TASK #1: Data structure problem solving
Please note: This task does not require code to be written.
Assuming we have a large set of patients (500,000+) in a relational database, we want the user to type part of a patient name and the system returns a list of matched patients.
o What data structure and search/matching algorithm to use and why?
o How fast the search is using this algorithm and data structure?
o Additional approaches from any point of view are acceptable to achieve best performance resul

Answer :
To efficiently search for patients by name in a large relational database, I recommend using a Trie (prefix tree) data structure
and a prefix matching algorithm.

Why Trie? A Trie is a suitable data structure for this problem because it allows for fast prefix matching, which is essential for searching patients by name.
A Trie is a tree-like data structure where each node represents a string prefix. This data structure is particularly useful when dealing with large datasets
and frequent prefix searches.

Why prefix matching algorithm? The prefix matching algorithm is a natural fit for this problem because it allows us to search for patients
by name using a partial name input. This algorithm works by traversing the Trie from the root node, following the edges corresponding to the input prefix. When a match is found, the algorithm returns a list of patients with matching names.
Time complexity The time complexity of the prefix matching algorithm using a Trie is O(m), where m is the length of the input prefix.
This is because we only need to traverse the Trie up to the length of the input prefix to find matching patients.

Additional approaches To further optimize the search performance, consider the following additional approaches:

1.Indexing: Create an index on the patient name column in the relational database to speed up the search.
2.Caching: Implement a caching mechanism to store frequently searched patient names and their corresponding IDs. This can reduce the number of database queries and improve search performance.
3.Data partitioning: Partition the patient data into smaller chunks based on the first letter or prefix of the patient name. This can reduce the search space and improve performance.

IMPLEMENTATION:

```
import java.util.*;

class TrieNode {
   private Map<Character, TrieNode> children;
   private List<Integer> patients;

   public TrieNode() {
      children = new HashMap<>();
      patients = new ArrayList<>();
   }

   public void addChild(char c, TrieNode node) {
      children.put(c, node);
   }

   public TrieNode getChild(char c) {
```

```
        return children.get(c);
    }

    public void addPatient(int patientId) {
        patients.add(patientId);
    }

    public List<Integer> getPatients() {
        return patients;
    }
}

public class Trie {
    private TrieNode root;

    public Trie() {
        root = new TrieNode();
    }

    public void insert(String patientName, int patientId) {
        TrieNode node = root;
        for (char c : patientName.toCharArray()) {
            TrieNode child = node.getChild(c);
            if (child == null) {
                child = new TrieNode();
                node.addChild(c, child);
            }
            node = child;
        }
        node.addPatient(patientId);
    }

    public List<Integer> search(String prefix) {
        TrieNode node = root;
        for (char c : prefix.toCharArray()) {
            node = node.getChild(c);
            if (node == null) {
                return new ArrayList<>();
            }
        }
        return node.getPatients();
    }

    public static void main(String[] args) {
        Trie trie = new Trie();
        // Insert patient data into the Trie
        List<Patient> patients = /* retrieve patients from database */;
        for (Patient patient : patients) {
            trie.insert(patient.getName(), patient.getId());
        }

        // Search for patients by name
        String prefix = "John";
        List<Integer> matchedPatients = trie.search(prefix);
        System.out.println(matchedPatients); // [patient_id1, patient_id2,...]
    }
}
```

```
class Patient {
   private int id;
   private String name;

   public Patient(int id, String name) {
      this.id = id;
      this.name = name;
   }

   public int getId() {
      return id;
   }

   public String getName() {
      return name;
   }
}
```

explanation:
In Java, we use a Map to store the children of each node, whereas in Python, we used a dictionary. We use a List to store the patient IDs associated with each node, whereas in Python, we used a list comprehension.
We use a TrieNode class to represent each node in the Trie, whereas in Python, we used a simple dictionary to store the node's children and patients.