Crafting a Dictionary and Phone Directory with Tries

November 5, 2023

Mudit Gupta (2022CSB1094) ,
Rakshit Kaushal (2022CSB1110) ,
Amitoj Singh (2022MCB1256)

Instructor:
Dr. Anil Shukla

Teaching Assistant:
Law Kumar

Summary: Our project focuses on the implementation and analysis of Tries data structure.It is a tree based data structure that is used for storing some collection of strings and performing efficient search operations on them.An important property this data structure follows is that if two strings have a common prefix then they have will have the same ancestor in the trie ,reducing time of the search operations.Apart from this other operations like insertion,deletion and auto completion of strings also make use of the same property to reduce the time complexities.In our project we have shown the use or application of this data structure in dictionaries and phone directories.The algorithms and pseudo-codes for various operations discussed above have been discussed in detail in coming subsections.

---

1. Introduction

In the subsequent sections,we will be implementing tries data structure which is used for storing,searching,updating and deleting collection of strings.This data structure is useful for preserving storage space as strings with common prefixes have common ancestors.

Why Use Tries? (1)

Tries[2] are extensively used for purposes like **Efficient Prefix Searches** and **Fast Autocomplete**.Other data structures like hash tables or binary search trees(BSTs) require linear scans or logarithmic searches,making tries faster for such operations.Unlike hash tables,tries do not suffer from hash collisions,which can degrade the performance of hash based data structures when multiple keys map to the same hash value.

IMPLEMENTATION (2)

In this project, we have implemented the following functions:

- **insert(string a)**:- Insert a string or a phone number to the current list of strings
- **search(string a)**:- Searches whether a particular string is currently present in the directory or not.
- **auto_complete(string a)**:- Given a string of characters or numbers , it provides the list of all such strings whose prefixes match with the given string i.e. it suggest words for incomplete strings
- **del(string a)**:- Deletes a string(if present) from the current list of words and numbers.

2. Analysis

This section provides the **time** and **space** complexities of various functions discussed in the code.

2.1. Functions:

2.1.1 insert(string a)

In this operation ,we prompt the user for a string input and insert the string into our main memory using memory optimization technique by storing the strings with common prefixes as children of common parents.
The Time Complexity of inserting a string into the tree is order of string length as at each level of the tree ,a single character of the string is stored and we need to traverse exactly n levels to insert the string.So we have $T(n)=\mathbf{1}+\mathbf{T}(n-1)$.
Hence $T(n)=\mathbf{O}(n)$;where n is the word length.

The space complexity analysis of the operation depends on the input strings i.e. larger the common part the string has to a previous inserted string the lesser is the number of new nodes required and the lesser is the space complexity.

2.1.2 search(string a)

In searching operation[1] , we will tell whether the given input string is present or not. From the first level we check whether the first character of the string is present on that level or not. If present , then we call the function again to the next level and now we pass the sub-string of length n-1.(The original string excluding the first character).Upon reaching the last character of the string, we check that isTerminal is true or not i.e. it is the end of string or not.If not present then return false i.e. the string is not present.
The Time Complexity of searching a string into the tree is order of string length as at each level of the tree ,a single character of the string is stored and we need to traverse exactly n levels to search the string.So we have $T(n)=\mathbf{1}+\mathbf{T}(n-1)$.
Hence $T(n)=\mathbf{O}(n)$;where n is the word length.

2.1.3 del(string a)

In this operation, we check for each character of the string in the stored tree. If at some point, we do not find a node corresponding to the required character, we simply return the function i.e. the string isn't even present. If we find all the characters of the string and the isTerminal value of the last node is TRUE then we conclude that the string is present and we start deleting. We change the isTerminal value to FALSE. If there are some children of the last node then we simply return the function.Else we delete the node and recursively perform the previous operation i.e. keep on deleting until we keep on getting nodes which have no child and also the isTerminal values of the nodes are FALSE.

The Time Complexity of deleting a string from the tree is order of string length as at each level of the tree ,a single character of the string is stored and we need to traverse exactly n levels to reach the end of the string and another n steps (recursive deletion) in the worst case to delete the string .So we have $T(n)=\mathbf{2}+\mathbf{T}(n-1)$.
Hence $T(n)=\mathbf{O}(n)$;where n is the word length.

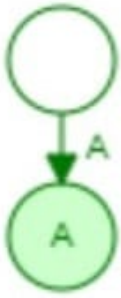
2.1.4 auto_complete(string s)

In this operation we first find the given prefix string in the tree and for each child of the last node of the prefix, we recursively call the same function with the appended string as input i.e. after appending the child character to the end of the string. If at some point during the execution of the function, we find that the isTerminal value of the string is TRUE, then we have found a string and we simply print the input string and continue the algorithm. We return the call only when we find that the node has no child.

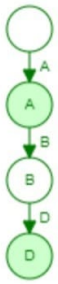
3. Figures, Tables and Algorithms

3.1. Figures

The following images show the pictorial implementation of various trie operations like inserting and deleting.[3]

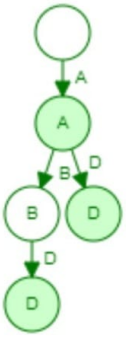


(a) INSERTING A

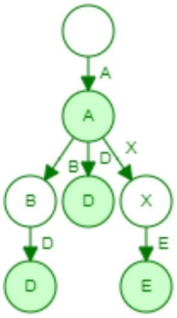


(b) INSERTING ABD

Figure 1: Inserting Operation

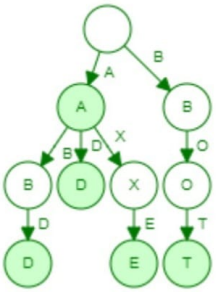


(a) INSERTING AD

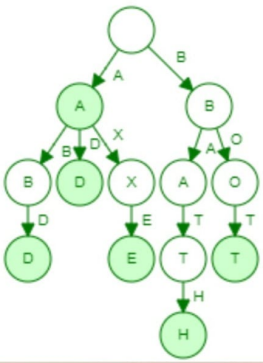


(b) INSERTING AXE

Figure 2: Inserting Operation.

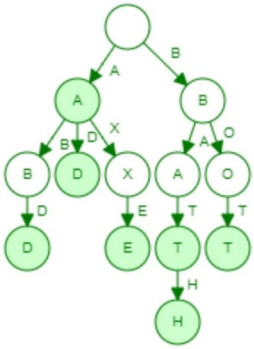


(a) INSERTING BOT

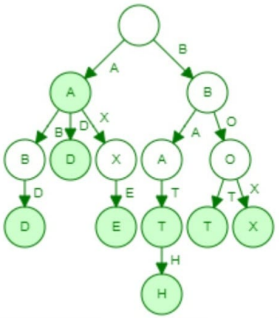


(b) INSERTING BATH

Figure 3: Inserting Operation.

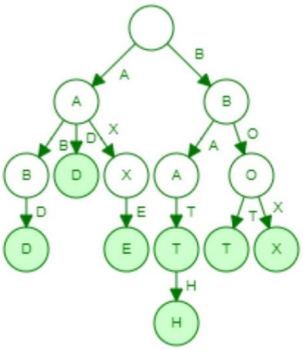


(a) INSERTING BAT

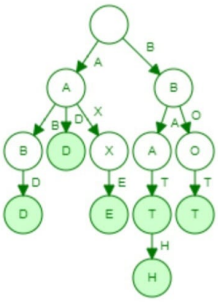


(b) INSERTING BOX

Figure 4: Inserting Operation.

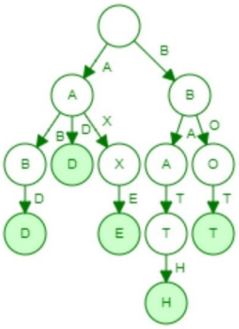


(a) DELETING A

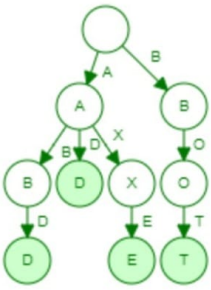


(b) DELETING BOX

Figure 5: Deletion Operation.

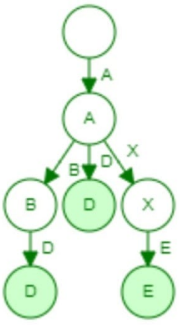


(a) DELETING BAT

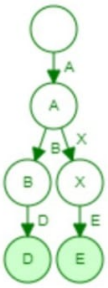


(b) DELETING BATH

Figure 6: Deletion Operation.



(a) DELETING BOT



(b) DELETING AD

Figure 7: Deletion Operation.

3.2. Tables

Below is the summary of the time complexities of various operations in our code given in Table 1.

	Worst Case Complexity
Search	O(word length)
Auto complete	O(word length)
Delete	O(word length)

Table 1: Worst case complexities of various operations.

3.3. Algorithms

Below are the pseudo-codes/Algorithms of various operations of our code[4].

Algorithm 1 Insert a string

if $string - a - is - empty$ **then**
  root->isTerminal=true
  return from the function
end if
i=child representing a[0]
if $root- > children[i] == NULL$ **then**
  root->children[i]=newNode with data=a[0]
  root->numberOfChildren++;
end if
insert(root->children[i],a.substr(1))

Algorithm 2 Search a string

```
if string − to − be − searched − is − over/empty then
    if root − is − an − ending − point − of − the − index then
        return true
    end if
end if
char c=letter corresponding to the first character of the string passed in the function
if the − child − correspong − to − charcter − c − is − NULL then
    return false(String not present)
end if
call the search function on the substring of the input string
```

Find the pointer of the node containing last letter of the string by using search node function and call auto complete on that node.Along with the node we send the string which has been detected in the tries till now.

Algorithm 3 Auto complete a string

```
if root − is − NULL then
    return from the function
end if
if root − points − to − the − ending − of − a − string(root− > isteminalistrue) then
    print the string traversed till now
end if
for i-from-0-to-number-of-children-of-root do
    auto complete(root->children[i],b)
    //string b corresponds to the input string + the character corresponding to the child i//
end for
```

Algorithm 4 Deleting a string

```
if string − to − be − searched − is − over/empty then
    if root − is − an − ending − point − of − the − index then
        root->isTerminal=false
    end if
    return
end if
Let i=child corresponding to the first letter of the string passed in the function
if i == NULL then
    return
end if
del(i,a.substr(1))
if root− > children[i]− > isTerminal == false then
    if root− > children[i]− > numberOfChildren == 0androot− > children[i]− >
    isTerminal == false then
        root->children[i]=NULL
        root->numberOfChildren=root->numberOfChildren-1
    end if
    root->children[i] = NULL
end if
```

4. Some further useful suggestions

Although tries solve various problems but there can also be many improvements which can be made depending on the use cases:

- **Compressed Tries**: A Compressed Trie, also known as a Patricia Trie (Practical Algorithm to Retrieve Information Coded in Alphanumeric), is a specialized variation of a trie data structure used for efficiently storing and searching strings or keys. The key feature of a compressed trie is the elimination of nodes with only one child to reduce storage space and improve search performance.

- **Hash Tries**: A hash trie is a hybrid data structure that combines elements of tries and hash tables to efficiently store and search for key-value pairs. Hash tries are often more memory-efficient than standard tries because they don't store explicit pointers for each character. They use hash tables, which can save memory, especially when dealing with a large number of keys.

5. Conclusions

Tries are specialized tree based data structures which use the concept of prefix tree to facilitate the operation of data retrieval, particularly in string based databases. The tries data structure offers several key advantages like fast insertions and deletions and space efficiency. On smaller data sets, it might not seem such a useful data structure but as data set grows, the advantages of using tries become evident. For ex: usually in a dictionary there will be a lot of words having common prefixes. Using tries we not only reduce the amortized search time for all such words but also conserve a lot of space.
Its ability to excel in tasks like autocomplete, dictionary search, and text processing, along with its space efficiency and versatility, makes it a compelling choice for a wide range of applications in computer science and software development

6. Bibliography and citations

Acknowledgements

We wish to thank our instructor, Dr Anil Shukla and Teaching Assistant Law Kumar for giving us their valuable time during the project.

References

[1] GeeksforGeeks. Trie - insert and search.

[2] JavaTpoint. Trie data structure.

[3] University of San Francisco. Trie visualization.

[4] Wikipedia. Trie.