and support real-time updates and recommendations.

TIME COMPLEXITY:

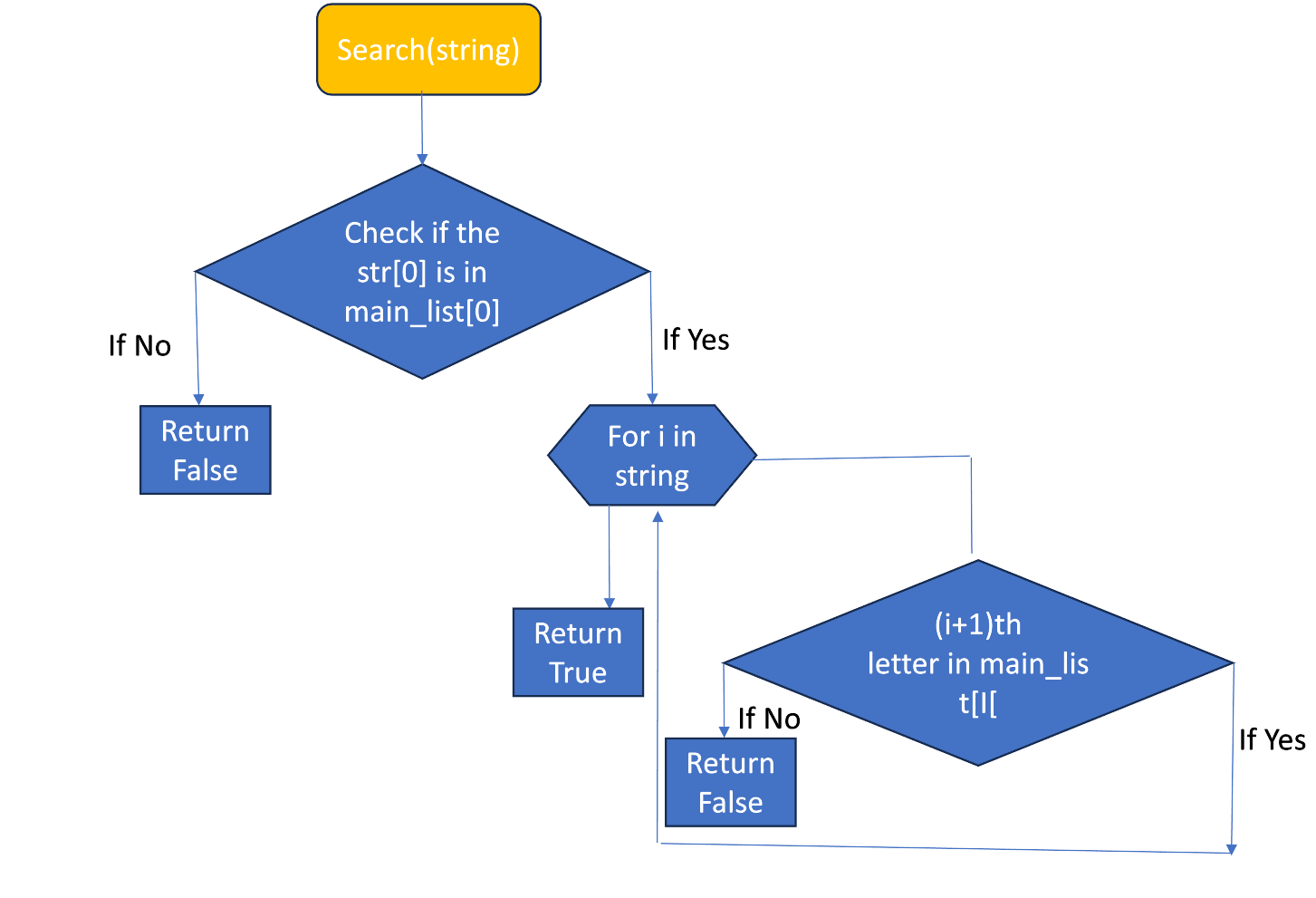
For insertion:

If we observe the flowchart of the insertion, for inserting the string we are connecting each letter in the string through for loop. so the time complexity is going to depend on the length of the string, so if the length of the string we are inserting is N then the time complexity of inserting string is O(N).​

If we compare time complexity with trie which is O(N) where n is length of the string and with graph which is O(N) where N is length of the string to insert there is no difference every thing does the insertion in same time.



For searching:

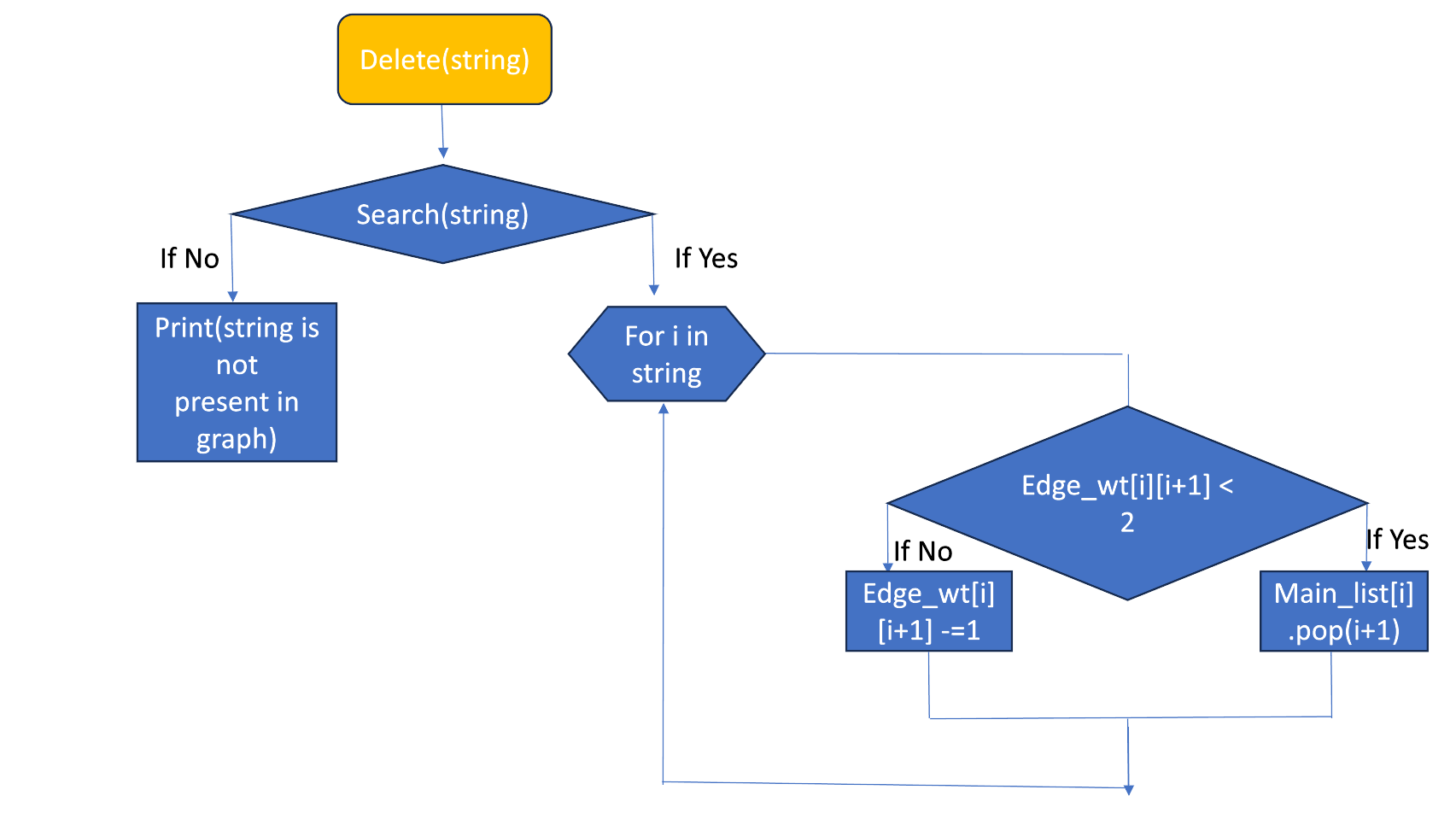


For searching the string it is going to depend on length of the string so if the length of the string is N then the time complexity of searching the string is O(N).​

If we compare the time complexity with the trie which is also O(N) ,so for searching the string our hybrid data structure and in trie takes the same time.

For deletion:

For searching the string it is going to depend on length of the string and the total number of vertices int the graph because in the worst case we have to search every vertex int the graph so if the length of the string is N and total number of vertices are V then the time complexity of searching the string is O(N\*V).​



If we compare the time complexity of deleting a string in the trie which is O(N) with our hybrid data structure it is smaller because in trie data structure the way we storing the string is different as compared to storing in our hybrid data structure , where the can be multiple repeated nodes in the trie but not in our hybrid data structure .

To delete the string in the hybrid data structure we are going to first check the each weight of the every edge in the string and then if the edge weight is 1 only then we are going to delete that edge so because of this process the time is going to be more.

SPACE COMPLEXITY:

The space complexity of the hybrid data structure graph with trie-based indexing is O(N\*N) where the N is the number of vertices in the graph .As compared to ths space complexity of the trie which is O(N\*K) where N is the nodes in the trie and K is the total number of unique characters in the alphabet so if we consider the worst case in both the scenario we are going to have a lesser space complexity for the hybrid data structure only because in the trie there are going be some nodes repeated to store the string where in the hybrid data structure it is not the case.

EXPERIMENTAL EVALUATION:

While considering the whole code the experiment to insert ,search and delete a string takes an

**Execution Time:**  0.05239144   
**Execution Memory:** 9542where we can analyze space and time taken to run our total code.

CONSIDER ONLY INSERTION

**Execution Time:** 0.04207999  
**Execution Memory:** 9436

The above executions are only for inserting a string.

CONSIDER ONLY SEARCHING

**Execution Time:** 0.040910704  
**Execution Memory:** 9448

The above executions are for insertion and searching because before searching we have to insert any string.

CONSIDER ONLY DELETION

**Execution Time:** 0.040865872  
**Execution Memory:** 9448

DISCUSSION:

The implementation of a hybrid data structure combining a graph and trie-based indexing can be practical and effective in various real-world scenarios, particularly when dealing with large-scale datasets and complex relationships between entities. Let's discuss its practicality, effectiveness, limitations, challenges, and potential future improvements.  
  
Practicality:  
1. Efficient Storage: The hybrid structure optimizes storage by leveraging the strengths of both graphs and tries. Graphs efficiently represent complex relationships between entities, while tries offer fast prefix-based searching and indexing capabilities.

2. Scalability: This hybrid approach allows for scaling to large datasets with minimal impact on performance. The graph component handles relationships between entities, while the trie-based indexing optimizes search operations, making it practical for scenarios with a high volume of data.

3. Flexibility: The hybrid structure can handle a wide range of scenarios, such as social networks, recommendation systems, knowledge graphs, and data exploration, where both relationships and efficient indexing are crucial.

Effectiveness:

1. Relationship Representation: The graph component of the hybrid structure excels at representing complex relationships between entities. It allows for efficient traversal and exploration of interconnected data, making it effective for analyzing patterns, making recommendations, or performing graph-based algorithms.

2. Fast Searching: The trie-based indexing provides efficient prefix-based searching, enabling quick retrieval of data based on partial matches. This makes it effective for autocomplete, spell checking, and search functionalities in various applications.

3. Combined Strengths: The hybrid approach combines the strengths of graphs and tries, providing a comprehensive data structure that efficiently handles both relationship management and indexing/search operations.

Limitations:

1. Update Complexity: Maintaining consistency between the graph and trie-based indexing can be challenging. Updates, such as adding or removing nodes/edges, require careful synchronization to ensure the graph and indexing remain coherent. This can impact the efficiency of updates, especially in highly dynamic scenarios.

2. Memory Overhead: The hybrid structure may require additional memory to store both graph and trie-based indexes. This overhead can be a limiting factor when dealing with extremely large datasets.

3. Trade-off in Performance: While the hybrid structure offers benefits in certain use cases, it may not outperform specialized data structures designed solely for relationship management or indexing. It represents a compromise between the two, which might result in reduced performance compared to dedicated data structures in specific scenarios.

Challenges:

1. Synchronization Complexity: Ensuring consistency between the graph and trie-based index during updates can be a non-trivial task. Developing efficient synchronization mechanisms is crucial to maintain data integrity and minimize performance impact.

2. Memory Management: Balancing memory usage between the graph and trie-based index requires careful consideration. Efficient memory allocation and usage strategies must be implemented to optimize the hybrid structure's overall performance.

3. Scalability and Performance Optimization: As datasets grow larger and more complex, optimizing the hybrid structure's performance becomes a significant challenge. Techniques such as distributed computing, parallel processing, and indexing strategies must be explored to handle scalability challenges effectively.

Potential Future Improvements:

1. Incremental Updates: Developing efficient algorithms to handle incremental updates can significantly improve the hybrid structure's performance. By minimizing the impact of updates on both the graph and indexing components, real-time data management becomes more feasible.

2. Adaptive Memory Management: Dynamic memory management strategies that intelligently allocate and deallocate resources based on usage patterns can help optimize memory utilization in the hybrid structure.

In summary, the hybrid data structure combining a graph with trie-based indexing demonstrates practicality and effectiveness in real-world scenarios. While it offers benefits in handling complex relationships and efficient searching, challenges regarding synchronization.

CONCLUSION:

This project focus on implementation of the hybrid data structure graph with trie-based indexing and we implemented using adjacent list.the outcomes and finding of the project are as follows:

practical applications: the structure can be used in scenarios such as social networks, recommendation systems, knowledge graphs, and data exploration. It allows for efficient representation and traversal of interconnected data, as well as fast retrieval of data based on partial matches.

The performance analysis: the implemented data structure demonstrates its effectiveness. The insertion of strings is supported, with each string being added as vertices and edges in the graph component. The trie-based indexing optimizes the search operations, facilitating quick retrieval of data. The implementation includes functionalities to find words, delete strings, and print the structure.

Efficiency is achieved through the combination of graph and trie-based indexing. The graph handles relationship representation, while the trie-based indexing offers fast prefix-based searching. The hybrid structure capitalizes on the strengths of both data structures.

Insights gained from the implementation of the project include the importance of synchronization between the graph and trie-based indexing components, the challenges of memory management in large datasets, and the trade-off between performance optimization and the combined functionality of the hybrid structure.

Overall, the project demonstrates the practicality, effectiveness, and potential of the hybrid data structure, highlighting its usefulness in real-world scenarios and providing valuable insights for future enhancements.

REFERENCES:

<https://www.geeksforgeeks.org/graph-data-structure-and-algorithms/>

<https://www.geeksforgeeks.org/trie-insert-and-search/>

<https://iq.opengenus.org/time-complexity-of-trie/>