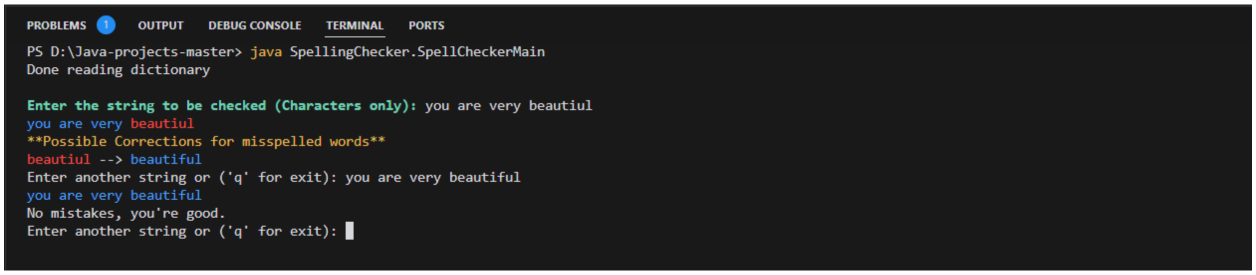
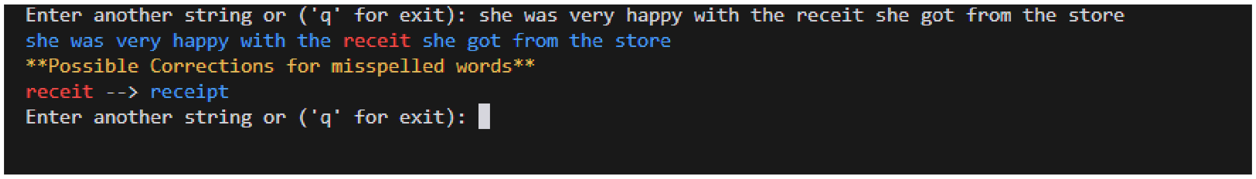
CUI Based Output

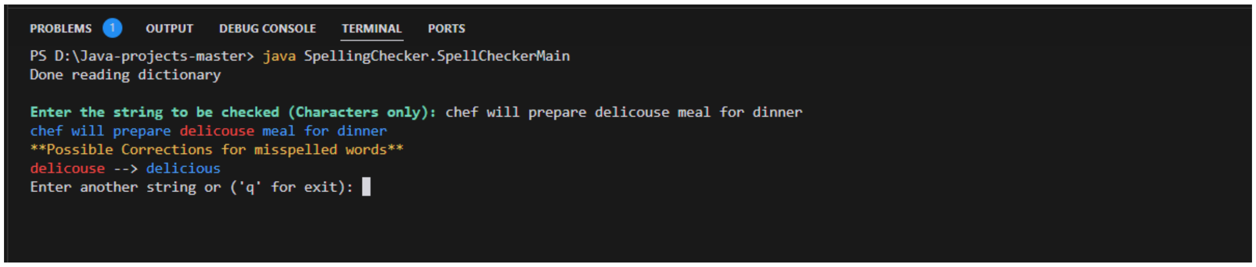


```
PS D:\Java-projects-master> java SpellingChecker.SpellCheckerMain
Done reading dictionary

Enter the string to be checked (Characters only): you are very beautiul
you are very beautiul
**Possible Corrections for misspelled words**
beautiul --> beautiful
Enter another string or ('q' for exit): you are very beautiful
you are very beautiful
No mistakes, you're good.
Enter another string or ('q' for exit):
```

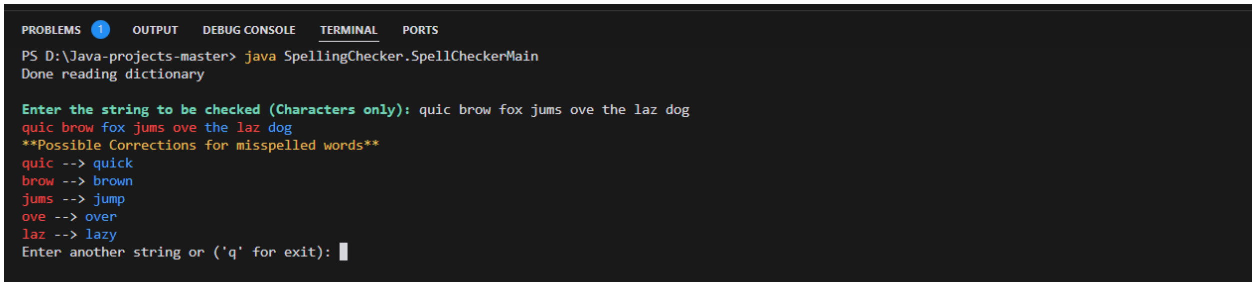
In above example we first add data which is not proper then we write data with given modification so there is no wrong spell in sentence there for it gives <u>No Mistakes,you're good.</u>



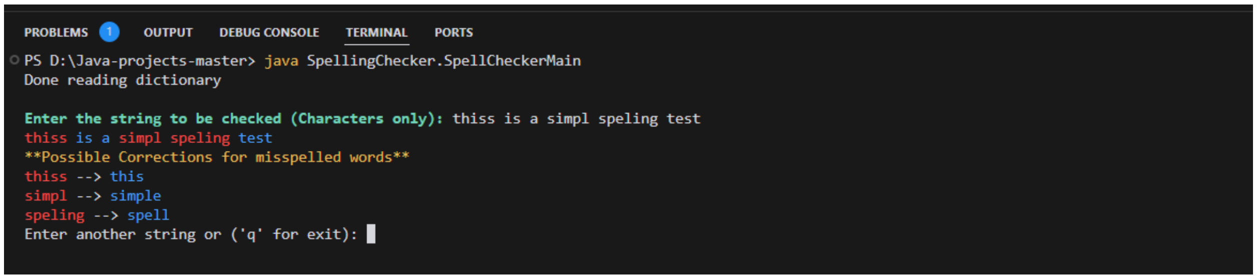
```
Enter another string or ('q' for exit): she was very happy with the receit she got from the store
she was very happy with the receit she got from the store
**Possible Corrections for misspelled words**
receit --> receipt
Enter another string or ('q' for exit):
```



```
PS D:\Java-projects-master> java SpellingChecker.SpellCheckerMain
Done reading dictionary

Enter the string to be checked (Characters only): chef will prepare delicouse meal for dinner
chef will prepare delicouse meal for dinner
**Possible Corrections for misspelled words**
delicouse --> delicious
Enter another string or ('q' for exit):
```

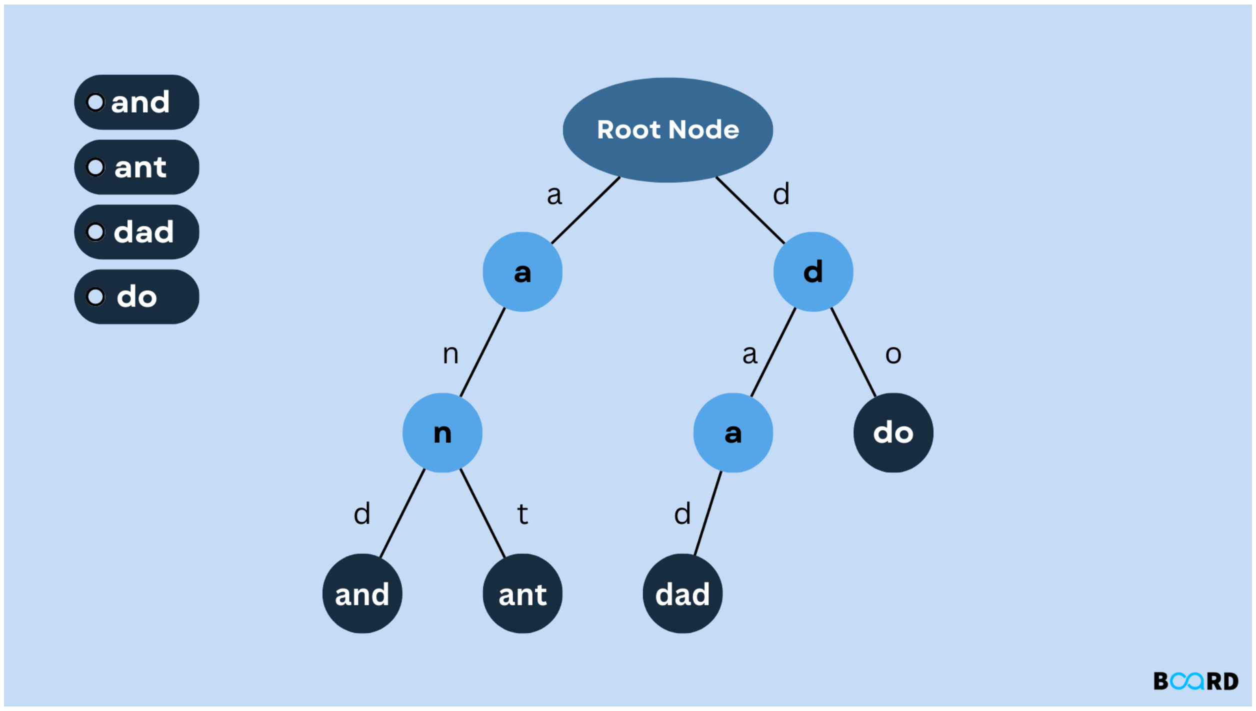


```
PS D:\Java-projects-master> java SpellingChecker.SpellCheckerMain
Done reading dictionary

Enter the string to be checked (Characters only): quic brow fox jums ove the laz dog
quic brow fox jums ove the laz dog
**Possible Corrections for misspelled words**
quic --> quick
brow --> brown
jums --> jump
ove --> over
laz --> lazy
Enter another string or ('q' for exit):
```



```
PS D:\Java-projects-master> java SpellingChecker.SpellCheckerMain
Done reading dictionary

Enter the string to be checked (Characters only): thiss is a simpl speling test
thiss is a simpl speling test
**Possible Corrections for misspelled words**
thiss --> this
simpl --> simple
speling --> spell
Enter another string or ('q' for exit):
```

Below Example of how Trie Data Structure works.



In this example we can see how it is adding data from our input here input is just like fetched from dictionary.txt .As every time common part of spelling

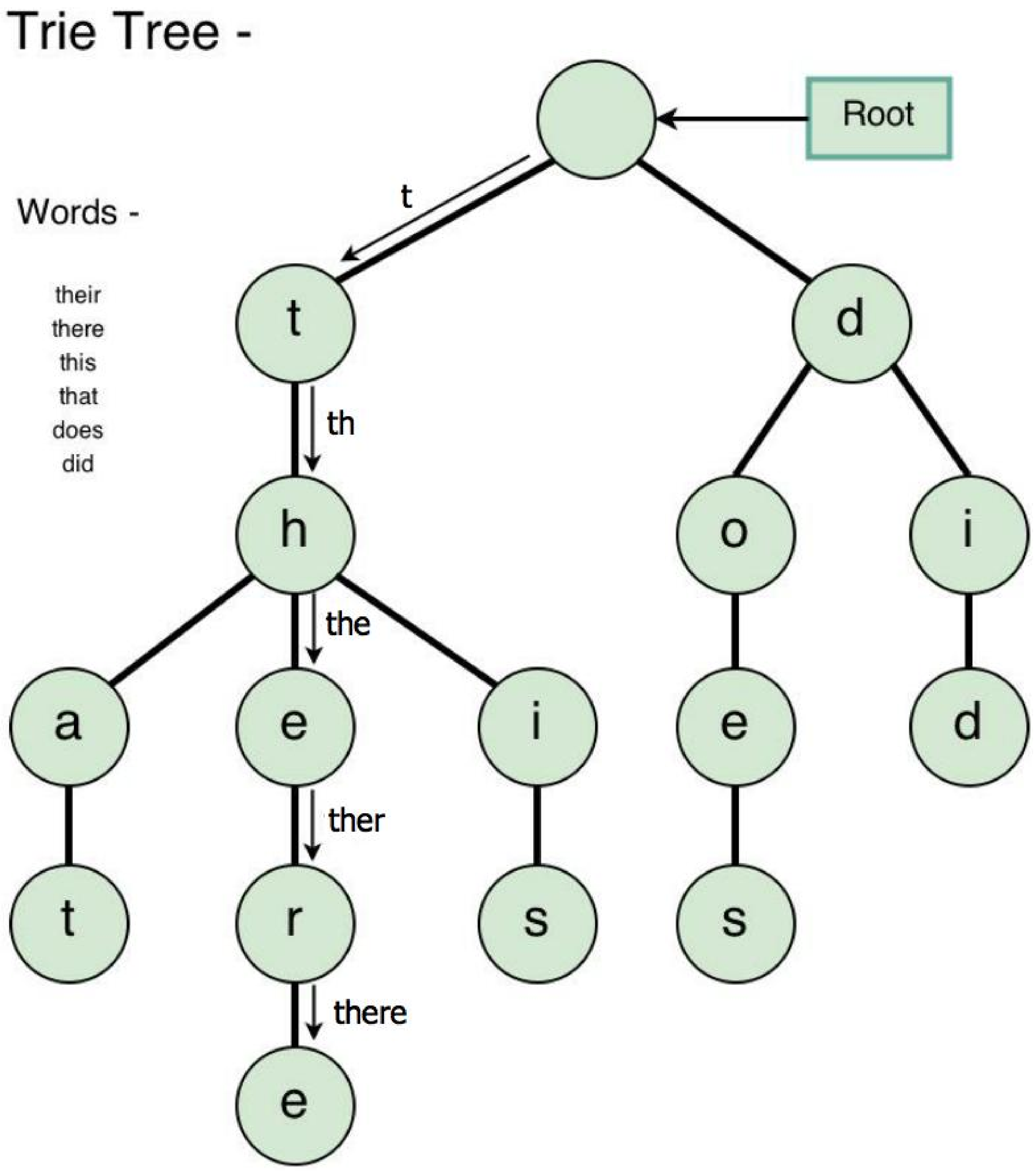
Explanation

In the context of a spelling checker project, particularly one that uses a Trie-based dictionary, the following process might be occurring:

1. **Input Data**: The input data for the spelling checker is typically fetched from a dictionary file, such as `dictionary.txt`. This file contains a list of correctly spelled words that will be used to verify and correct the spelling of input strings.

2. **Common Part of Spelling**: The term "common part of spelling" likely refers to the process of identifying and processing common prefixes, suffixes, or substrings in words. This step is crucial for efficiently checking the spelling and suggesting corrections. The Trie data structure excels in this aspect because it organizes words in a way that allows for quick prefix and substring lookups.
3. **Adding Data**: As the spelling checker processes each word from `dictionary.txt`, it adds these words into the Trie data structure. This addition helps in creating a comprehensive dictionary that the checker can use to determine whether a given word is spelled correctly.
4. **Fetching from Dictionary**: When a user inputs a word, the spelling checker searches the Trie for this word. If the word is not found, it may be flagged as potentially misspelled. The Trie structure allows for efficient searching and retrieval of words, making it faster to find and suggest corrections.
5. **Highlighting Misspelled Words**: If the input word does not match any word in the Trie, the spelling checker identifies it as misspelled. It then uses the Trie to suggest possible correct words based on common prefixes or similar spellings.

In summary, the spelling checker uses `dictionary.txt` to populate a Trie data structure with correctly spelled words. This allows the system to efficiently check spelling and suggest corrections by leveraging common parts of words stored in the Trie.



Explanation of Trie Search Operation

Assume you have a Trie data structure with the following components:

- **Trie Node**: A node in the Trie typically contains:

    - `children`: A map (or dictionary) where each key is a character and each value is another Trie node.
    - `is_end_of_word`: A boolean flag indicating if the node represents the end of a valid word.
- **Trie**: The Trie itself has methods for inserting and searching words.

Here's a simplified example of how the Trie search operation might be implemented in code:

Trie Node Class

```
class TrieNode {
    Map<Character, TrieNode> children;
    boolean is_end_of_word;

    TrieNode() {
        children = new HashMap<>();
        is_end_of_word = false;
    }
}
```

Trie Class with Search Method

```
class Trie {
    private TrieNode root;

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

    // Method to insert a word into the Trie
    public void insert(String word) {
        TrieNode node = root;
        for (char ch : word.toCharArray()) {
            node = node.children.computeIfAbsent(ch, c -> new
TrieNode());
```

```
        }
        node.is_end_of_word = true;
    }

    // Method to search for a word in the Trie
    public boolean search(String word) {
        TrieNode node = root;
        for (char ch : word.toCharArray()) {
            node = node.children.get(ch);
            if (node == null) {
                return false; // Character not found; word is
not in the Trie
            }
        }
        return node.is_end_of_word; // Check if the node repr
esents the end of a valid word
    }
}
```

How It Works

1. **Insertion of Words**:
    - When inserting a word, the code starts at the root of the Trie and iterates through each character of the word.
    - For each character, it checks if there is a corresponding child node.
    - If a child node for the character does not exist, it creates a new node.
    - Once all characters of the word are processed, the `is_end_of_word` flag is set to `true` at the final node to indicate that the word is valid.
2. **Searching for a Word**:
    - To search for a word, the code starts at the root node and processes each character of the word sequentially.
    - For each character, it checks if there is a corresponding child node.

- If a character’s child node is not found, it means the word is not in the Trie, and the search returns `false`.
- If all characters are found, the search checks the `is_end_of_word` flag of the final node to determine if the word is a valid complete word in the Trie. If `true`, the word exists; otherwise, it does not.

Summary

In essence, the Trie data structure allows for efficient searching of words by leveraging its hierarchical organization. The search operation traverses the Trie nodes based on the characters of the word and verifies the presence of the word by checking the end-of-word flag.

This approach ensures that both insertion and search operations are fast and scalable, especially useful for applications like spelling checkers where rapid lookups are crucial.