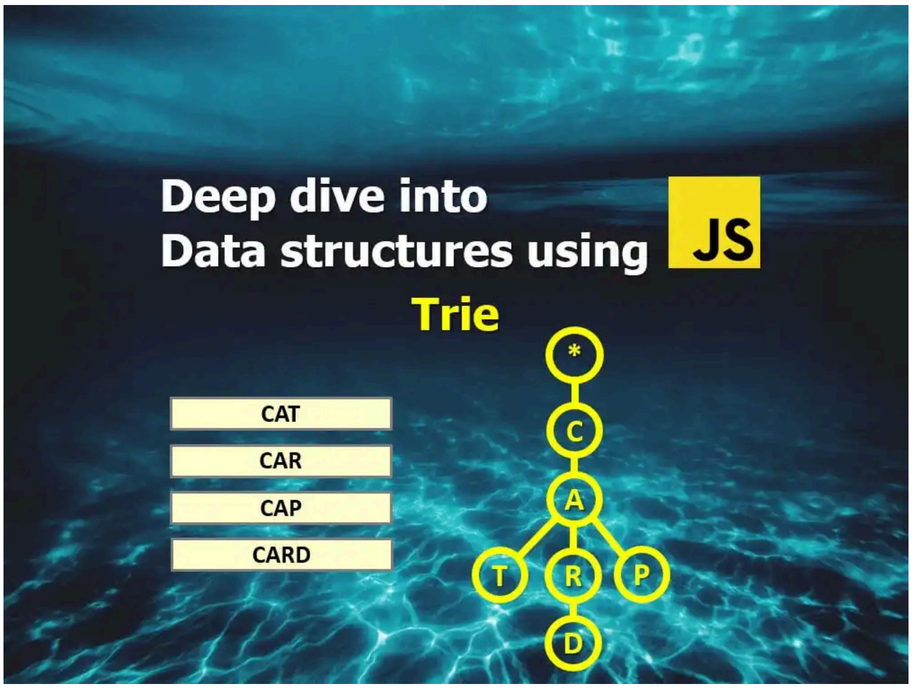
Deep Dive into Data structures using Javascript - Trie



Trie, often referred to as a prefix tree, stands out as a specialized data structure tailored for handling string-based data. Its design is unique among tree structures, focusing on storing associative data that can be accessed quickly and dynamically - which is a necessity for extensive word databases. This makes it particularly suited for applications such as autocomplete systems and spell-check tools, where rapid data retrieval is key.

The real strength of Trie lies in its approach to organizing data. Unlike binary search trees which sort data by node value, Tries are built around character sequences. This structural difference is what sets Trie apart, offering a more direct route for tasks like searching specific strings or prefixes.

This article assumes you're already comfortable with basic tree data structures and JavaScript. If you need a quick refresher, I'd suggest you to start from "Introduction to Trees" article below, then come back and continue here later. Once you're up to speed, you'll find diving into Tries much more straightforward:

[Deep dive into data structures using Javascript - Introduction to Trees]

Anatomy of a Trie

A Trie consists of connected nodes that form a hierarchy, where each node represents a character, and each node can have any number of children nodes, each representing a different character. The Trie starts with a root node, and every node in the tree descends from it. There can be only one root in the Trie and it does not hold any letter of alphabet in it's value property - as it serves as the common starting point which all words in Trie to branch out. Due to this reason it is often initialized with an empty string value.

The Building Blocks: Nodes and Pointers

- **Nodes**: Each node in a Trie represents a single character of a string. Some nodes are flagged to emphasize the end of a word, making it easy to determine when a complete word has been formed.

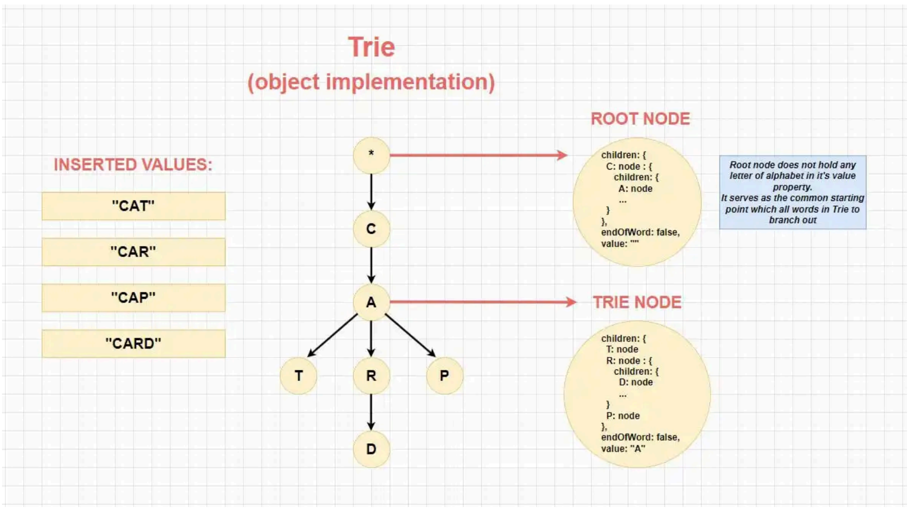
- **Pointers**: These are the connectors between nodes, forming paths from the root to the individual characters of stored words. They are the key concept for navigating through the Trie and for constructing words.

Node Representation: Objects vs. Arrays

When constructing a Trie, there are two primary ways to represent the children of each node: using objects or arrays. We'll be taking a look at the both approaches, and I'll be sharing the implementation for both of them at the end of this article.

Trie using Object based implementation

This approach uses an object to store children nodes. Each key represents a character, and the value is the corresponding child node. This method offers more flexibility and can handle dynamic and larger character sets more efficiently. It also simplifies operations like insertion and search, as it allows direct access to child nodes without needing to calculate index positions.



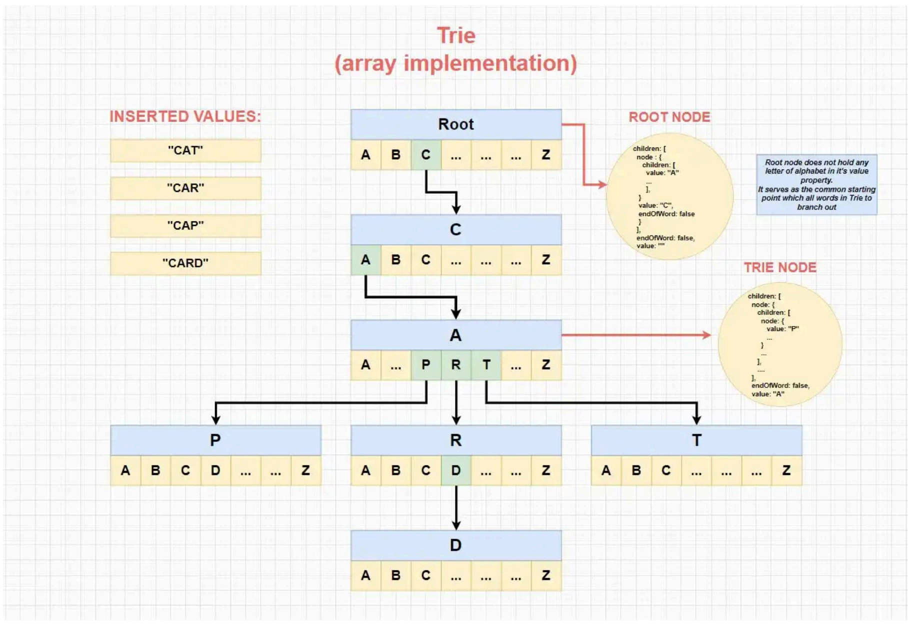
```
children: {
  C: node : {
    children: {
      A: node
      ...
  }
},
endOfWord: false,
value: ""
```

*Root node does not hold any letter of alphabet in it's value property.
It serves as the common starting point which all words in Trie to branch out*

```
children: {
  T: node
  R: node : {
    children: {
      D: node
      ...
  }
  P: node
},
endOfWord: false,
value: "A"
```

Trie using Array based implementation

In this approach, each node is an array where each index corresponds to a character. This can be efficient for fixed-size character sets, like the 26 letters of the English alphabet, but may not be as flexible or space-efficient for larger or dynamic character sets.



```
children: [
 node : {
   children: [
    value: "A"
    ...
    ],
  }
  value: "C",
  endOfWord: false
  }
 ],
 endOfWord: false,
 value: ""
```

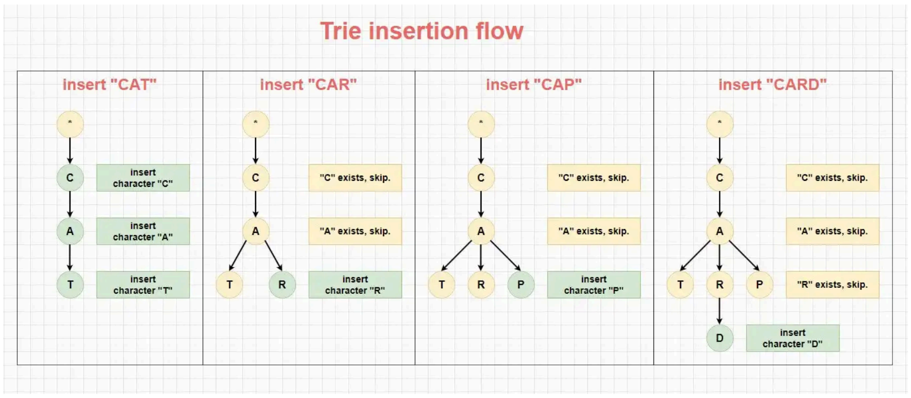
Root node does not hold any letter of alphabet in it's value property.
It serves as the common starting point which all words in Trie to branch out

```
children: [
 node: {
   children: [
    node: {
      value: "P"
      ...
    }
    ...
   ],
   ....
 ],
 endOfWord: false,
 value: "A"
```

The Process of Storing Data

In a Trie, data insertion starts at the root. As each character of a word is processed, the Trie either uses existing nodes (for common prefixes) or creates new ones. This method maximizes space efficiency and speeds up data retrieval, especially noticeable in applications dealing with large datasets.

Take a look at the visual below to see how space efficient insertion flow works in Trie:

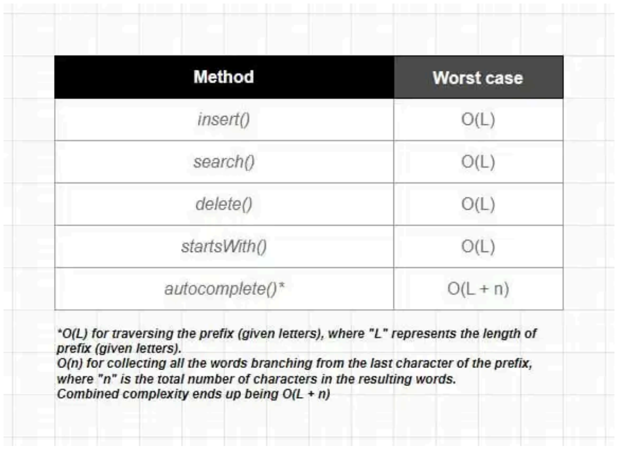


Key Characteristics

- **Space Optimization**: By sharing common prefixes, Tries efficiently utilize space. This is particularly effective for datasets with overlapping strings.
- **Dynamic Adaptability**: The structure of a Trie naturally expands or contracts based on the data it holds.
- **Search Efficiency**: Trie enables quick searches, where the time taken is proportional to the length of the word, rather than the size of the dataset.

When to Use a Trie

Since Trie is a specialized data structure, it’s unique properties makes it favorable in specialized cases. Let’s start with taking a quick look at the Big O of common operations:



Method	Worst case
insert()	O(L)
search()	O(L)
delete()	O(L)
startsWith()	O(L)
*autocomplete()**	O(L + n)

**O(L) for traversing the prefix (given letters), where "L" represents the length of prefix (given letters).*
O(n) for collecting all the words branching from the last character of the prefix, where "n" is the total number of characters in the resulting words.
Combined complexity ends up being O(L + n)

Insertion Operation (Insert): For inserting a new word, the time complexity is O(L), where L is the length of the word. This is because the operation only needs to process each character of the word once.

Search Operation (Search): Searching for a word also has a time complexity of O(L), where L is the length of the word. This efficiency is due to the direct path traversal from the root to the node representing the last character.

Deletion Operation (Delete): Deletion in a Trie can vary, but it is generally O(L) for deleting a word, as it involves traversing the word and potentially cleaning up nodes.

Finding All Words Starts With a Given Prefix: This operation is where Tries truly shine, with an initial O(L) complexity to find the prefix, and then an additional cost proportional to the number of results found.

Autocomplete: Autocomplete operation has a time complexity of O(L + n), where L is the length of the prefix and n is the total number of characters in the resulting words.

Given these time complexity characteristics, Tries become particularly advantageous in scenarios like:

Autocomplete Systems: Tries are ideal for autocomplete systems where the user inputs a few characters and the system suggests possible completions. The Trie's ability to quickly find all words with a given prefix makes it a perfect fit.

Spell Checking and Word Games: With their quick search capability, Tries are well-suited for spell checking and word games where rapid validation of word existence is crucial.

Prefix-Based Search: Any application that requires frequent prefix-based searches, such as searching for contacts in a phone book, can benefit from the structure of a Trie, as it allows for a quick retrieval of all entries under a specific prefix.

Dictionary Implementation: Tries are effective for implementing dictionaries due to their quick insert and search operations for words, and their ability to easily handle dynamic word addition and removal.

IP Routing: In networking, Tries can be used for storing and searching IP routing information efficiently, where IP addresses are viewed as strings of bits.

Bioinformatics Applications: Due to their efficiency in dealing with strings, Tries are used in various bioinformatics applications for pattern matching within DNA sequences.

Trie implementation in Javascript

We will be using ES6 Classes to build our Trie in both object and array based implementations. Here is the list of methods in the implementations:

Object based Trie implementation:

- **insert(word)**: Integrates a new word into the Trie, creating a path of nodes that represents the word if it does not already exist.
- **search(word)**: Confirms whether a word is present in the Trie by traversing the nodes corresponding to the characters of the word.
- **delete(word)**: Removes a word from the Trie, carefully detaching nodes when necessary to avoid disrupting the remaining words.
- **startsWith(prefix)**: Determines if any word in the Trie begins with the given prefix, an essential feature for prefix-based queries.
- **autocomplete(prefix)**: Retrieves a list of all words that stem from the given prefix, providing the backbone for functionalities like search prediction.
- **_deleteRecursively(node, word, index)**: A private helper function to support the delete operation.
- **_findAllWords(node, prefix)**: A private helper function that recursively accumulates all words that start from the given node, used in the autocomplete method.
- **displayAllWords()**: Gathers and returns all words currently stored in the Trie. It traverses the Trie and accumulates complete words, providing a clear view of the Trie's content.
- **displayTrie()**: Executes a breadth-first traversal, yielding the Trie's contents level by level, which is instrumental for visualization and verification during development.

Array based Trie implementation:

- **insert(word)**: Adds a new word to the Trie. It iterates over each character of the word, creating a path of nodes that represents the word. For each character, it converts it to an index and to make sure there is a corresponding TrieNode in the children array. The final node in the path is marked as the end of a word.
- **search(word)**: Checks if a word is present in the Trie. This method traverses the Trie, following the path of nodes corresponding to each character in the word. If all characters are found in sequence, and the final node is marked as the end of a word, the method returns true, indicating the word exists in the Trie.
- **delete(word)**: Removes a word from the Trie. It uses a recursive helper function to traverse the Trie and unset the endOfWord flag at the final node of the word. It also checks whether each node along the path can be removed (i.e., if it has no children and is not part of another word).
- **startsWith(prefix)**: Determines if there is any word in the Trie that starts with the given prefix. It traverses the Trie using the characters of the prefix. If the traversal can continue for the entire length of the prefix, it returns true, indicating that there is at least one word in the Trie that starts with that prefix.
- **autocomplete(prefix)**: Finds all words in the Trie that start with the given prefix. It first finds the node corresponding to the last character of the prefix, then uses a recursive helper function to find and return all words that stem from that node.
- **_charToIndex(char)**: A private helper function that converts a character into an index (0-25). This is used to map each character to a specific index in the children array of a TrieNode.
- **_findAllWords(node, prefix)**: A private recursive helper function that accumulates all words starting from a given node. It is used in the autocomplete method to find all continuations of a prefix, constructing words by appending characters to the prefix as it traverses the Trie.
- **displayAllWords()**: Gathers and returns all words currently stored in the Trie. It traverses the Trie and accumulates complete words, providing a clear view of the Trie's content.
- **displayTrie()**: Performs a level-order traversal of the Trie and displays the contents of the children arrays at each level. This method provides a visualization of the Trie's structure, showing which characters are present at each node's children array.

Tries may seem straightforward, but their underlying power is in the systematic way they handle strings. If you want to see how Trie works in an interactive way, I highly recommend playing around with this great Trie visualizer at the link below:

https://www.cs.usfca.edu/~galles/visualization/Trie.html

Furthermore, I've also included line-by-line explanations for each method in the implementation for you to follow up what is happening in the code. I hope this article helped you to understand what Tries are and how they work! I'd like to encourage you to experiment with the implementations below in your favorite code editor. Thanks for reading!

Object based Trie implementation:

```
class TrieNode {
  constructor(value = "") {
    // Initialize an empty object to hold child nodes.
    this.children = {};
    // Flag to indicate whether this node represents the end of a word.
    this.endOfWord = false;
    // The value property stores the character value for this Trie node.
    this.value = value;
  }
}

class Trie {
  constructor() {
    // The root node, which acts as the
    // starting point of the Trie and holds no character value.
    this.root = new TrieNode();
  }

  // Inserts a word into the trie
  insert(word) {
    // Start at the root node.
    let current = this.root;
    // Iterate over each character in the word.
    for (let char of word) {




      // If the current node doesn't have
      // a child with the current character...
      if (!current.children[char]) {
        // ...create a new TrieNode and assign it
        // to the corresponding character in the children object.
        current.children[char] = new TrieNode(char);
      }
      // Move the current pointer to the
      // child node of the current character.
      current = current.children[char];
    }
    // After inserting the last character of the word,
    // set the endOfWord flag to true.
    current.endOfWord = true;
  }

  // Search for a word in the trie
  search(word) {
    // Start at the root node.
    let current = this.root;
    // Iterate over each character in the word.
    for (let char of word) {
      // If the current node doesn't have a child
      // with the current character...
      if (!current.children[char]) {
        // ...the word doesn't exist in the trie.
        return false;
      }
      // Move the current pointer to the
      // child node of the current character.
      current = current.children[char];
    }
    // Return true if the current node has
    // the endOfWord flag set to true,
    // indicating the end of the word
    // has been reached and the word exists.
    return current.endOfWord;
  }

  // Delete a word from the trie
  delete(word) {
    // Begin the recursive deletion process starting from the root node.
    this._deleteRecursively(this.root, word, 0);
  }

  // Helper function for delete
  _deleteRecursively(node, word, index) {
    // If the end of the word has been reached...
    if (index === word.length) {
      // ...and the current node marks the end of a word...
      if (node.endOfWord) {
        // ...unset the endOfWord flag,
        // indicating that the word is no longer valid.
        node.endOfWord = false;
      }
      // Return true if the node has no children,
      // indicating it can be deleted from its parent.
      return Object.keys(node.children).length === 0;




    }

    // Get the character at the current index of the word.
    const char = word[index];
    // Find the child node that corresponds to the character.
    const childNode = node.children[char];

    // If the child node does not exist...
    if (childNode == null) {
      // ...the word is not present in the Trie,
      // and there's nothing to delete.
      return false;
    }

    // Recursively attempt to delete the next node in the sequence.
    const shouldDeleteChild = this._deleteRecursively(
      childNode,
      word,
      index + 1
    );

    // If the recursive call returns true (the child can be deleted)...
    if (shouldDeleteChild) {
      // ...delete the reference to the child node.
      delete node.children[char];

      // Return true if the current node has no more children
      // and it's not marking the end of another word,
      // signaling that this node can also be potentially deleted.
      return Object.keys(node.children).length === 0 && !node.endOfWord;
    }

    // If the child node should not be deleted, return false,
    // indicating this node still has valid children.
    return false;
  }

  // Check if there is any word in the trie that starts with the given prefix
  startsWith(prefix) {
    // Start at the root node of the Trie.
    let current = this.root;
    // Iterate over each character in the prefix.
    for (let char of prefix) {
      // If the current node does not
      // have a child corresponding to the character...
      if (!current.children[char]) {
        // ...then no words with this prefix exist in the Trie.
        return false;
      }
      // Move to the child node associated with the character.
      current = current.children[char];
    }
    // If all characters in the prefix are found, return true.
    return true;
  }

  // Find all words in the trie that start with the given prefix
  autocomplete(prefix) {



    // Start the search at the root of the Trie.
    let current = this.root;
    // Iterate over each character in the prefix.
    for (let char of prefix) {
      // If a character in the prefix is not found...
      if (!current.children[char]) {
        // ...return an empty array as there are no words with this prefix.
        return [];
      }
      // Proceed to the next node
      // corresponding to the character in the prefix.
      current = current.children[char];
    }
    // Once the end of the prefix is reached,
    // find all words starting from this node.
    return this._findAllWords(current, prefix);
  }

  // Helper function to recursively find all words starting from a given node
  _findAllWords(node, prefix) {
    // Initialize an array to hold all the words found.
    let words = [];
    // If the current node marks the end of a word...
    if (node.endOfWord) {
      // ...add the prefix to the list of words as it's a valid word.
      words.push(prefix);
    }

    // Iterate over each child node.
    for (let char in node.children) {
      // Recursively find all words from the child node,
      // adding the current character to the prefix.
      words = words.concat(
        this._findAllWords(node.children[char], prefix + char)
      );
    }

    // Return the list of words found.
    return words;
  }

  // Method to display all words stored in the Trie
  displayAllWords() {
    // Initialize an array to hold all the words found
    let words = [];

    // Recursive helper function to find all words from a given node
    const findAllWords = (node, currentWord) => {
      // If the current node marks the end of a word...
      if (node.endOfWord) {
        // ...add the current word to the list of words
        words.push(currentWord);
      }

      // Iterate over each child node
      for (let char in node.children) {
        // Recursively call the function for each child node,
        // adding the character to the current word




        findAllWords(node.children[char], currentWord + char);
      }
    };

    // Start the recursive process from the root node with an empty string
    findAllWords(this.root, "");
    return words; // Return the list of words found in the Trie
  }

  // Displays an array that will represent the Trie
  // in level-order, with each sub-array representing a level of the Trie.
  // such as: [ [ 'c' ], [ 'a' ], [ 't', 'r', 'p' ] ]
  displayTrie() {
    // Initialize the array that will hold the level-order representation of the Trie.
    let result = [];
    // Start with a queue containing the root node.
    let queue = [this.root];

    // Continue looping until there are no more nodes to process.
    while (queue.length > 0) {
      // Number of nodes at the current level.
      let currentLevelSize = queue.length;
      // Array to hold the values of nodes at the current level.
      let currentLevel = [];

      // Iterate over all nodes at the current level.
      for (let i = 0; i < currentLevelSize; i++) {
        // Get the next node from the queue.
        let currentNode = queue.shift();
        // If the current node is not the root node...
        if (currentNode !== this.root) {
          // ...add its value to the current level array.
          currentLevel.push(currentNode.value);
        }

        // Iterate over all children of the current node.
        for (let child in currentNode.children) {
          // Add each child node to the queue to process at the next level.
          queue.push(currentNode.children[child]);
        }
      }

      // If the current level has nodes...
      if (currentLevel.length > 0) {
        // ...add the array of values for this level to the result array.
        result.push(currentLevel);
      }
    }

    // Return the level-order representation of the Trie.
    return result;
  }
}

let trie = new Trie();
trie.insert("cat");
trie.insert("cap");
trie.insert("car");




trie.insert("card");
trie.insert("caramel");

trie.autocomplete("car");
trie.delete("car");
trie.autocomplete("ca");

trie.displayAllWords()
trie.displayTrie()
```

Array based Trie implementation:

```
class TrieNode {
  constructor() {
    // A node in the Trie may have a maximum of 26 children,
    // one for each letter of the English alphabet
    this.children = new Array(26);
    this.endOfWord = false; // Indicates whether the node represents the end of a word
  }
}

class Trie {
  constructor() {
    // The root node, which acts as the
    // starting point of the Trie and holds no character value.
    this.root = new TrieNode();
  }
```

```javascript
// Inserts a word into the trie
insert(word) {
  // Start at the root node of the Trie.
  let current = this.root;
  // Iterate over each character in the word.
  for (let char of word) {
    // Convert the character to its corresponding index in the children array.
    const index = this._charToIndex(char);
    // If there is no node at this index...
    if (!current.children[index]) {
      // ...create a new TrieNode at this index.
      current.children[index] = new TrieNode();
    }
    // Move to the node that represents the current character.
    current = current.children[index];
  }
  // Mark the end of the word by setting endOfWord to true at the final node.
  current.endOfWord = true;
}

// Search for a word in the trie
search(word) {
  // Start at the root node of the Trie.
  let current = this.root;
  // Iterate over each character in the word.
  for (let char of word) {
    // Convert the character to its corresponding index in the children array.
    const index = this._charToIndex(char);




    // If there is no node at this index...
    if (!current.children[index]) {
      // ...the word does not exist in the trie.
      return false;
    }
    // Move to the node that represents the current character.
    current = current.children[index];
  }
  // Return true if the final node marks the end of a word,
  // indicating that the word exists in the Trie.
  return current.endOfWord;
}

// Deletes a word from the trie
delete(word) {
  const deleteRecursively = (node, idx) => {
    // Check if the end of the word is reached
    if (idx === word.length) {
      // The word is not in the trie
      if (!node.endOfWord) return false;
      // Mark the endOfWord as false
      node.endOfWord = false;
      // Check if node has no children and can be deleted
      return Object.keys(node.children).every((key) => !node.children[key]);
    }

    // Get the index of the current character
    const charIndex = this._charToIndex(word[idx]);
    // Get the child node corresponding to the current character
```

```
      const childNode = node.children[charIndex];

      // If the child node does not exist, the word is not in the trie
      if (!childNode) return false;

      // Recursively call delete on the child node
      const shouldDeleteChild = deleteRecursively(childNode, idx + 1);

      if (shouldDeleteChild) {
        // Delete the child node if it is no longer needed
        delete node.children[charIndex];
        // Check if the current node has no more children
        // and is not the end of another word
        return (
          !node.endOfWord &&
          Object.keys(node.children).every((key) => !node.children[key])
        );
      }

      // If the child node should not be deleted, return false
      return false;
    };

    // Start the deletion process from the root
    deleteRecursively(this.root, 0);
  }

  // Checks if there is any word in the trie that starts with the given prefix
  startsWith(prefix) {




    // Start at the root node of the Trie
    let current = this.root;
    // Iterate over each character in the prefix
    for (let char of prefix) {
      // Get the index of the current character
      const index = this._charToIndex(char);
      // If the current node doesn't have a child for the given character, return false
      if (!current.children[index]) {
        return false;
      }
      // Move to the child node associated with the current character
      current = current.children[index];
    }
    // All characters in the prefix are found, return true
    return true;
  }

  // Finds all words in the trie that start with the given prefix
  autocomplete(prefix) {
    // Start at the root node
    let current = this.root;
    // Iterate over each character in the prefix
    for (let char of prefix) {
      // Convert the character to its corresponding index
      const index = this._charToIndex(char);
      // If there is no child for this character,
      // return an empty array (no words with this prefix)
      if (!current.children[index]) {
        return [];
```

```javascript
      }
      // Move to the child node for the current character
      current = current.children[index];
    }
    // Call the helper function to find all words starting from the current node
    return this._findAllWords(current, prefix);
  }

  // Helper function to convert a character into an index (0-25)
  _charToIndex(char) {
    // Convert 'a' to 0, 'b' to 1, ..., 'z' to 25
    return char.charCodeAt(0) - "a".charCodeAt(0);
  }

  // Recursive helper function to find all words starting from a given node
  _findAllWords(node, prefix) {
    // Initialize an array to hold words
    let words = [];
    // If this node marks the end of a word, add the prefix to the list of words
    if (node.endOfWord) {
      words.push(prefix);
    }
    // Iterate over each potential child node
    for (let i = 0; i < node.children.length; i++) {
      const child = node.children[i];
      // If the child node exists...
      if (child !== undefined) {
        // Convert the index back to a character
        const char = String.fromCharCode("a".charCodeAt(0) + i);




        // Recursively find words from the child node,
        // adding the current character to the prefix
        words = words.concat(this._findAllWords(child, prefix + char));
      }
    }
    // Return the list of words found
    return words;
  }

  displayAllWords() {
    // Define a recursive function to find all words from a given Trie node
    const findAllWords = (node, prefix) => {
      // Initialize an array to store words
      let words = [];
      // If the current node represents the end of a word,
      // add the prefix to the words array
      if (node.endOfWord) {
        words.push(prefix);
      }
      // Iterate over the possible children of the node
      for (let i = 0; i < node.children.length; i++) {
        // Get the child node at index i
        const child = node.children[i];
        // If the child node exists...
        if (child) {
          // Convert the index to a character
          const char = String.fromCharCode("a".charCodeAt(0) + i);
          // Recursively find all words from the child node
          // and concatenate them with the current prefix
          words = words.concat(findAllWords(child, prefix + char));
        }
      }
      // Return the array of words found from this node
      return words;
    };

    // Start the recursive word finding process from
    // the root node with an empty string as the initial prefix
    return findAllWords(this.root, "");
  }

  // Displays an array that will represent the Trie
  // in level-order, with each sub-array representing a level of the Trie.
  // such as: [ [ 'c' ], [ 'a' ], [ 't', 'r', 'p' ] ]
  // empty indexes are indicated with "null"
  displayTrie() {
    // Initialize an array to hold the level order representation of the Trie
    const result = [];
    // Start with a queue containing the root node and its level
    const queue = [{ node: this.root, level: 0 }];

    // Continue until there are no nodes left to process
    while (queue.length > 0) {
       // Dequeue the next node along with its level
      const { node, level } = queue.shift();

      // Initialize the sub-array for this level if it doesn't already exist
      if (result[level] === undefined) {




        result[level] = [];
      }

      // Flag to check if the current node has any children
      let hasChildren = false;
      // Create an array representing the current node's children
      const childrenRepresentation = new Array(26).fill(null);
      // Iterate through the possible children
      for (let i = 0; i < node.children.length; i++) {
        // If a child node exists at index i...
        if (node.children[i]) {
          // Convert the index to a character
          const char = String.fromCharCode('a'.charCodeAt(0) + i);
          // Place the character in the children representation array
          childrenRepresentation[i] = char;
          // Enqueue the child node along with the next level information
          queue.push({ node: node.children[i], level: level + 1 });
          // Mark that the current node has children
          hasChildren = true;
        }
      }

      // Only add the children representation if the node has children
      if (hasChildren) {
        result[level].push(childrenRepresentation);
      }
    }

    // Return the level order representation of the Trie
    return result;
  }
}

let trie = new Trie();
trie.insert("cat");
trie.insert("cap");
trie.insert("car");
trie.insert("card");
trie.insert("caramel");

trie.autocomplete("car");
trie.delete("car");
trie.autocomplete("ca");

trie.displayAllWords()
trie.displayTrie()
```