TREE DATA STRUCTURES (HTTP://THEORYOFPROGRAMMING.COM/CATEGORY/TREE-DATA-STRUCTURES/)

Compressed Trie Tree

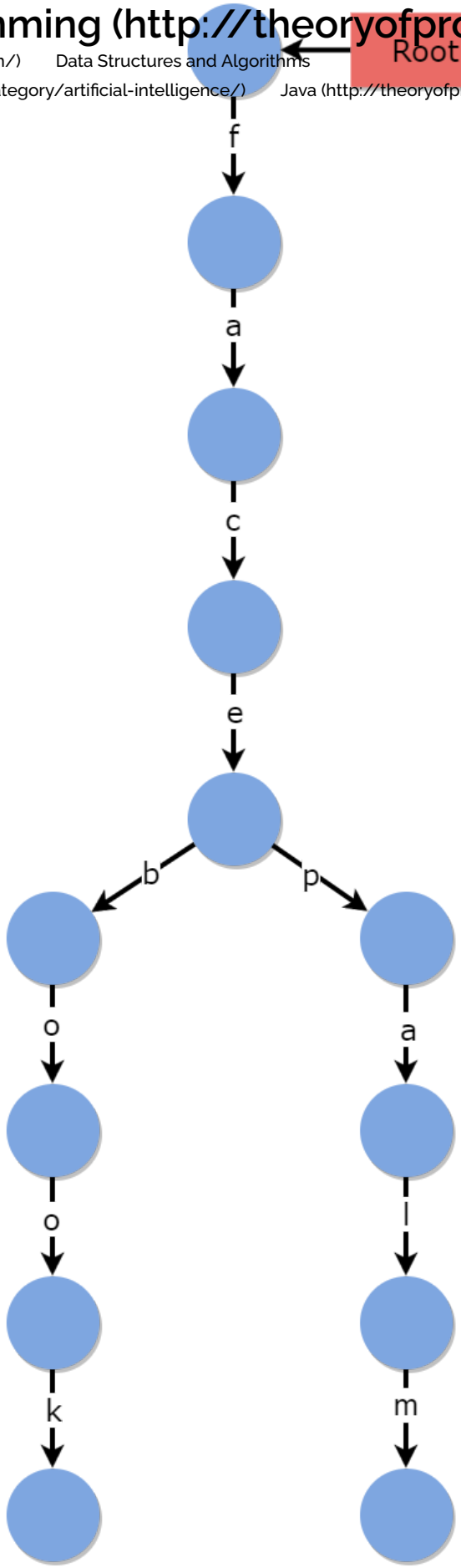
POSTED NOVEMBER 15, 2016 (HTTP://THEORYOFPROGRAMMING.COM/2016/11/15/COMPRESSED-TRIE-TREE/) VAMSI SANGAM (HTTP://THEORYOFPROGRAMMING.COM/AUTHOR/VAMSI-SANGAM/)

Hello, people! In this post, we will discuss a commonly used data structure to store strings, the Compress Trie Tree, also known as Radix Tree or Patricia (*Practical Algorithm to Retrieve Information Coded in Alphanumeric*) Tree. If you remember, the problem with a Trie Tree (http://theoryofprogramming.azurewebsites.net/2015/01/16/trie-tree-implementation/) is that it consumes a lot of memory. So, due to its memory consumption, developers prefer using a compressed trie tree to get the same functionality at the same runtime complexity in memory-critical situations such as in android apps.

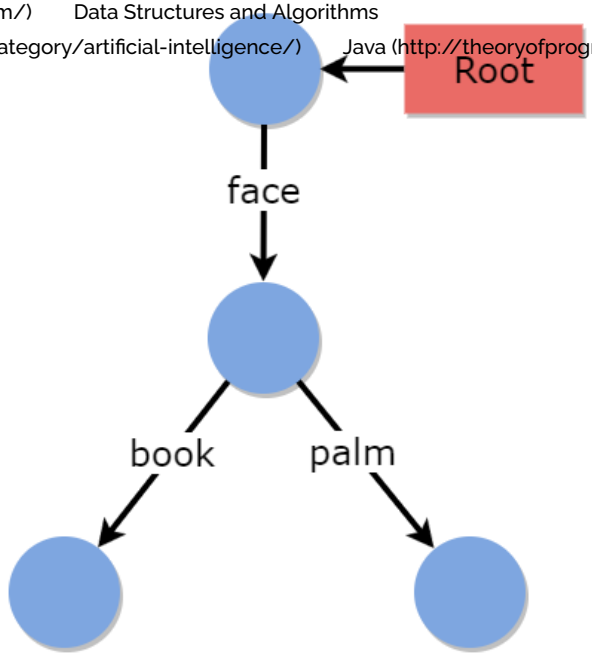
“Compress”-ing the tree

Now, the idea of the compressed trie tree is to convert long chains of single-child edges to one single edge. Example, suppose we have two words “facebook” and “facepalm” in our regular trie tree, it would look like –

Theory of Programming (http://theoryofprogramming.com/)



It's a pretty long one, isn't it? Now, how about this one below?



This one surely looks a lot compact! This is the compressed trie tree. As you can see what we do here is to make long chains of single-child edges into one edge. Unlike a regular trie tree, in a compressed trie tree, the edge labels are strings.

Node in Compressed Trie Tree

For a regular trie tree, our tree node looked something like this –

```
1  class Node {
2      Node[] children = new Node[26];
3      boolean isWordEnd;
4  }
```

So, in every node, we had an array of references. The first references corresponded to 'a', the second to 'b', and so on. So, essentially, we had a mapping of alphabets to references. We had a way of saying, "An edge 'a' is denoted by this particular element in the array of references". Now, in a compressed trie tree, the edges are labeled by strings. Now, we need a way of saying, "An edge 'face' is denoted by this particular element in the array of references". To accomplish this, we re-design our tree node as follows –

```
1  class Node {
2      Node[] children = new Node[26];
3      StringBuilder[] edgeLabel = new StringBuilder[26];
4      boolean isEnd;
5  }
```

So, what we did is that we added an additional array of Strings along with the array of references. Now **edgeLabel[0]** will denote the string starting with 'a', **edgeLabel[1]** will denote the string starting with 'b', and correspondingly, **children[0]** will denote the edge with the label **edgeLabel[0]**.

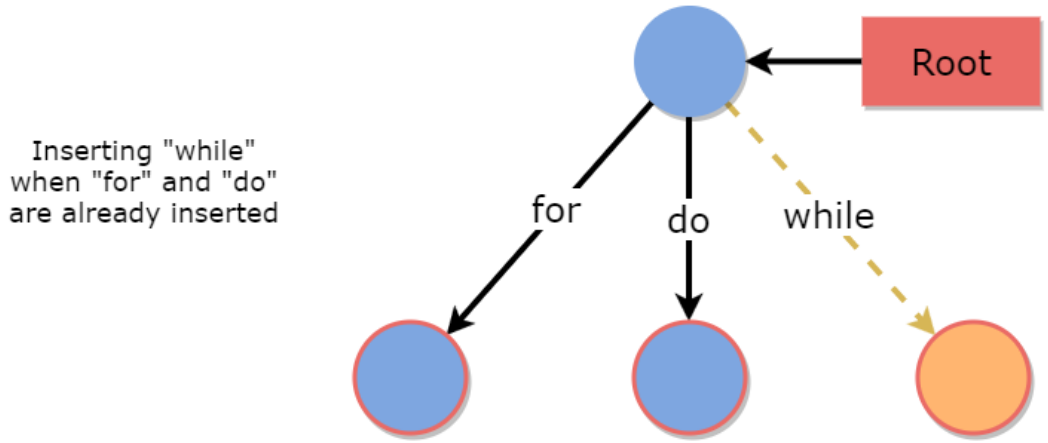
Example, in the above diagram, the root node will have **edgeLabel[5]** = "face" and **children[5]** will point to the internal node. The internal node will have **edgeLabel[1]** = "book" and **children[1]** will point to the leaf node which will denote the occurrence of the word "facebook". The same internal node will also have **edgeLabel[15]** = "palm" and **children[15]** will point to the leaf node which will denote the occurrence of the word "facepalm". The rest of the values of **edgeLabel** and **children** in the internal node will be null.

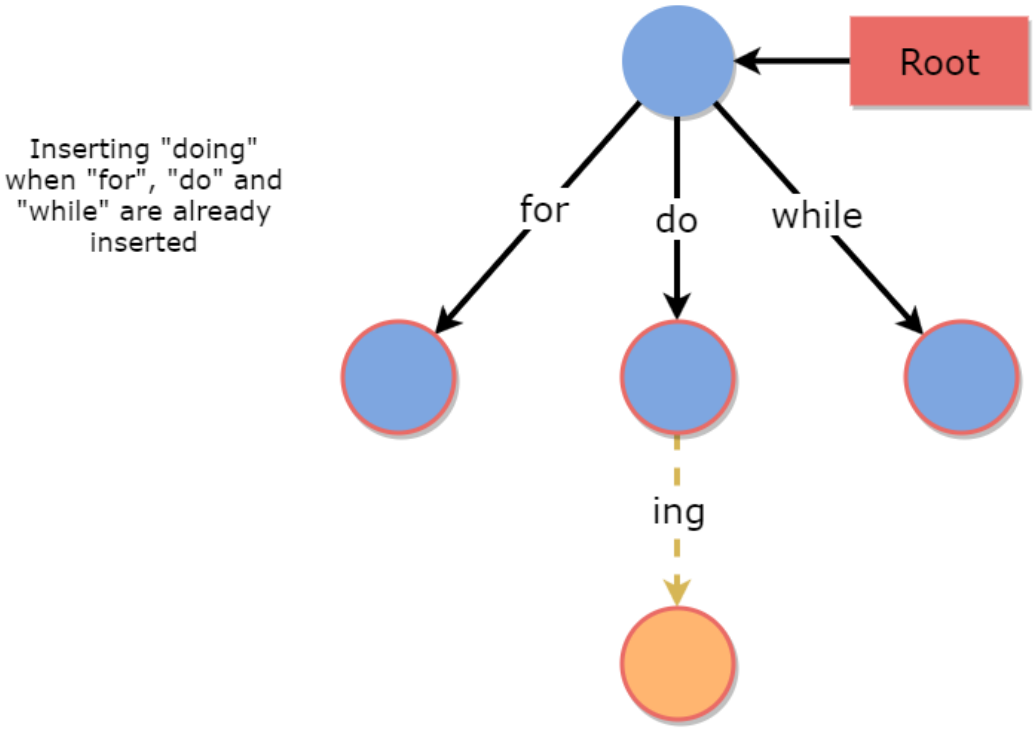
The above code is written in Java. For Java, it is much better to use StringBuilder rather than String because we would be doing a lot of string manipulation. Using String will make your program slow. You can read more about it in my post (http://theoryofprogramming.azurewebsites.net/2015/06/05/java-tutorials-strings-string-buffer-builder-in-java/).

insert() operation

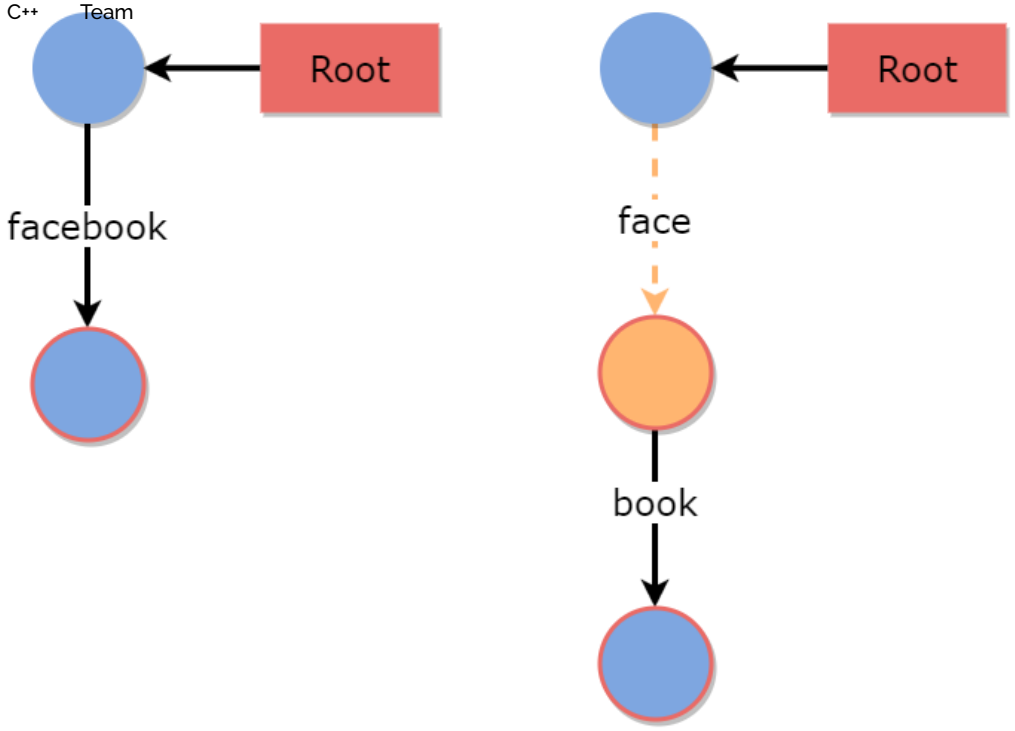
All operations in the compressed trie tree are similar to what we would do in a regular trie tree. Insert operation is the one which will differ the most. In the insert operation, we need to take care of a few cases, they are –

1. Inserting new node for a new word – This occurs when the starting character of the word is new and there's no edge corresponding to it. This may occur at root, or after traversing to an internal node.



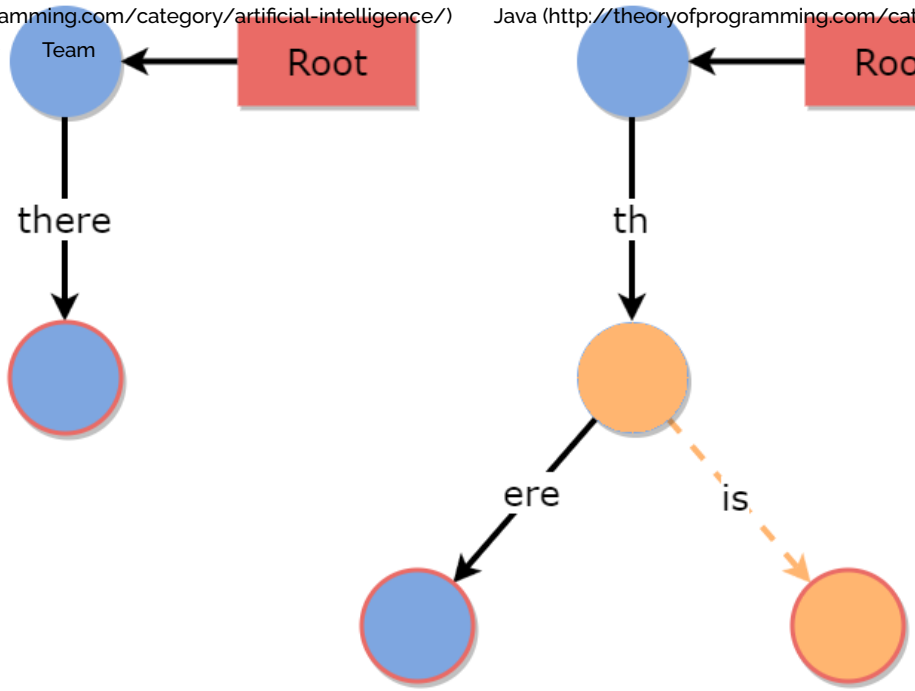


2. Inserting a prefix of an existing word –

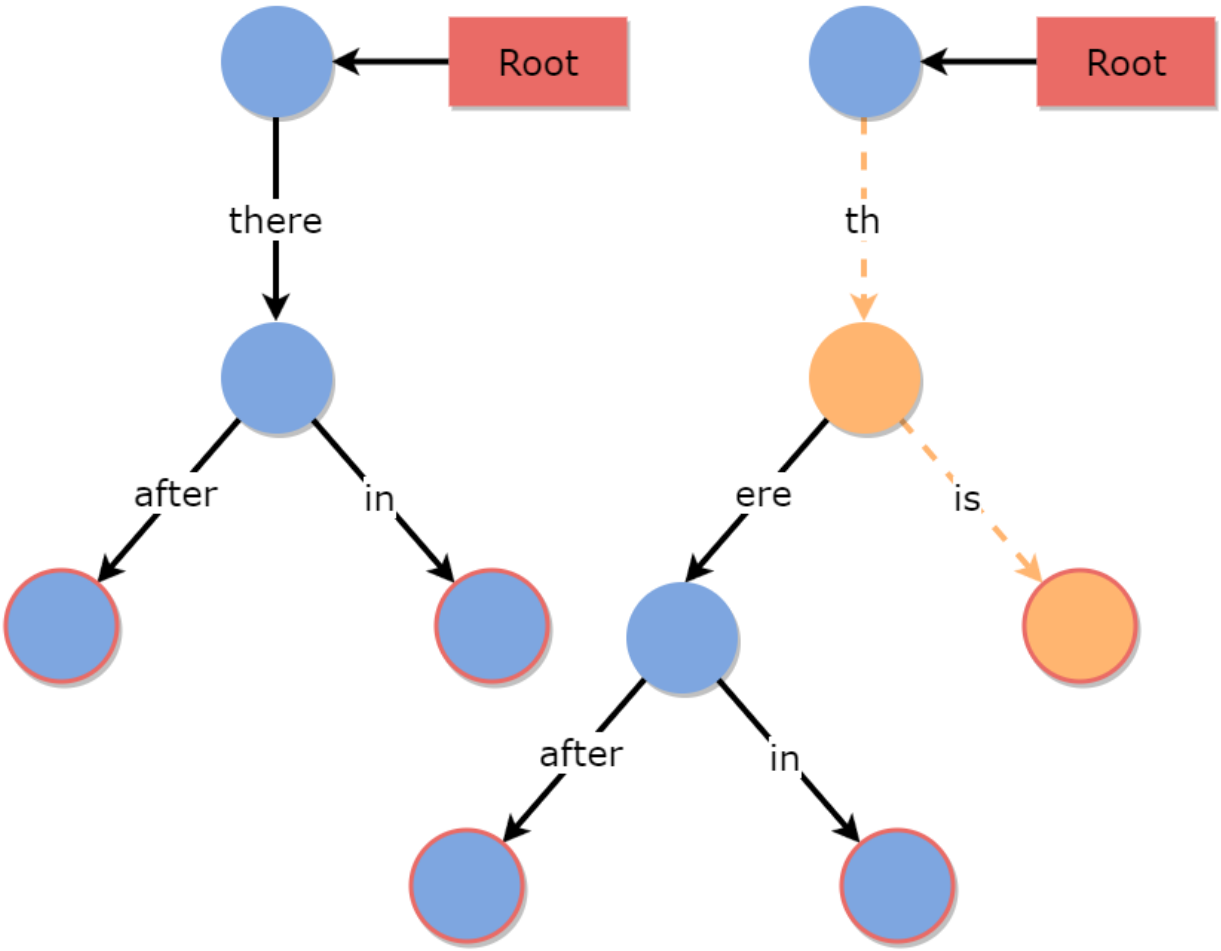


3. Inserting a word which has a partial match to an existing edge – This occurs when we are trying to insert “this” when “there” is already inserted into the tree. Remember that “there” can have further children too, like if “thereafter” and “therein” are already inserted.

Theory of Programming (http://theoryofprogramming.com/)



Inserting "this" when "thereafter" and "therein" are already inserted



(http://theoryofprogramming.azurewebsites.net/wp-content/uploads/2016/11/compressed-trie-tree-7.png)

So, for these cases, we would have to break the existing word of the newly inserted word accordingly. The faster we perform these string operations, the faster the insert operation will be.

search() operation

Searching in a compressed trie tree is much like searching. Here, instead of comparing a single character, we compare strings. The following cases will arise –

- The string we want to search does not exist. Example, searching for “owl” when the tree only has “crow” in it.
- The string we want to search exists as a prefix. Example, searching for “face” when the tree only has “facebook”.
- Only the prefix of the target string exists. Converse of the previous case. Searching for “facebook” when the tree only has “face”.
- The string we want to search matches partially with an existing string. Example, searching for “this” where the tree only has “there”.
- Another case is when the edge label fully matches to the starting portion of the string. Example, searching for “thereafter” when “thereafter” and “therein” exist in the tree. For this, after a full match with “there”, we traverse to the node which corresponds to that label and then resume our search (searching for “after”).

If we are able to fully traverse the tree via the target string and arrive on a tree node, we check if that node is a word ending or not. If yes, we return true or, we return false. For rest of the cases, return false.

startsWith() operation

The startsWith() operation is a popular operation performed on the compressed trie tree which checks if there’s any word in the tree which starts with the target word. This method would be exactly as the searching method. The minor change with the search operation would be, in this operation, we will just check if we are able to fully traverse the tree via the target string and arrive on a node (which may be the root). If we can we return true, regardless of whether the current node is a word ending or not. This is because, even if it is not a word ending, its children will lead to a node which would be a word ending.

Printing the compressed trie tree

For each edge traversed, add its label to a variable and recursively do this for the child node. While traversing another edge, remove the previously added label and add the new label of the new edge traversing. Print the variable only if the current node is a word ending.

This recursive method should print all the words in the compressed trie tree in a lexicographical order.

Code

Start with your existing trie tree code, modify the node definition and then work on the insert method cases. Once you get the insert correctly, then the rest will work out easily. For the insert cases, you just have to do the string operations and change the references carefully. Try to code those cases. Come back and have a look at the diagrams if you need to.

You can check your code’s correctness with LeetCode’s Implement Trie (https://leetcode.com/problems/implement-trie-prefix-tree/) problem. Try to solve that question using a compressed trie tree. Once you solve it, try to reduce the runtime of your program.

You can refer to my code if you get stuck. 🙂
(https://github.com/VamsiSangam/theoryofprogramming/blob/master/Tree%20Data%20Structures/Compressed%20Trie%20Tree/Java/Trie.java)

This is the Java implementation. I will update this post with the C/C++ implementation soon.

In terms of runtime complexity, compressed trie tree is same as that of a regular trie tree. In terms of memory, a compressed trie tree uses very few amount of nodes which makes it memory efficient. Advantage is clear when long strings with long common prefixes. In terms of speed, a regular trie tree would be slightly faster because it operations don't involve any string operations, they are simple loops.

Theory of Programming (http://theoryofprogramming.com/)

Home (http://theoryofprogramming.com/) Data Structures and Algorithms (http://theoryofprogramming.com/category/data-structures-and-algorithms/)

AI (http://theoryofprogramming.com/category/artificial-intelligence/) Java (http://theoryofprogramming.com/category/java/)

In this post, I have demystified everything regarding a compressed trie tree. Tutorial and code for a compressed trie tree are hard to find. I hope my post saved you the effort of finding further tutorials. Do comment below if you have any doubts. Keep practising! Happy Coding!! 😀

Related

Trie Tree Implementation (http://theoryofprogramming.com/2015/01/…tree-implementation/)
January 16, 2015
In "Tree Data Structures"

Trie Tree Practise - SPOJ - DICT (http://theoryofprogramming.com/2015/09/…tree-practise-spoj-dict/)
September 1, 2015
In "Competitive Coding"

Trie Tree Practise - SPOJ - PHONELST (http://theoryofprogramming.com/2015/08/…tree-practise-spoj-phonelst/)
August 24, 2015
In "Competitive Coding"

COMPRESSED TRIE TREE (HTTP://THEORYOFPROGRAMMING.COM/TAG/COMPRESSED-TRIE-TREE/)

DATA STRUCTURES (HTTP://THEORYOFPROGRAMMING.COM/TAG/DATA-STRUCTURES/)

PATRICIA TREE (HTTP://THEORYOFPROGRAMMING.COM/TAG/PATRICIA-TREE/) RADIX TREE (HTTP://THEORYOFPROGRAMMING.COM/TAG/RADIX-TREE/)

STRINGS (HTTP://THEORYOFPROGRAMMING.COM/TAG/STRINGS/) TRIE TREE (HTTP://THEORYOFPROGRAMMING.COM/TAG/TRIE-TREE/)

⟵ Writing Professional Code in C++ (http://theoryofprogramming.com/2016/11/14/writing-professional-code-c/)

Java Tutorials – Inheritance (http://theoryofprogramming.com/2017/12/02/inheritance-in-java/) ⟶

4 thoughts on “Compressed Trie Tree”

Kalyankumar Ramaseshan says:
JUNE 26, 2019 AT 9:01 AM (HTTP://THEORYOFPROGRAMMING.COM/2016/11/15/COMPRESSED-TRIE-TREE/#COMMENT-29916)

where is the code?

REPLY

Vamsi Sangam (http://vamsisangam.com) says:
JUNE 26, 2019 AT 9:16 AM (HTTP://THEORYOFPROGRAMMING.COM/2016/11/15/COMPRESSED-TRIE-TREE/#COMMENT-29918)

I have edited the post giving the link to code which is kept in my Git repo.

The recent migration has caused glitches. Sorry for the inconvenience.

REPLY

nicky kumari (https://plus.google.com/106799023216754752723) says:
APRIL 18, 2018 AT 3:20 PM (HTTP://THEORYOFPROGRAMMING.COM/2016/11/15/COMPRESSED-TRIE-TREE/#COMMENT-17975)

Hi,

How to find all the entries which prefix is “str”

REPLY

Vamsi Sangam (http://vamsisangam.com) says:

APRIL 19, 2018 AT 11:29 PM (HTTP://THEORYOFPROGRAMMING.COM/2016/11/15/COMPRESSED-TRIE-TREE/#COMMENT-18015)

You can refer to this http://theoryofprogramming.azurewebsites.net/2015/09/01/trie-tree-practise-spoj-dict/ (http://theoryofprogramming.azurewebsites.net/2015/09/01/trie-tree-practise-spoj-dict/) …. It tries to print all the nodes with a certain prefix.

REPLY

Leave a Reply

Enter your comment here...

Theory of Programming (http://theoryofprogramming.com/)

RECENT POSTS

Bidirectional Search (http://theoryofprogramming.com/2018/01/21/bidirectional-search/) January 21, 2018

Iterative Deepening Depth First Search (IDDFS) (http://theoryofprogramming.com/2018/01/14/iterative-deepening-depth-first-search-iddfs/) January 14, 2018

N-ary tree or K-way tree data structure (http://theoryofprogramming.com/2018/01/14/n-ary-tree-k-way-tree-data-structure/) January 14, 2018

Rotate matrix clockwise (http://theoryofprogramming.com/2017/12/31/rotate-matrix-clockwise/) December 31, 2017

Print matrix in spiral order (http://theoryofprogramming.com/2017/12/31/print-matrix-in-spiral-order/) December 31, 2017

Theory of Programming (http://theoryofprogramming.com/)

LIKE THIS WEBSITE ON FACEBOOK..! (HTTPS://WWW.FACEBOOK.COM/PAGES/THEORY-OF-PROGRAMMING/886385758049299)

Theory of Programming (http://theoryofprogramming.com/)

SUBSCRIBE TO BLOG VIA EMAIL

Enter your email address to subscribe to this blog and receive notifications of new posts by email.

Join 163 other subscribers

Email Address

SUBSCRIBE

CATEGORIES

Select Category

Theory of Programming (http://theoryofprogramming.com/)

ARCHIVES
Home (http://theoryofprogramming.com/) Data Structures and Algorithms
AI (http://theoryofprogramming.com/category/artificial-intelligence/) Java (http://theoryofprogramming.com/category/java/)
Interview Corner C++ Team
Select Month

SUBSCRIBE TO BLOG VIA EMAIL

Enter your email address to subscribe to this blog and receive notifications of new posts by email.

Join 163 other subscribers

Email Address

SUBSCRIBE