|  |
| --- |
| Implementation of TRIE Digital Trees using JAVA |
| DATA STRUCTURES (CS - 201) FAST - NUCES |
| Khizer Mehmood (16k-3731)  Sec –B |

**IMPLEMENTATION OF TRIE DIGITAL TREES**

**ABSTRACT:**

In this vastly improving field of Computer Science, we need to search some information stored in data structure. There are multiple ways and method to search an integer or a string in our data. The searching algorithm has improved a lot in the past few decades. The searching algorithm can be implemented through array, linked list or a tree. These algorithms are evaluated on their run time complexity. However, we need a whole data structure that can store and maintain huge sets of data as well as perform these operations. In this report, an effort is made to compare the performance of TRIE digital tree using array and hash table. This study shows the time complexity of insertion, deletion and searching and how the runtime changes as the data size increases.

**I-INTRODUCTION**

The term ‘TRIE’ is derived from “retrieval”. Trie is a data structure storing a dynamic set or associative array where the keys are usually strings. It is one which allows strings or words having same prefixes to use the same data, and stores the ending of every string in a different node. Each node contains one character of a particular string and also a boolean type variable which keeps the record of whether the last character inserted is the the closure, or tail, of our string. Therefore, words are inserted in trie according to their prefixes.

There are different types of trie data structures. A standard trie for a set of string is an ordered tree such that:

* Each node is labeled with a character (without root)
* The children of a node are alphabetically ordered
* Path from the external node to the root yield the string

**STATEMENT OF THE PROBLEM**

The main issue is that there are a lot of algorithms, one more efficient than the other, when it comes to pattern search in a string. However, these algorithms do not work effectively or are very time consuming when it comes to large sets of data. For this purpose, a whole different data structure is assigned to manage such huge amounts of data effectively. Furthermore, the keys used all have the same length. So another problem that is resolved using this data structure is that the keys can be of varying length, and be different and the string can still be found out.

**AIMS AND OBJECTIVES OF THE PROJECT**

To resolve the issues stated above and to have a proper runtime analysis.

**SCOPE OF THE PROJECT**

This project consists of TRIE Digital Trees and its implementation using the language Java. The IDE used is Net-Beans.

**II-METHODOLOGY**

**Insertion**

Inserting a key into Trie is simple approach. Every character of input key is inserted as an individual Trie node. The children are array of pointers (or references) to next level trie nodes. The key character acts as an index into the array children. If the input key is new or an extension of existing key, we need to construct non-existing nodes of the key, and mark EndOfWord for last node. If the input key is prefix of existing key in Trie, we simply mark the last node of key as end of word. The key length determines Trie depth.

**Deletion**

During delete operation we delete the key in bottom up manner using recursion. The following are possible conditions when deleting key from trie:

1. Key may not be there in trie. Delete operation should not modify trie.
2. Key present as unique key (no part of key contains another key (prefix), nor the key itself is prefix of another key in trie). Delete all the nodes.
3. Key is prefix key of another long key in trie. Unmark the leaf node.
4. Key present in trie, having atleast one other key as prefix key. Delete nodes from end of key until first leaf node of longest prefix key.

**Searching**

When it comes to the data structure operations, the important idea behind trie digital trees is that searching can be done both word based and prefix based. To access the information stored from the nodes we simply follow the path moving down from the root, based on the characters in our key, via comparing. When we reach the appropriate word we check itsEndOfWord, if it is true then the word we were searching for does exist but if it is false then the word is not present in our trie’s data set.

**Pseudo Code for Functions working:**

1. public void insert(String Temp) {

Node current = root;

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

char ch = Temp.charAt(i);

Node node = current.Maping.get(ch);

if (node == null) {

node = new Node();

current.Maping.put(ch, node);

}

current = node;

}

//mark the current nodes Terminator as true

current.Terminator = true;

System.out.println("Word Inserted Successfully. ");

}

1. public boolean search(String Temp) {

Node current = root;

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

char ch = Temp.charAt(i);

Node node = current.Maping.get(ch);

//if node does not exist for given char then return false

if (node == null) {

System.out.println("Word Not Found.");

return false;

}

current = node;

}

//return true of current's Terminator is true else return false.

System.out.println("Word Found. ");

return current.Terminator;

}

3 ) public boolean delete(String Temp) {

return (delete(root, Temp, 0));

}

/\*

\* Returns true if parent should delete the mapping

\*/

private boolean delete(Node current, String Temp, int index) {

if (index == Temp.length()) {

//only delete if currrent.Terminator is true.

if (!current.Terminator) {

System.out.println("Prefix Found, Not Deleted.");

return false;

}

current.Terminator = false;

System.out.println("Word Deleted Successfully.");

//no other mapping then return true

return current.Maping.isEmpty();

}

char ch = Temp.charAt(index);

Node node = current.Maping.get(ch);

if (node == null) {

System.out.println("Word Does not exist.");

return false;

}

boolean CurrentNodeEmpty = delete(node, Temp, index + 1);

//if true is returned then delete the mapping of character and trienode reference from map.

if (CurrentNodeEmpty) {

current.Maping.remove(ch);

//return true if no mappings are left in the map.

return current.Maping.isEmpty();

}

return false;

}

**PERFORMANCE ANALYSIS**

**Insertion:**

|  |  |
| --- | --- |
| **No. of words** | **Runtime(seconds)** |
| 615 | 0.037 |
| 3611 | 0.107 |
| 4126 | 0.136 |
| 252716 | 25.97 |
| 255963 | 28.55 |

**Searching:**

|  |  |
| --- | --- |
| **No. of words** | **Runtime(seconds)** |
| 615 | 1.451e-5 |
| 3611 | 2.33e-5 |
| 4549 | 2.26e-5 |
| 252716 | 4.70e-5 |
| 255963 | 7.96e-5 |

**Deletion:**

|  |  |
| --- | --- |
| **No. of words** | **Runtime(seconds)** |
| 615 | 2.05e-5 |
| 3611 | 2.12e-5 |
| 4549 | 2.72e-5 |
| 252716 | 2.22e-5 |
| 255963 | 3.14e-5 |