Lecture Notes
Data Structures and Algorithms

Tries

Overview

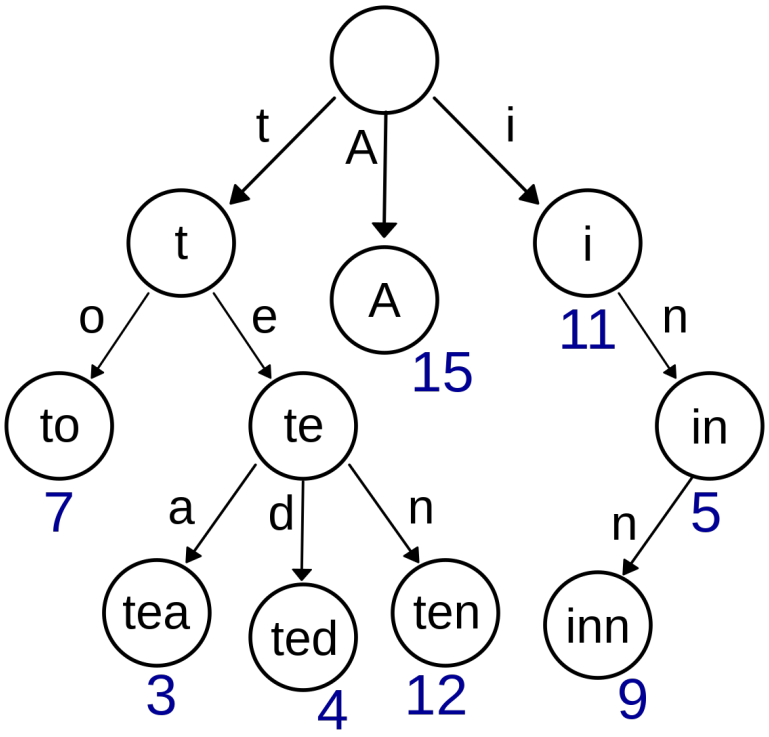

Figure 1: The Trie data structure. Source wikipedia.org

1. Introduction to Tries
    - Definition of a Trie (also known as a prefix tree)
    - Example use cases for Tries (e.g., spelling correction, predictive text input)
2. Trie Data Structure
    - Overview of the Trie node structure (e.g., children nodes, end-of-word marker)
    - Insertion and search operations in a Trie
3. Compressed Tries
    - Definition of a compressed Trie (also known as a Patricia Trie or Radix Tree)
    - Overview of the compressed Trie node structure (e.g., common prefixes stored at the parent node, terminal nodes)
    - Insertion and search operations in a compressed Trie
4. Applications of Tries
    - Implementing a spelling checker using a Trie

    - Implementing predictive text input using a Trie
5. Comparison to Other Data Structures
    - Comparison to hash tables and binary search trees
    - Time and space complexity of Tries
6. Conclusion
    - Recap of the main points covered in the lecture
    - Suggestions for further reading or resources for practicing with Tries

Introduction

- Definition of a Trie: A Trie (also known as a prefix tree) is a tree-like data structure that stores an ordered set of strings. Each node in a Trie represents a single character in a string, and the path from the root to a particular node represents a prefix of a string. The root node represents the empty string, and the children of a node represent all of the possible single-character extensions of the prefix represented by the parent node.

- Example use cases for Tries: Tries are often used to efficiently store and search large sets of strings. Some common examples include:

    - Spelling correction: A Trie can be used to store a large dictionary of words, allowing for fast search and spelling correction by traversing the Trie and checking for the presence of a given word.

    - Predictive text input: A Trie can be used to store a large set of words and phrases, allowing a predictive text input system to quickly suggest completions for a partially-typed word or phrase based on the prefix.

    - IP routing: A Trie can be used to store a large set of IP addresses, allowing a router to quickly determine the next hop for a given destination IP address by traversing the Trie and finding the longest prefix match.

The Trie Data Structure

- Each node in a Trie contains the following fields:

    - An array of children nodes: Each child node represents a single-character extension of the prefix represented by the parent node. The size of the array is typically set to the size of the alphabet being used (e.g., 26 for lowercase English letters).

    - An end-of-word marker: This field is used to indicate whether the prefix represented by the node is the end of a complete word.

- Insertion operation in a Trie: To insert a new string into a Trie, we start at the root node and traverse down the tree, one character at a time. At each step, we check if the current character is already present in the children of the current node. If it is, we move to the child node corresponding to that character. If it is not, we create a new child node for that character and move to that node. We repeat this process for each character in the string, and when we reach the end of the string, we set the end-of-word marker for the final node to indicate that the string is present in the Trie.

- Search operation in a Trie: To search for a string in a Trie, we start at the root node and traverse down the tree, one character at a time, in the same way as we do during insertion. If we reach a null pointer or a non-matching character, the search fails. If we reach the end of the string and the end-of-word marker for the final node is set, the search succeeds. Otherwise, the search fails.

An example Trie ADT as a Java interface:

```
public interface Trie {
    // Insert a key into the Trie
    public void insert(String key);

    // Search for a key in the Trie
    public boolean search(String key);

    // Delete a key from the Trie
    public void delete(String key);

    // Check if the Trie is empty
    public boolean isEmpty();

    // Return the number of keys in the Trie
    public int size();
}
```

The **insert()** method would add a new key to the Trie. It would take a string as an argument and insert it into the Trie by creating a new leaf node for each character in the string and linking the nodes together.

The **search()** method would search for a key in the Trie. It would take a string as an argument and search the Trie for a leaf node with that key. If the key is found, the method would return **true**; otherwise, it would return **false**.

The **delete()** method would delete a key from the Trie. It would take a string as an argument and search the Trie for the leaf node with that key. If the key is found, the method would delete the node and any internal nodes that are no longer needed. If the key is not found, the method would do nothing.

The **isEmpty()** method would check if the Trie is empty. It would return **true** if the Trie is empty, and **false** otherwise.

The **size()** method would return the number of keys in the Trie. It would count the number of leaf nodes in the Trie and return that value.

Compressed Tries

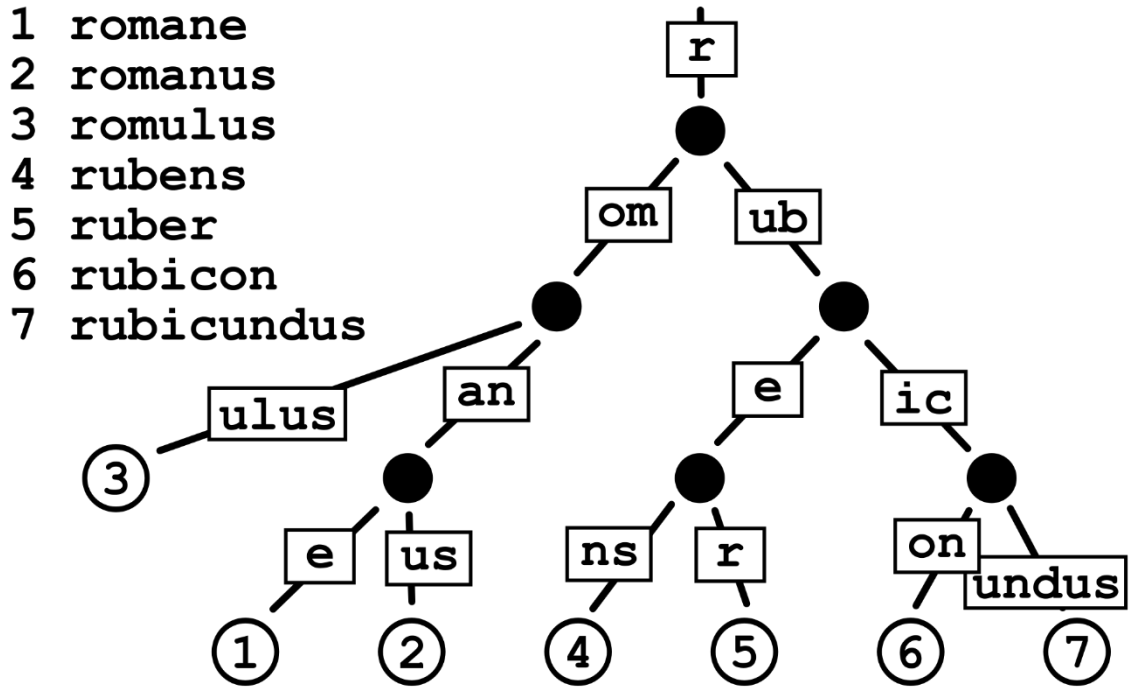

Figure 2: Compressed Trie. Source: wikipedia.org

- A compressed Trie (also known as a Patricia Trie or Radix Tree) is a Trie data structure in which common prefixes are stored at the parent node, rather than at the individual child nodes. This allows for more efficient storage and search, as it reduces the number of nodes needed to represent a large set of strings with common prefixes.
- Overview of the compressed Trie node structure: Each node in a compressed Trie contains the following fields:
    - A list of child nodes: Each child node represents a single-character extension of the prefix represented by the parent node.
    - A string value: This field stores the common prefix shared by all of the child nodes.
    - A boolean value: This field is used to indicate whether the node is a terminal node (representing the end of a complete word).
- Insertion operation in a compressed Trie: To insert a new string into a compressed Trie, we start at the root node and traverse down the tree, one character at a time. At each step, we check if the current character is already present in the children of the current node. If it is, we move to the child node corresponding to that character. If it is not, we create a new child node for that character and move to that node. If the new string has a common prefix with the string value of the current node, we update the string value of the current node to include the common prefix. We repeat this process for each character in the string, and when we reach the end of the string, we set the terminal node marker for the final node to indicate that the string is present in the Trie.

- Search operation in a compressed Trie: To search for a string in a compressed Trie, we start at the root node and traverse down the tree, one character at a time, in the same way as we do during insertion. If we reach a null pointer or a non-matching character, the search fails. If we reach the end of the string and the terminal node marker for the final node is set, the search succeeds. Otherwise, the search fails.

Here is a brief overview of how the **insertion** operation works in a compressed Trie:

1. Start at the root node and traverse down the tree, one character at a time.
2. At each step, check if the current character is already present in the children of the current node.
3. If it is, check for a common prefix between the current string and the string value of the child node.
4. If there is a common prefix, update the string value of the current node to include the common prefix and move to the child node.
5. If there is no common prefix, create a new child node for the current character and move to that node.
6. Repeat this process until you reach the end of the string.
7. Set the terminal node marker for the final node to indicate that the string is present in the Trie.

The compressed insertion algorithm in Java:

```java
import java.util.ArrayList;
import java.util.List;

class CompressedTrie {
    // Nested CompressedTrieNode class
    private static class CompressedTrieNode {
        // A list of child nodes
        private List<CompressedTrieNode> children;

        // A string value
        private String value;

        // A terminal node marker
        private boolean isTerminal;

        // Constructor
        public CompressedTrieNode(String value) {
```

```
            children = new ArrayList<>();
            this.value = value;
            isTerminal = false;
        }
    }

    // The root node
    private CompressedTrieNode root;

    // Constructor
    public CompressedTrie() {
        root = new CompressedTrieNode("");
    }

    // Insert a new string into the CompressedTrie
    public void insert(String s) {
        CompressedTrieNode curr = root;
        int i = 0;
        while (i < s.length()) {
            // Check if the current character is already present in the children of the
            // current node
            boolean found = false;
            for (CompressedTrieNode child : curr.children) {
                if (child.value.charAt(0) == s.charAt(i)) {
                    // Check for a common prefix
                    int j = 1;
                    while (j < child.value.length() && i + j < s.length() && s.charAt(i + j)
== child.value.charAt(j)) {
                        j++;
                    }
                    // Split the child node if there is a common prefix
                    if (j < child.value.length()) {
                        CompressedTrieNode splitNode = new
CompressedTrieNode(child.value.substring(j));
                        splitNode.children = child.children;
                        splitNode.isTerminal = child.isTerminal;
                        child.value = child.value.substring(0, j);
```

```
                    child.children = new ArrayList<>();
                    child.children.add(splitNode);
                    child.isTerminal = false;
                } else {
                    i += j;
                }
                // Move to the child node
                curr = child;
                found = true;
                break;
            }
        }
        if (!found) {
            // Create a new child node
            CompressedTrieNode newNode = new CompressedTrieNode(s.substring(i));
            curr.children.add(newNode);
            curr = newNode;
            break;
        }
    }
    // Set the terminal node marker for the final node
    curr.isTerminal = true;
  }
}
```

The **insert** method in a compressed Trie is used to add a new string to the Trie. Here is a step-by-step explanation of how the **insert** method works:

1. The method starts by initializing a **CompressedTrieNode** variable called **curr** to the root node of the Trie. It also initializes an integer variable called **i** to 0. This variable will be used to keep track of the current position in the string being inserted.
2. The method then enters a loop that continues until **i** becomes greater than or equal to the length of the string. At each iteration of the loop, the method does the following:

a. It checks if the current character in the string (the character at index **i**) is already present in the children of the current node (**curr**).

b. If the current character is present in the children, the method checks for a common prefix between the current string and the string value of the child node. It does this by comparing each character of the

two strings starting at the index immediately following **i**. If a mismatching character is found, or if the end of either string is reached, the method stops checking for a common prefix.

c. If a common prefix is found, the method updates the string value of the current node (**curr**) to include the common prefix. It then moves **curr** to the child node and increments **i** by the length of the common prefix.

d. If no common prefix is found, the method creates a new child node for the current character and sets **curr** to that node. It then breaks out of the loop.

3. After the loop finishes, the method sets the terminal node marker for the final node (**curr**) to **true** to indicate that the string is present in the Trie.

Difference between a regular and a compressed Trie:

1. In a regular Trie, each node represents a single character in the string being inserted. This means that for each character in the string, a new child node must be created. In a compressed Trie, however, each node can represent a longer prefix of the string being inserted. This allows for more efficient storage, as common prefixes are stored at the parent node rather than at individual child nodes.
2. In a regular Trie, the terminal node marker is set at the child node corresponding to the final character of the string being inserted. In a compressed Trie, however, the terminal node marker is set at the final node in the path taken during insertion, which may not be a child node if there are common prefixes present.

Applications of Tries

Example 1. Spell Checking

A Trie can be used to store a dictionary of valid words, and then quickly search for a given word in the dictionary to check if it is spelled correctly. Here is an example of how to implement a simple spell checker using a Trie in Java:

```
import java.util.Scanner;

public class SpellChecker {
    // Nested TrieNode class
    private static class TrieNode {
        // An array of child nodes
        private TrieNode[] children;
        // A terminal node marker
        private boolean isTerminal;
        // Constructor
        public TrieNode() {
```

```
        children = new TrieNode[26];
        isTerminal = false;
    }
}
// The root node
private TrieNode root;
// Constructor
public SpellChecker() {
    root = new TrieNode();
}
// Insert a new word into the Trie
public void insert(String s) {
    TrieNode curr = root;
    for (int i = 0; i < s.length(); i++) {
        int index = s.charAt(i) - 'a';
        if (curr.children[index] == null) {
            curr.children[index] = new TrieNode();
        }
        curr = curr.children[index];
    }
    curr.isTerminal = true;
}
// Check if a word is spelled correctly
public boolean spellCheck(String s) {
    TrieNode curr = root;
    for (int i = 0; i < s.length(); i++) {
        int index = s.charAt(i) - 'a';
        if (curr.children[index] == null) {
            return false;
        }
        curr = curr.children[index];
    }
    return curr.isTerminal;
}
public static void main(String[] args) {
    SpellChecker spellChecker = new SpellChecker();
    // Insert some words into the Trie
```

```
        spellChecker.insert("hello");
        spellChecker.insert("world");
        spellChecker.insert("goodbye");
        // Check the spelling of some words
        Scanner scanner = new Scanner(System.in);
        while (true) {
            System.out.print("Enter a word to check: ");
            String word = scanner.nextLine();
            if (spellChecker.spellCheck(word)) {
                System.out.println("The word is spelled correctly.");
            } else {
                System.out.println("The word is not spelled correctly.");
            }
        }
    }
}
```

Example 2. Autocomplete

A Trie can be used to store a set of strings, and then quickly search for strings that have a given prefix. This can be used to implement an autocomplete feature, where the user can type in a prefix and the Trie will suggest possible completions. Here is an example of how to implement an autocomplete feature using a Trie in Java:

```
import java.util.ArrayList;

public class Autocomplete {
    // Nested TrieNode class
    private static class TrieNode {
        // An array of child nodes
        private TrieNode[] children;

        // A terminal node marker
        private boolean isTerminal;

        // Constructor
        public TrieNode() {
            children = new TrieNode[26];
            isTerminal = false;
```

```
    }
}

// The root node
private TrieNode root;

// Constructor
public Autocomplete() {
    root = new TrieNode();
}

// Insert a new string into the Trie
public void insert(String s) {
    TrieNode curr = root;
    for (int i = 0; i < s.length(); i++) {
        int index = s.charAt(i) - 'a';
        if (curr.children[index] == null) {
            curr.children[index] = new TrieNode();
        }
        curr = curr.children[index];
    }
    curr.isTerminal = true;
}

// Find all strings with a given prefix
public ArrayList<String> find(String prefix) {
    ArrayList<String> results = new ArrayList<>();
    TrieNode curr = root;
    for (int i = 0; i < prefix.length(); i++) {
        int index = prefix.charAt(i) - 'a';
        if (curr.children[index] == null) {
            return results;
        }
        curr = curr.children[index];
    }
    find(curr, prefix, results);
    return results;
```

```java
    }

    // Find all strings with a given prefix (recursive helper function)
    private void find(TrieNode curr, String prefix, ArrayList<String> results) {
        if (curr.isTerminal) {
            results.add(prefix);
        }
        for (int i = 0; i < 26; i++) {
            if (curr.children[i] != null) {
                find(curr.children[i], prefix + (char) ('a' + i), results);
            }
        }
    }

    public static void main(String[] args) {
        Autocomplete autocomplete = new Autocomplete();
        // Insert some strings into the Trie
        autocomplete.insert("hello");
        autocomplete.insert("world");
        autocomplete.insert("goodbye");
        // Find all strings with a given prefix
        if (args.length != 1) {
            System.out.println("Usage: java Autocomplete <prefix>");
        } else {
            String prefix = args[0];
            ArrayList<String> results = autocomplete.find(prefix);
            if (results.isEmpty()) {
                System.out.println("No strings found with prefix '" + prefix + "'.");
            } else {
                System.out.println("Strings found with prefix '" + prefix + "':");
                for (String s : results) {
                    System.out.println(s);
                }
            }
        }
    }
}
```

Example 3. Data Compression

There are a number of data compression algorithms that make use of Tries in order to efficiently encode and decode data. One example is the Huffman coding algorithm, which is used to compress data by assigning shorter codes to more frequently occurring symbols in the data.

The Huffman coding algorithm is a data compression algorithm that assigns shorter codes to more frequently occurring symbols in a given data set, in order to compress the data. It does this by building a binary Trie based on the frequencies of the symbols in the data, and then using the Trie to generate the codes for each symbol.

The idea behind the algorithm is that symbols that occur more frequently in the data should have shorter codes, because they will contribute more to the overall size of the encoded data. For example, if the letter 'e' is the most common letter in the English alphabet, it would make sense to assign it a shorter code than the letter 'z', which is much less common. By doing this, the overall size of the encoded data is reduced, because the more frequently occurring symbols take up less space.

To use the Huffman coding algorithm, you first need to calculate the frequency of each symbol in the data that you want to compress. Then, you can build a Trie based on the frequencies of the symbols, using a priority queue to merge the nodes of the Trie in the correct order. Once the Trie is built, you can use it to generate the code for each symbol by traversing the Trie from the root to the leaf node corresponding to the symbol.

To decode the data, you can simply use the same Trie and the generated code for each symbol to traverse the Trie and reconstruct the original data.

```
  import java.util.PriorityQueue;
import java.util.Scanner;

public class HuffmanCoding {
    // Nested TrieNode class
    private static class TrieNode implements Comparable<TrieNode> {
        // The character represented by this node
        private char c;
        // The frequency of the character
        private int frequency;
        // The left and right child nodes
        private TrieNode left, right;
        // Constructor for an internal node
        public TrieNode(int frequency, TrieNode left, TrieNode right) {
            this.c = '\0';
```

```
            this.frequency = frequency;
            this.left = left;
            this.right = right;
        }
        // Constructor for a leaf node
        public TrieNode(char c, int frequency) {
            this.c = c;
            this.frequency = frequency;
            this.left = null;
            this.right = null;
        }
        @Override
        public int compareTo(TrieNode o) {
            return this.frequency - o.frequency;
        }
    }

    // Build the Huffman Trie
    public static TrieNode buildTrie(int[] charFrequencies) {
        // Create a priority queue of TrieNodes
        PriorityQueue<TrieNode> pq = new PriorityQueue<>();
        for (int i = 0; i < charFrequencies.length; i++) {
            if (charFrequencies[i] > 0) {
                pq.add(new TrieNode((char) i, charFrequencies[i]));
            }
        }
        // Special case for empty input
        if (pq.size() == 0) {
            return null;
        }
        // Merge the TrieNodes until there is only one left
        while (pq.size() > 1) {
            TrieNode left = pq.poll();
            TrieNode right = pq.poll();
            TrieNode parent = new TrieNode(left.frequency + right.frequency, left,
right);
            pq.add(parent);
```

```
        }
        return pq.poll();
    }
    // Build the code table
    public static String[] buildCodeTable(TrieNode root) {
        String[] codeTable = new String[256];
        buildCodeTable(root, "", codeTable);
        return codeTable;
    }
    // Build the code table (recursive helper function)
    private static void buildCodeTable(TrieNode curr, String s, String[] codeTable) {
        if (curr.c != '\0') {
            codeTable[curr.c] = s;
            return;
        }
        buildCodeTable(curr.left, s + '0', codeTable);
        buildCodeTable(curr.right, s + '1', codeTable);
    }

    // Encode a string using the code table
    public static String encode(String s, String[] codeTable) {
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < s.length(); i++) {
            sb.append(codeTable[s.charAt(i)]);
        }
        return sb.toString();
    }
    // Decode a string using the code table
    public static String decode(String s, TrieNode root) {
        StringBuilder sb = new StringBuilder();
        TrieNode curr = root;
        for (int i = 0; i < s.length(); i++) {
            if (s.charAt(i) == '0') {
                curr = curr.left;
            } else {
                curr = curr.right;
            }
```

```
            if (curr.c != '\0') {
                sb.append(curr.c);
                curr = root;
            }
        }
        return sb.toString();
    }
    public static void main(String[] args) {
        // Test the Huffman coding algorithm
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter a string to compress: ");
        String s = scanner.nextLine();
        // Calculate the frequency of each character
        int[] charFrequencies = new int[256];
        for (int i = 0; i < s.length(); i++) {
            charFrequencies[s.charAt(i)]++;
        }
        // Build the Huffman Trie
        TrieNode root = buildTrie(charFrequencies);
        // Build the code table
        String[] codeTable = buildCodeTable(root);
        // Encode the string
        String encoded = encode(s, codeTable);
        // Decode the string
        String decoded = decode(encoded, root);
        // Print the results
        System.out.println("Original string: " + s);
        System.out.println("Encoded string: " + encoded);
        System.out.println("Decoded string: " + decoded);
    }
}
```

The **TrieNode** class is used to represent nodes in the Huffman Trie. It has four fields:

- **c**: the character represented by the node (for leaf nodes) or **'\0'** for internal nodes
- **frequency**: the frequency of the character (for leaf nodes) or the sum of the frequencies of the child nodes (for internal nodes)

- **left**: the left child node
- **right**: the right child node

The **buildTrie()** method is used to build the Huffman Trie based on the frequencies of the symbols in the data. It takes an array **charFrequencies** of frequencies, where the index of the array corresponds to the character and the value at that index corresponds to the frequency of the character.

Complexity and Relation to Other Data Structures

Tries are a specialized data structure that is well suited for storing and searching for strings or other sequences of data. They offer a number of advantages compared to other data structures:

- Fast insertion, search, and deletion: Tries have a time complexity of ***O(L)*** for insertion, search, and deletion, where L is the length of the key being inserted, searched for, or deleted. This is faster than other data structures such as hash tables and balanced binary search trees, which have a time complexity of O(1) or O(log N) on average, but can have a worst-case time complexity of O(N) in some cases.
- Efficient use of space: Tries have a space complexity of ***O(AL)***, where A is the size of the alphabet and L is the maximum length of the keys in the Trie. This is more space-efficient than hash tables, which have a space complexity of O(N), where N is the number of keys in the table, and balanced binary search trees, which have a space complexity of O(N) in the worst case.
- Fast prefix matching: Tries are particularly well suited for prefix matching, where you want to search for all keys that start with a given prefix. This operation can be performed in time ***O(L)***, where L is the length of the prefix, making Tries faster than hash tables and balanced binary search trees, which both have a time complexity of O(N) for prefix matching.
- Compressed Tries can reduce the space complexity by using techniques such as child pointer compression and prefix sharing to reduce the number of nodes in the Trie. This can make them more space-efficient than regular Tries, especially for large data sets with a large alphabet and long keys.

However, Tries also have some limitations compared to other data structures:

- Poor performance for non-string data: Tries are not well suited for storing and searching for non-string data, such as numbers or other data types, because they rely on the structure of the keys to organize the data. Other data structures such as hash tables and balanced binary search trees may be more appropriate for this type of data.
- Poor performance for keys with low diversity: Tries may have poor performance for keys with low diversity, such as keys that are all the same or keys that differ by only a few characters. This is because the Trie will have many nodes with only a few keys, leading to a larger number of nodes and a slower search time. Other data structures such as hash tables may be more appropriate in these cases.

Conclusion

Tries are a specialized data structure that is used to store and search for strings or other sequences of data. They are built using a tree-like structure, with each node representing a prefix of a key, and the leaf nodes representing the complete keys. Tries offer a number of advantages, including fast insertion, search, and deletion, and efficient use of space. They are particularly well-suited for prefix matching and are often used in data compression algorithms such as Huffman coding.

However, Tries also have some limitations, including poor performance for non-string data and keys with low diversity. It is important to consider these limitations and choose the appropriate data structure for the task at hand.

Overall, Tries are a useful tool for storing and searching for string data, and are worth considering when designing algorithms and data structures for applications that involve large amounts of text or other sequence data.