Vietnam National University-Ho Chi Minh City

Ho Chi Minh University of Science

Faculty of Information Tecnology

Knowledge Engineering Department

Trie Data Structure Report

Report topic: Research on Trie Data Structure

Course: Data Structure and Algorithm

Student Group:
23127004 - Le Nhat Khoi
23127011 - Le Anh Duy
23127165 - Nguyen Hai Dang
23127189 - Tran Trong Hieu

Instructors:
Ms Nguyen Ngoc Thao
Mr Bui Huy Thong
Ms Tran Thi Thao Nhi

July 18, 2024

STT	Full Name	Job Description	Percentage
1	Le Nhat Khoi	Implement Trie's basic operations Research on "Prefix Search" Secretary	100%
2	Le Anh Duy	Project's problem solver Research on "Introduction" and "Insert operation" Leader	100%
3	Nguyen Hai Dang	Implement Trie's basic operations Project's problem solver Reseach on "Delete operation"	100%
4	Tran Trong Hieu	Project's problem solver Research on "Comparison Trie to other data structures" Writing down idea of Project's problem	100%

Contents

1 Theory part

1.1 Introduction to Trie Data Structure

Imagine you have a massive dictionary and need to find words quickly. Brute force searching (checking every word) would be slow and inefficient. Hashing can improve speed, but it doesn't handle operations like finding words with a specific prefix (e.g., all words starting with "ap").

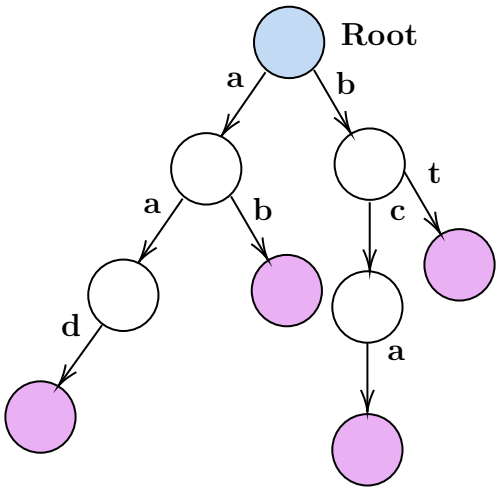
This is where **Tries** come in. A **Trie** is a specialized tree data structure designed for efficient storage and retrieval of strings. It excels at operations like: **Searching for words, Prefix search, Autocompletion, etc....**.

Formally, a trie is a tree structure, where:

- Each node represents a partial string or a complete word.
- Edges connecting nodes are labeled with characters.
- No two outgoing edges from a node have the same label.

By using a **trie**, you can leverage the power of a tree structure to organize words efficiently and perform searches based on prefixes, leading to faster retrieval and improved performance compared to brute force methods.

The diagram depicts a trie presentation for random words such as: "aad", "ab", etc. The blue node is the root node and the pink ones are the ending node of the Trie structure.



1.2 Trie Data Structure operation

Trie data structure can handle many operations effectively, some can be handled in linear or logarithm time complexity. In this section, we will examine an the following operation:

1.2.1 Adding a word

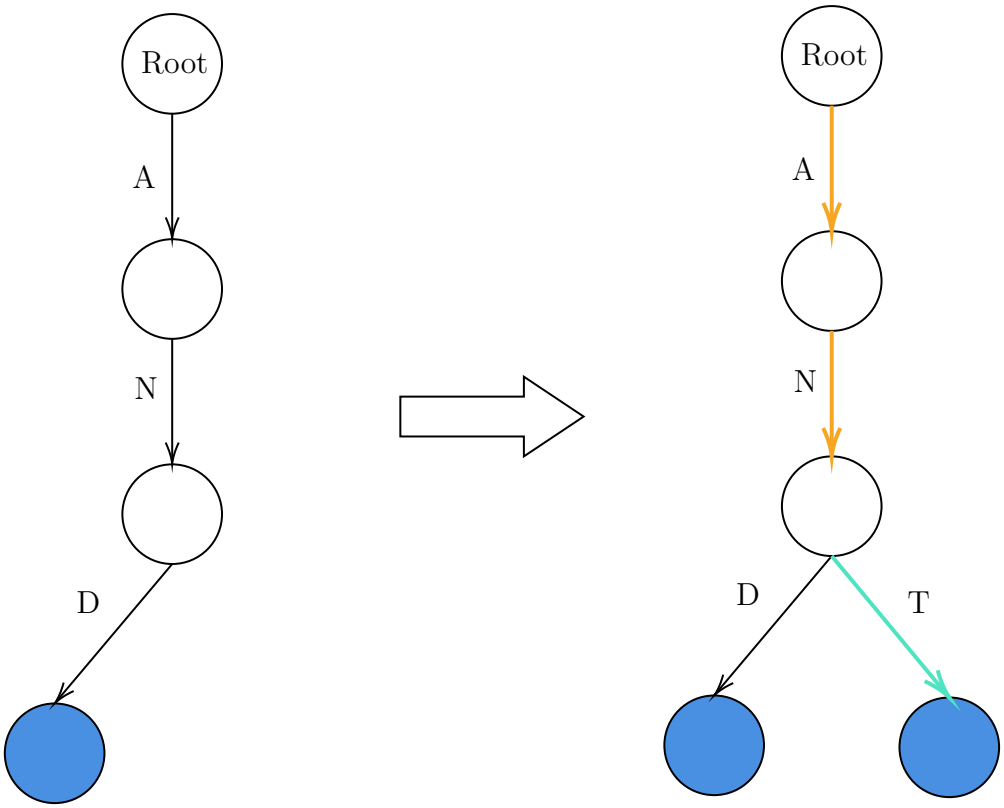
When a new word is added into the trie, the procedure happens as following:

Step 1: Starting from the root node of the tree structure

Step 2: Each character in the word is sequentially traversed and added into the trie.

Step 3: When the prefix is currently in the trie, we go along that link where the prefix could continue.

Step 4: When we encounter a character that make the prefix does not exist in the trie yet, we create new link to add the remaining suffix into the trie.



Here, from a trie that is storing the word "**AND**", we will insert the word "**ANT**" into it. We can see, for the prefix "**AN**" we follow the edges "**A**" and "**N**", which are colored in ograng for showing we are following those links, and a new link is added for inserting the "**T**", which is colored in blue.

Complexity analysis:

Time complexity: $O(n)$, because we only traverse along the word and create new node is $O(1)$. Therefor, by the time we finish traversing, we also finish adding.

Space complexity: $O(n)$, in the worst case, where the word has no common prefix with any of the inserted word.

1.2.2 Removing a Word

When deleting a word from a trie, the operation must be carefully performed to ensure that the structure and integrity of the trie are maintained. The delete operation is carried out in a bottom-up manner using recursion. The following conditions **must be considered**[2]:

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

Step-by-Step Procedure for Deleting a Word

Step 1: **Start at the Root Node:** Begin at the root node of the trie.

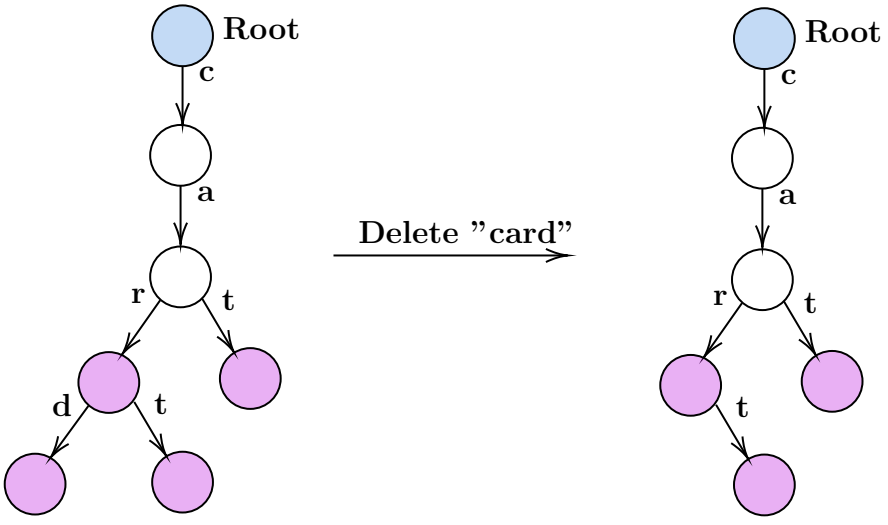
Step 2: **Traverse to the End of the Word:** Sequentially follow the links for each character in the word to reach the end of the word.

Step 3: **Check End of Word:** Once at the final character, check if it is marked as the end of a word. If not, the word does not exist in the trie, and the operation should be aborted.

Step 4: **Unmark the End of the Word:** If the final character is marked as the end of a word, unmark it.

Step 5: **Delete Nodes if Necessary:** Check if the node has any children. If it does, stop here as other words may depend on this node. If the node has no children, backtrack and delete nodes until you reach a node that has other children or is the end of another word.

Below is the visualization of deleting the word "card" from Trie Structure presents for "card", "cart", "cat":



Algorithm 1 Delete a Word from a Trie

```
function DELETE(word)
    return DELETEHELPER(root, word, 0)
end function

function DELETEHELPER(node, word, depth)
    if node is None then
        return False
    end if
    if depth == length(word) then
        if node.is_end_of_word == False then
            return False                                  ▷ Word not present in trie
        end if
        node.is_end_of_word = False
        return length(node.children) == 0                 ▷ If true, delete this node
    end if
    char = word[depth]
    if char not in node.children then
        return False                                      ▷ Word not present in trie
    end if
    should_delete_child = DELETEHELPER(node.children[char], word, depth + 1)
    if should_delete_child then
        delete node.children[char]
        return length(node.children) == 0 and node.is_end_of_word == False
    end if
    return False
end function
```

Time Complexity: The time complexity of the deletion operation is O(n) where n is the key length.

Auxiliary Space: O(n * m), where n is the key length of the longest word and m is the total no of words.

1.2.3 Searching a word

When we need to search for the existence of a word, the searching process traverse the same as when we insert. When we encounter a character that make the current pointer point to **NULL**, we know we do not have this word in our structure.

When we traverse to the end of the word, we need to check if the current node indicates any ending of a word or not, if it is, then the word exist, else the word do not exist in our trie.

Complexity analysis:

Time complexity: $O(n)$, because we only traverse along the existing node in the trie and do not create any new node. If the word exist, that mean we traverse through at most $n + 1$ node including the root node.

1.2.4 Searching words which has the same prefix with given length

In fact, Trie data structure is also called "Prefix Tree" since the root is connected to the initial part of strings that are inserted to the entire tree. Therefore, the path from the root to a node represents a prefix of some strings stored in the Trie. By following the path corresponding to a prefix, you can find all strings in the Trie that begin with that prefix.

Below is the procedure of searching words with the same prefix:

Step 1: Starting from the root node of the tree structure

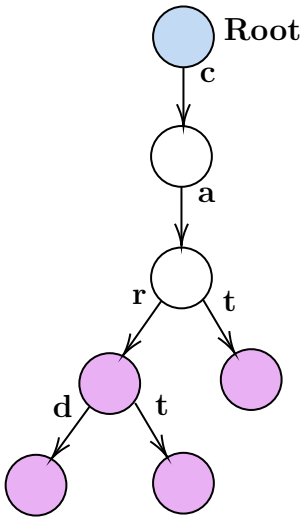
Step 2: Find and follow the path of the given prefix in the tree

Step 3: Print all the words in the subtree under the given prefix

Algorithm 2 Prefix Search

```
Current ← Root
for i ← 0 to Prefix Length - 1 do
    if Current.Child[Prefix[i]] = NULL then
        End the procedure
    else
        Current ← Current.Child[Prefix[i]]
    end if
end for
Print all the words in the subtree(root as Current) with given prefix
```

For visualization, in the Trie structure below, the words "card", "cart" and "cat" has the same prefix "ca":



Assume the p is the length of the prefix. The time complexity of reaching the node corresponding to the prefix is $O(p)$. In addition, assume n is the number of node in the subtree below the prefix, then the time complexity of printing all the words are $O(n)$

Therefore, the total time complexity takes to search for words with the same prefix is:

$$O(p) + O(n) = O(p + n)$$

1.3 Comparison

1.3.1 Trie & Binary Search Tree

Binary Search Tree idea: The main idea behind using binary search tree (BST) is to efficiently manage and retrieve data through binary decisions. In BST, each node has a key and pointers to left and right children, ensuring that left children have smaller values and right children have larger values. This structure allows for logarithmic time complexity for insertion, search, and deletion on average. BST is ideal for dynamic datasets and search and retrieval tasks, providing efficient data management through their binary structure.

Aspect	Trie	Binary Search Tree
Node Structure	Each node contains m links (pointers), m is the size of the character set.	Each slot contains an integer key
Space Efficiency	Efficient for storing words with common prefixes.	Can require more space due to multiple pointers.
Time Complexity	Insertion and search are $O(L)$, where L is the word length.	Insertion and search are $O(\log N)$ on average, $O(N)$ in the worst case.
Validation of Words	Easy, traverse from root to a node marked as the end of a word.	More cumbersome, involves comparing keys.
Prefix Matching	Efficiently finds all words with a given prefix.	Not efficient or intuitive for prefix searches.
Suitability for Task	Highly suitable, matches problem requirements well.	Less suitable, more complex for sequences of characters.

Table 1: Comparison of Trie and Binary Search Tree for the Given Task

Conclusion

Using a trie is more suited for this problem because it offers efficient storage and retrieval of words based on their prefixes. This is crucial for generating and validating words from a list of characters. A binary search tree, while useful for certain applications, doesn't provide the same level of efficiency and simplicity for this specific task.

1.3.2 Trie & Hash table

Hash table idea: To implement hash table for this problem, we need a hash function, that takes in a string and converts it into an integer key. In this section, the hash function will execute in $O(n)$ time, theoretically, where n is the length of the string.

Conclusion Hash table can handle vary of matching problems, whom require the exact search of a word. But hash is limited when dealing with fuzzy search, such as a word miss spelled at some position, ... In the problem we are required to complete, trie is more suitable, because it handing prefix matching more effectively.

Aspect	Trie	Binary Search Tree
Node Structure	Each node contains m links (pointers), m is the size of the character set.	Each slot contains a word, a left pointer and a right pointer
Space Efficiency	Efficient for storing words with common prefixes.	Efficient for storing distinct words
Time Complexity	Insertion and search are $O(n)$, where n is the word length.	Insertion and search are $O(logN)$ on average, $O(N)$ in the worst case, where N is the number of words inserted.
Collision	No collision guarantee	Chance of collision increases as the number of word are inserted increases
Exact match	Easy and fast	Easy and fast
Prefix Matching	Efficiently finds all words with a given prefix.	Do not support
Suitability for Task	Highly suitable, matches problem requirements well.	Not suitable, can't handle prefix search with hash table.

Table 2: Comparison of Trie and Binary Search Tree for the Given Task

2 Programming part

2.1 Preparation Part

2.1.1 Trie Construction

A trie data structure might be pre-built and stored externally. The trie efficiently represents a dictionary of valid words, where each node signifies a letter and branches represent possible following letters. A node marks the end of a valid word or not.

2.1.2 Input Acquisition

The input consists of two components:

- String A: This string contains all possible characters in sorted lowercase order (duplicates have been removed). It essentially represents the bag of tiles available for word formation.
- Integer array B: This array corresponds to the frequency of each character in input string. The value B[i] represents the number of tiles available for the character A[i].

2.2 Processing Part

We know that a path from the root of the trie to any children node **might** form a valid word. With this properties, using recursive **DFS** to traverse the trie is both efficient and simple to implement.

2.2.1 Recursive Function Call

The function `wordGenerateRecursion` is invoked with the following arguments:

- Pointer to the root node of the trie.
- String A containing all possible characters.
- Empty vector `res`: This vector will ultimately store all valid words found.
- Empty string `str`: This string serves as a temporary buffer to hold the current word being built during exploration.
- Integer array B: This array represents the remaining character frequencies.

2.2.2 Recursive Function (`wordGenerateRecursion`)

Base Case This condition checks two aspects:

- Is the current node representing the end of a word in the trie?
- Does the current string (`str`) have a minimum length of 3 characters?

If both conditions are met, it signifies a valid word has been constructed. The current string (`str`) is then appended to the results vector (`res`).

Recursive Case The function iterates through each character A[i] in the string A.
For each character A[i]:
This condition checks two crucial points:

- Are there still tiles of type A[i] available (using the frequency array B)?
- Does a valid path exist in the trie for the character A[i] (ensuring potential word formation)?

If both conditions are true (valid character and sufficient tiles), do the following operations:

- Decrement the count for character A[i] in the frequency array B, simulating the usage of one such tile in the current word.
- Append the character A[i] to the current string (`str`), representing its inclusion in the word being built.

A recursive call is made to `wordGenerateRecursion` with updated arguments:

- The child node corresponding to A[i] in the trie (to explore the path for this character).
- The remaining character string (A).
- The results vector (`res`).
- The updated string (`str`) with the additional character.
- The updated frequency array (B).

Backtracking After the recursive call finishes exploring this specific path:

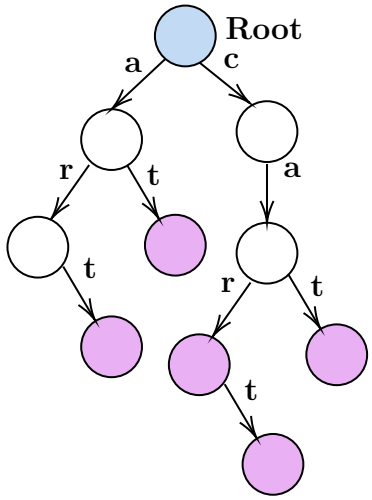
- .Backtracking occurs by removing the last character (A[i]) from the current string (`str`).
- The count for character A[i] in the frequency array B is incremented, undoing the previous usage.

2.3 Printing Part

Retrieving Results Upon completion of the recursive exploration, the results vector (res) will contain all valid words with a minimum length of 3 that could be formed using the available characters in the bag (A).

Output Generation The final step involves printing out the size of the results vector (res) iterating through it and printing each valid word found on a separate line. This presents the desired output of the word generation process.

Below is the visualization of the problem solution. Although the word "at" is in the Trie structure, the variable **count = 2** results in invalid word according to the problem requirements:



Input: a c r t
Output:
art
car
cart
cat

2.4 Optimization Made

Lexicographic order: The generated words are added to the results vector (res) in sorted ascending lexicographic order because, during the recursive function calls that traverse the trie structure, we follow the lexicographic order of the characters in the bag of tiles.

Avoid duplication: By removing duplicates from string A, you reduce the number of iterations during the recursive calls, as you only consider each unique character once. This optimization can significantly improve the performance, especially if the original string A contains many duplicate characters.

Backtracking: Using a string for backtracking simplifies the code by eliminating the need for additional data structures to store and track the characters selected for each word. It reduces memory consumption and provides a more streamlined implementation.

3 List of reference

References

[1] WikipediA, Trie

[2] GeeksforGeeks, Delete a word from a Trie

[3] Baeldung, Trie (Prefix Tree)