GeeksforGeeks

Related Articles

Compressed Tries

Difficulty Level : Hard • Last Updated : 13 Jan, 2021

A trie is a data structure that stores strings like a tree data structure. The maximum number of children in a node is equal to the size of the alphabet. One can easily print letters in alphabetical order which isn't possible with hashing.

Properties of Trie:

- It's a multi-way tree.
- Each node has from **1** to **N** children.
- Each leaf node corresponds to the stored string, which is a chain of characters on a path from the root to its side.

Types of Trie:

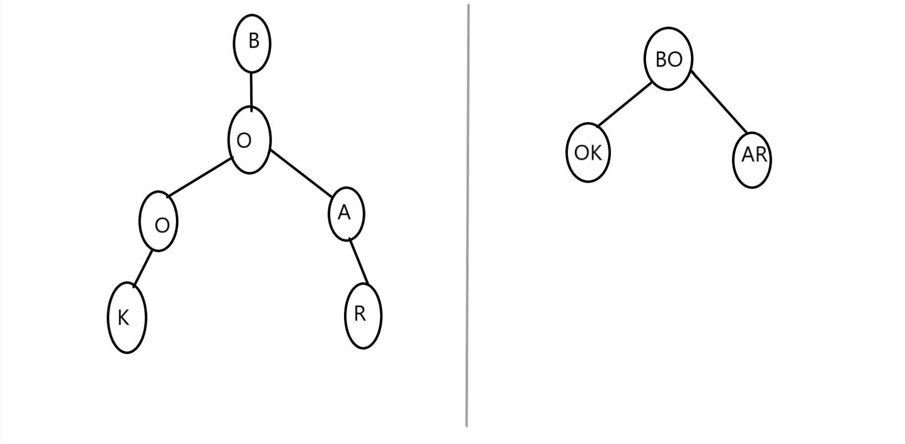
- Standard Trie
- Suffix Trie
- Compressed Trie

Compressed Trie:

Tries with nodes of degree **at least 2**. It is accomplished by compressing the nodes of the standard trie. It is also known as **Radix Tries**. It is used to achieve space optimization

Since the nodes are compressed. Let's visually compare the structure of the Standard tree and the **compressed tree** for a better approach. In terms of memory, a compressed trie tree uses very few amounts of nodes which gives a huge memory advantage(especially for long) strings with long common prefixes. In terms of speed, a regular trie tree would be slightly faster because its operations don't involve any string operations, they are simple loops.

In the below image, the left tree is a Standard trie, the right tree is a compressed trie.



Implementation:

A standard trie node looks like this:

Java

```
class node {
    node[] children = new node[26];
    boolean isWordEnd;
}
```

But for a compressed trie, redesigning of the tree will be as follows, in the general trie, an edge 'a' is denoted by this particular element in the array of references, but in the compressed trie, "An edge 'face' is denoted by this particular element in the array of references". The code is-:

Java

```
class node {
    node[] children = new node[26];
    StringBuilder[] edgeLabel = new StringBuilder[26];
    boolean isEnd;
}
```

Node in Compressed Trie:

Java

```
class CompressedNode {
    int bitNumber;
    int data;
```

Class Compressed trie:

Java

```
class CompressedNode {

    // Root Node
    private CompressedNode root;

    private static final int MaxBits = 10;

    // Constructor
    public CompressedNode() { root = null; }

    // Function to check if empty
    public boolean isEmpty() { return root == null; }

    // Function to clear
    public void makeEmpty() { root = null; }
}
```

Searching in Compressed Trie:

Searching in a compressed Trie tree is much like searching. Here, instead of comparing a single character, we compare strings.

Java

```
// Function to search a key k
// in the trie
public boolean search(int k)
{
    // Find the number of bits
    int numOfBits = (int)(Math.log(k) / Math.log(2)) + 1;

    // If error occurs
    if (numOfBits > MaxBits) {
        System.out.println("Error : Number too large");
        return false;
    }

    // Search Node
    CompressedNode searchNode = search(root, k);

    // If the data matches
    if (searchNode.data == k)
        return true;

    // Else return false
    else
        return false;
}
```

Inserting an element in Compressed Trie:

Java

```
// Function to implement the insert
// functionality in the trie
private CompressedNode insert(
    CompressedNode t, int ele)
{
    // If Node is NULL
    if (t == null) {
        t = new CompressedNode();
        t.bitNumber = 0;
        t.data = ele;
        t.leftChild = t;
        t.rightChild = null;
        return t;
    }

    // Search the key ele
    lastNode = search(t, ele);

    // If already present key
    if (ele == lastNode.data) {
        System.out.println(
            "Error : key is already present\n");
        return t;
    }

    for (i = 1; bit(ele, i) == bit(lastNode.data, i); i++)
        ;

    current = t.leftChild;
    parent = t;
    while (current.bitNumber > parent.bitNumber
           && current.bitNumber < i) {
        parent = current;
        current = (bit(ele, current.bitNumber))
                      ? current.rightChild
                      : current.leftChild;
    }

    newNode = new CompressedNode();
    newNode.bitNumber = i;
    newNode.data = ele;
    newNode.leftChild = bit(ele, i) ? current : newNode;
    newNode.rightChild = bit(ele, i) ? newNode : current;

    if (current == parent.leftChild)
        parent.leftChild = newNode;
    else
        parent.rightChild = newNode;

    return t;
}
```

Below is the program to implement all functionality of the compressed Trie:

Java

```
// Java program to implement the
// Compressed Trie

class Trie {

    // Root Node
    private final Node root = new Node(false);

    // 'a' for lower, 'A' for upper
    private final char CASE;

    // Default case
    public Trie() { CASE = 'a'; }
    public Trie(char CASE)
    {
        this.CASE = CASE;
    }

    // Function to insert a word in
    // the compressed trie
    public void insert(String word)
    {
        // Store the root
        Node trav = root;
        int i = 0;

        // Iterate i less than word
        // length
        while (i < word.length()
               && trav.edgeLabel[word.charAt(i) - CASE]
                      != null) {

            // Find the index
            int index = word.charAt(i) - CASE, j = 0;
            StringBuilder label = trav.edgeLabel[index];

            // Iterate till j is less
            // than label length
            while (j < label.length() && i < word.length()
                   && label.charAt(j) == word.charAt(i)) {
                ++i;
                ++j;
            }

            // If is the same as the
            // label length
            if (j == label.length()) {
                trav = trav.children[index];
            }
            else {

                // Inserting a prefix of
                // the existing word
                if (i == word.length()) {
                    Node existingChild
                        = trav.children[index];
                    Node newChild = new Node(true);
                    StringBuilder remainingLabel
                        = strCopy(label, j);

                    // Making "faceboook"
                    // as "face"
                    label.setLength(j);

                    // New node for "face"
                    trav.children[index] = newChild;
                    newChild
                        .children[remainingLabel.charAt(0)
                                  - CASE]
                        = existingChild;
                    newChild
                        .edgeLabel[remainingLabel.charAt(0)
                                   - CASE]
                        = remainingLabel;
                }
                else {

                    // Inserting word which has
                    // a partial match with
                    Node newChild = new Node(false);
                    StringBuilder remainingWord
                        = strCopy(word, i);

                    // Store the trav in
                    // temp node
                    Node temp = trav.children[index];

                    label.setLength(j);
                    trav.children[index] = newChild;
                    newChild
                        .edgeLabel[remainingLabel.charAt(0)
                                   - CASE]
                        = remainingLabel;
                    newChild
                        .children[remainingLabel.charAt(0)
                                  - CASE]
                        = temp;
                    newChild
                        .edgeLabel[remainingWord.charAt(0)
                                   - CASE]
                        = remainingWord;
                    newChild
                        .children[remainingWord.charAt(0)
                                  - CASE]
                        = new Node(true);
                }

                return;
            }
        }

        // Insert new node for new word
        if (i < word.length()) {
            trav.edgeLabel[word.charAt(i) - CASE]
                = strCopy(word, i);
            trav.children[word.charAt(i) - CASE]
                = new Node(true);
        }
        else {

            // Insert "there" when "therein"
            // and "thereafter" are existing
            trav.isEnd = true;
        }
    }

    // Function that creates new String
    // from an existing string starting
    // from the given index
    private StringBuilder strCopy(
        CharSequence str, int index)
    {
        StringBuilder result
            = new StringBuilder(100);

        while (index != str.length()) {
            result.append(str.charAt(index++));
        }

        return result;
    }

    // Function to print the Trie
    public void print()
    {
```

```
    // Fuction to print the word
    // starting from the given node
    private void printUtil(
        Node node, StringBuilder str)
    {
        if (node.isEnd) {
            System.out.println(str);
        }

        for (int i = 0;
             i < node.edgeLabel.length; ++i) {

            // If edgeLabel is not
            // NULL
            if (node.edgeLabel[i] != null) {
                int length = str.length();

                str = str.append(node.edgeLabel[i]);
                printUtil(node.children[i], str);
                str = str.delete(length, str.length());
            }
        }
    }

    // Function to search a word
    public boolean search(String word)
    {
        int i = 0;

        // Stores the root
        Node trav = root;

        while (i < word.length()
               && trav.edgeLabel[word.charAt(i) - CASE]
                      != null) {
            int index = word.charAt(i) - CASE;
            StringBuilder label = trav.edgeLabel[index];
            int j = 0;

            while (i < word.length()
                   && j < label.length()) {

                // Character mismatch
                if (word.charAt(i) != label.charAt(j)) {
                    return false;
                }

                ++i;
                ++j;
            }

            if (j == label.length() && i <= word.length()) {

                // Traverse further
                trav = trav.children[index];
            }
            else {

                // Edge label is larger
                // than target word
                // searching for "face"
                // when tree has "facebook"
                return false;
            }
        }
        return i == word.length() && trav.isEnd;
    }

    // Function to search the prefix
    public boolean startsWith(String prefix)
    {
        int i = 0;

        // Stores the root
        Node trav = root;

        while (i < prefix.length()
               && trav.edgeLabel[prefix.charAt(i) - CASE]
                      != null) {
            int index = prefix.charAt(i) - CASE;
            StringBuilder label = trav.edgeLabel[index];
            int j = 0;

            while (i < prefix.length()
                   && j < label.length()) {

                // Character mismatch
                if (prefix.charAt(i) != label.charAt(j)) {
                    return false;
                }

                ++i;
                ++j;
            }

            if (j == label.length()
                && i <= prefix.length()) {

                // Traverse further
                trav = trav.children[index];
            }
            else {

                // Edge label is larger
                // than target word,
                // which is fine
                return true;
            }
        }

        return i == prefix.length();
    }
}

// Node class
class Node {

    // Number of symbols
    private final static int SYMBOLS = 26;
    Node[] children = new Node[SYMBOLS];
    StringBuilder[] edgeLabel = new StringBuilder[SYMBOLS];
    boolean isEnd;

    // Function to check if the end
    // of the string is reached
    public Node(boolean isEnd)
    {
        this.isEnd = isEnd;
    }
}
```

```
    public static void main(String[] args)
    {
        Trie trie = new Trie();

        // Insert words
        trie.insert("facebook");
        trie.insert("face");
        trie.insert("this");
        trie.insert("there");
        trie.insert("then");

        // Print inserted words
        trie.print();

        // Check if these words
        // are present or not
        System.out.println(
            trie.search("there"));
        System.out.println(
            trie.search("therein"));
        System.out.println(
            trie.startsWith("th"));
        System.out.println(
            trie.startsWith("fab"));
    }
}
```

Output:

```
face
facebook
then
there
this
true
false
true
false
```

Like 0

Previous Next

RECOMMENDED ARTICLES

Page : 1 2

01 Types of Tries
15, Apr 20

02 How to create new elements with ReactJS mapping props ?
30, Apr 21

03 How to write Into a File in PHP ?
30, Apr 21

04 How to transform child elements preserve the 3D transformations ?
30, Apr 21

05 How to set the order of the flexible items using CSS ?
30, Apr 21

06 How to escape everything in a block in HTML ?
30, Apr 21

07 How to detect device and swap the CSS file ?
30, Apr 21

08 How to convert String to Float in PHP ?
30, Apr 21

Article Contributed By :

narutouzamaki120316
@narutouzamaki120316

Vote for difficulty

Current difficulty : Hard

Easy Normal Medium Hard Expert

Article Tags : Picked, Technical Scripter 2020, Trie, Advanced Data Structure, Data Structures, Technical Scripter, Tree

Practice Tags : Data Structures, Tree, Trie

Improve Article Report Issue

Writing code in comment? Please use ide.geeksforgeeks.org, generate link and share the link here.

Load Comments

5th Floor, A-118,
Sector-136, Noida, Uttar Pradesh - 201305

feedback@geeksforgeeks.org

Company

About Us

Careers

Privacy Policy

Contact Us

Copyright Policy

Learn

Algorithms

Data Structures

Languages

CS Subjects

Video Tutorials

Practice

Courses

Company-wise

Topic-wise

How to begin?

Contribute

Write an Article

Write Interview Experience

Internships

Videos