VNUHCM UNIVERSITY OF SCIENCE
Faculty of Information Technology

DATA STRUCTURE AND ALGORITHMS

RESEARCH PROJECT: TRIE

23CLC03

1653012 - Phạm Viết Minh Đạo
23127157 - Nguyễn Hoàng Gia Bảo
23127396 - Lương Linh Khôi
23127494 - Nguyễn Huỳnh Tiến

Ho Chi Minh city, 7/2024

Table of contents

1 Introduction

Trie Data Structure: An Overview:
A Trie is a tree-like data structure that stores a dynamic set of strings, typically to facilitate fast retrieval. Unlike binary search trees and hash tables, Tries leverage the common prefixes of string keys, making them particularly effective for scenarios where rapid search and retrieval of strings are critical. The time complexity for search operations in a Trie is proportional to the length of the search key, which is optimal for many applications.

Project Scope:
This project is divided into two main parts: theoretical analysis and practical implementation. In the theoretical part, we will explore the time complexity of various Trie operations, including adding, removing, and searching for words, as well as finding words with a common prefix. Additionally, we will compare Tries with binary search trees and hash tables, discussing their respective advantages and disadvantages in the context of search operations.

The practical part involves building a Trie from a given list of English words and implementing a program to generate valid English words from a set of provided characters. This exercise will not only reinforce the theoretical concepts but also demonstrate the practical utility of Tries in real-world applications.

2 Theory Part

2.1 Adding a word

Key operation:

- **Node Creation**: When a node does not exist for a character in the key, a new node is created and initialized.
- **Pointer Assignment**: The next pointers of the current node are assigned to the new node for the corresponding character.
- **Value Assignment**: The value for the key is assigned at the leaf node (or the node corresponding to the end of the key).

Time Complexity Analysis Given that:

- $|\Sigma|$ is the size of the alphabet
- n is the length of the key
- The `put` function is called recursively for each character in the key. Each recursive call processes one character and potentially creates a new node.
- For each character in the key (up to n characters), the following operations are performed:
  - **Node Creation (if needed)**: $O(|\Sigma|)$ to initialize an array of size $|\Sigma|$.
  - **Pointer Assignment**: $O(1)$.
  - **Value Assignment (at the end of the key)**: $O(1)$.

The overall time complexity for inserting a key into the trie is:

$$O(n) \times O(|\Sigma|) = O(n \cdot |\Sigma|)$$

2.2 Deleting a Word

Key Operations

- **Node Traversal**: Recursively traverse the trie to find the node corresponding to the end of the key.
- **Value Reset**: Set the value of the node to `NIL` to indicate the key has been deleted.
- **Cleanup**: Remove unnecessary nodes if they no longer contribute to any other keys in the trie.

Time Complexity Analysis Given that $|\Sigma|$ is the size of the alphabet and n is the length of the key:

- The `del` function is called recursively for each character in the key. Each recursive call processes one character.
- For each character in the key (up to n characters), the following operations are performed:
  - **Node Traversal**: $O(1)$ per character, leading to $O(n)$ recursive calls.
  - **Value Reset**: $O(1)$.
  - **Cleanup**: Checking for non-null children takes $O(|\Sigma|)$.
  - **Node Deletion**: $O(1)$.

The overall time complexity for deleting a key from the trie is:

$$O(n) \times O(|\Sigma|) = O(n \cdot |\Sigma|)$$

2.3 Searching a Word

Key Operations

- **Node Existence Check**: Checking if the current node is 'NULL'.
- **End of Key Check**: Checking if the current position in the key is at the end.
- **Recursive Call**: Recursively calling `get` for the next character in the key.

Time Complexity Analysis Given that n is the length of the key:

- The `get` function is called recursively for each character in the key.
- For each character in the key (up to n characters), the following operations are performed:
  - **Node Existence Check**: $O(1)$.
  - **End of Key Check**: $O(1)$.
  - **Recursive Call**: If the key has a length of n, there will be n recursive calls so the time complexity is $O(n)$.

The overall time complexity for searching a key in the trie is: $O(n)$

2.4 Searching Words with the Same Prefix of Length i

Key Operations

- **Traversing Nodes (`get` function)**: Search for the prefix.
- **Node Existence Check (`collect` function)**: Checking if the current node is `NULL`.
- **Value and Prefix Length Check (`collect` function)**: This check ensures that the current node has a non-`NIL` value and that the length of the prefix matches the desired length i.
- **Loop and Recursive Calls (`collect` function)**: This loop iterates through all possible children of the current node. The number of iterations is determined by the alphabet size ($|\Sigma|$). Each iteration involves making a recursive call to `collect`.

Time Complexity Analysis

Given that:

- T is the length of the prefix
- T' is the number of nodes in the sub-Trie starting from the prefix node
- **Traversing Nodes (`get` function)**: $O(T)$
- **Node Existence Check (`collect` function)**: $O(1)$
- **Value and Prefix Length Check (`collect` function)**: $O(1)$
- **Loop and Recursive Calls (`collect` function)**: The loop iterates $|\Sigma|$ times for each node. However, the recursive call is only made for non-`NULL` children, ensuring that each node is visited exactly once so the time complexity is $O(T' \cdot |\Sigma|)$

The overall time complexity for searching words which have the same prefix with length i in the trie is:

$$O(T) + O(T' \cdot |\Sigma|) = O(T + T' \cdot |\Sigma|)$$

2.5 Advantages

Compare to Binary Search Tree

A trie represents a sequence in its structure. It is very different in that it stores sequences of values rather than individual single values. Each level of recursion says, "what is the value of item i of the input list". This is different from a binary tree which compares the single searched value to each node.

- In tree structure, we store whole words. There is a lot of overhead. If you search for "antic", you would traverse each word and compare all the strings in "antic". This takes up too much time and memory.
- However, in tries, compression is the key. We store only the common prefixes. This will reduce the redundancy.

Pros of Tries

- The search time only depends on the length of the searched string.
  - The search time in a trie is $O(m)$, where m is the length of the searched string.
  - The search time in a BST can vary from $O(\log n)$ to $O(n)$, depending on the balance of the tree.
- Search misses only involve examining a few characters (in particular, just the longest common prefix between the search string and the corpus stored in the tree). Suppose you have a trie with words like "apple", "app", and "apricot" stored.
  - When searching for "applesauce":
    - You would compare "a" (root), "p" (child node), "p" (next child node), and then "l" (next child node).
    - At "l", you find no further nodes matching "e", so "apple" is the longest common prefix.
    - The search would confirm the existence of "apple" in the trie and stop at that point.
- There are no collisions of unique keys in a trie.
  - Each key (or string) is uniquely represented in the trie by its path from the root to a leaf node. The path consists of nodes, where each node corresponds to a character in the key.
- A trie can provide an alphabetical ordering of the entries by key.
  - Retrieving keys in alphabetical order from a trie is efficient because once the trie is constructed, traversing it to gather keys in order requires only linear time relative to the number of keys and their lengths ($O(n)$ where n is the total number of characters in all keys).

Compare to Hash Table

1. Faster Search Time

Trie: $O(m)$ where m is the length of the search key

Hash Table: $O(1)$ on average, but can be $O(n)$ in the worst case due to collisions

In a Trie, each node represents a character in the search key, and the search time is proportional to the length of the key. In a Hash Table, the search time is constant on average, but can be linear in the worst case if there are many collisions.

Example: Suppose we have a Trie and a Hash Table containing a large number of strings. We want to search for the string "banana". In the Trie, we start at the root node and traverse down to the node corresponding to the first character 'b', then to the node corresponding to the second character 'a', and so on. The search time is proportional to the length of the string, which is 6 in this case. In the Hash Table, we compute the hash value of the string "banana" and look up the corresponding bucket. If there are many collisions, we may have to iterate through a large number of strings in the bucket to find the desired string, leading to a longer search time.

2. Space Efficiency

Trie: Can store a large number of strings in a relatively small amount of space

Hash Table: Requires a large amount of space to store a large number of strings

In a Trie, each node only stores a small amount of information (e.g. a character and a pointer to the next node), so the total space required is relatively small. In a Hash Table, each entry requires a significant amount of space to store the key and value, so the total space required can be large.

Example: Suppose we have a Trie and a Hash Table containing 1 million strings, each with an average length of 10 characters. The Trie may require around 10-20 MB of space, depending on the implementation. The Hash Table, on the other hand, may require around 100-200 MB of space, depending on the implementation and the size of the keys and values.

3. Prefix Matching

Trie: Supports prefix matching, where we can find all strings that start with a given prefix

Hash Table: Does not support prefix matching

In a Trie, we can start at the root node and traverse down to the node corresponding to the last character of the prefix. From there, we can traverse down to all nodes that correspond to strings that start with the given prefix. In a Hash Table, we can only look up exact matches, not prefixes.

Example: Suppose we have a Trie containing a large number of strings, and we want to find all strings that start with the prefix "ban". We start at the root node and traverse down to the node corresponding to the last character of the prefix, which is 'n'. From there, we can traverse down to all nodes that correspond to strings that start with the prefix "ban", such as "banana", "band", etc.

4. Autocomplete

Trie: Supports autocomplete, where we can find all strings that start with a given prefix and have a certain length.

Hash Table: Does not support autocomplete

In a Trie, we can start at the root node and traverse down to the node corresponding to the last character of the prefix. From there, we can traverse down to all nodes that correspond to strings that start with the given prefix and have a certain length. In a Hash Table, we can only look up exact matches, not prefixes or lengths.

Example: Suppose we have a Trie containing a large number of strings, and we want to find all strings that start with the prefix "ban" and have a length of 6. We start at the root node and traverse down to the node corresponding to the last character of the prefix, which is 'N'. From there, we can traverse down to all nodes that correspond to strings that start with the prefix "ban" and have a length of 6, such as "banana".

3 Programming Part

3.1 Implement and Build a Trie (with Words from the Given Files)

***First Step*: Declaring Structure for the Trie and TrieNode**

```
Structure TrieNode:
    Character data
    Boolean wordEnd
    Array of TrieNode pointers children[26]
```

The TrieNode structure contains a character `data`, a boolean `wordEnd` to indicate the end of a word, and an array of pointers `children[26]` representing the 26 possible children for each node, corresponding to the 26 letters of the alphabet.

```
Structure Trie:
    Pointer to TrieNode root
```

The Trie structure contains a pointer which points to the root node of the Trie.
***Second Step:* Adding Necessary Functions to the Trie and TrieNode Structure**

- Constructor of TrieNode

```
Function TrieNode():
    Initialize data
    Initialize wordEnd as False
    Initialize each element in children to None
```

- Constructor of Trie

```
Function Trie():
    Initialize root as a new TrieNode
```

- Function `insertKey(key)`: This Function Inserts a Given Word into the Trie Structure

```
Function insertKey(key):
    Pointer to TrieNode cur = root
    For i from 0 to size of key - 1:
        Index = key[i] - 'a'
        If cur.children[Index] is null:
            Pointer to TrieNode newNode = new TrieNode
```

```
                Set cur.children[Index] to newNode
            Set cur to cur.children[Index]
        Set cur.wordEnd to true
```

Description: This function starts at the root node and iterates through each character of the input string `key`. For each character, it calculates the corresponding index in the `children` array by subtracting the ASCII value of 'a' from the character. If the child node at that index is `null`, it creates a new TrieNode and assigns it to that position. The function then moves the `cur` pointer to the child node. After processing all characters in the `key`, it marks the end of the word by setting the `wordEnd` flag of the current node to `true`, implying that a valid word is stored in the Trie.

Third Step: **Adding Necessary Functions to Read Words from Given File**

- Function `initTrie(fileName)`:

```
Function initTrie(fileName):
    Open file with name fileName as fIn
    While fIn can read word:
        Call insertKey(word)
    Close file fIn
```

- Description: This function, which is a part of the Trie structure, is used to generate a Trie from a given file containing a list of words.

Fourth Step: **Calling `initTrie` Function in the `main.cpp` File**

3.2 Implement a Program to Generate a List of Valid English Words Which Have Letters from a Given Character List

First Step: **Read a list of character from the console**

```
Function input():
    Declare listOfChars as empty list of characters
    Declare input as empty string

    Read a line of input from standard input into input

    For each character ch in input:
        If ch is an alphabetical character:
            Append ch to listOfChars
    Return listOfChars
```

- Description: This function is an ultility function that will read a list of characters from the console and return them as vector.

Second Step: **Generate Words Using the Letters from the Given List of Characters**

This step includes two parts, corresponding to two functions respectively:

- Part 1: Populate the `charCount` Array, Which Stores the Number of Occurrences of Each Character in the List of Characters

```
Function generateWords(trie, listOfChars, listOfWords):
    Initialize charCount array of size 26 to all zeros

    For each character in listOfChars:
        Increment charCount at index (character - 'a')

    Call backtrack with arguments (trie.root, empty string,
    charCount, listOfChars, listOfWords)
```

Description: This function will accept a Trie object and a list of characters as vector. It iterates through the listOfChars vector and increments the corresponding index in the charCount array for each character. The index is calculated by subtracting the ASCII value of 'a' from the character itself, ensuring that 'a' maps to index 0, 'b' to index 1, and so on. After populating the charCount array, the function calls the backtrack function

- Part 2: Recursively Find All Possible Words from the Trie Using Given Characters

```
Function backtrack(node, path, charCount, listOfChars, listOfWords):
    If length of path is greater than or equal to 3 AND node.wordEnd
    is true:
        If path is not in listOfWords:
            Append path to listOfWords

    For i from 0 to 25:
        If charCount[i] > 0 AND node.children[i] is not null:
            Decrement charCount[i]
            Call backtrack(node.children[i], path +
            character(i + 'a'), charCount,
            listOfChars, listOfWords)
            Increment charCount[i]
```

Description: The backtrack function recursively generates words from a Trie using a specified character count. It takes a TrieNode, a current word path, an array of character counts, and a pass-by-reference list to store generated words. If the current path is at least 3 characters long and a valid word in the Trie, it adds the word to the generated word list if it's not already exist. The function then iterates through possible characters, decrementing their counts, exploring child nodes in the Trie, and recursively calling itself to build longer words. After each recursion, it restores the character count for further exploration.

Third Step: **Write the list of generated word into the console**

```
Function output(listOfWords):
    Print size of listOfWords

    For each word in listOfWords:
        Print word
```

- Description: This function will write to the console a list of generated words in the following format: The first line is the number of generated words, the subsequent lines are the generated words, each line contains only one word.

Fourth Step: **Call** `input`**,** `generateWords`**, and** `output` **Functions in** `main.cpp` **Together with Appropriate Arguments**

3.3 Side Notes

We also implement `deleteNodes` function to delete all nodes in the Trie to prevent memory leaks. This function will be called in the destructor of Trie.

```
Function deleteNodes(reference to TrieNode root):
    If root is not null:
        For each index i from 0 to 25:
            Call deleteNodes(reference to root.children[i])
        Delete root
    Set root to null
```

4 Project Organization and Programming Notes

Project Organization

There are a total of three `.cpp` files and two `.h` header files in the submission, each of them having a distinguished functionality:

- `main.cpp`: The main entry file, used to run the program by executing all necessary functions.
- `Trie.cpp` and `Trie.h`: Contain the structure of the Trie and functions that perform various operations on the Trie.
- `Utility.cpp` and `Utility.h`: Contain functions for reading and writing files and generating words.

To build an executable file with `g++`:

1. Open Command Prompt and locate the folder containing all those `.cpp` and `.h` files.
2. Type the following: `g++ main.cpp Trie.cpp Utility.cpp -o trie.exe` to begin building the executable file.
3. An executable file should be created. Then, in Command Prompt, run that file together with an appropriate command line.

Programming Notes

Some special libraries were used to make the program work precisely:

- The `<algorithm>` library (used in `Utility.cpp`) was used to search through a list of generated words to check if the current word already exists, thereby avoiding duplication.
- The `<vector>` library (used in `Utility.cpp`) was used to create a container for storing all characters when reading from a file and for storing all generated words when writing to the output file.

Other Notes

- The `backtrack` function (used in `Utility.cpp`) is a backtracking function that generates all words from the Trie based on the given list of characters.

5 References

- Advantages of Trie over BST, GeekforGeeks, Available at: `https://www.geeksforgeeks.org/difference-between-binary-tree-and-binary-search-tree/`
- Advantages of Trie over BST, Stackoverflow, Available at: `https://stackoverflow.com/questions/4737904/difference-between-tries-and-trees`
- Advantages of Trie over Hash table, GeekforGeeks, Available at: `https://www.geeksforgeeks.org/hash-table-vs-trie/`