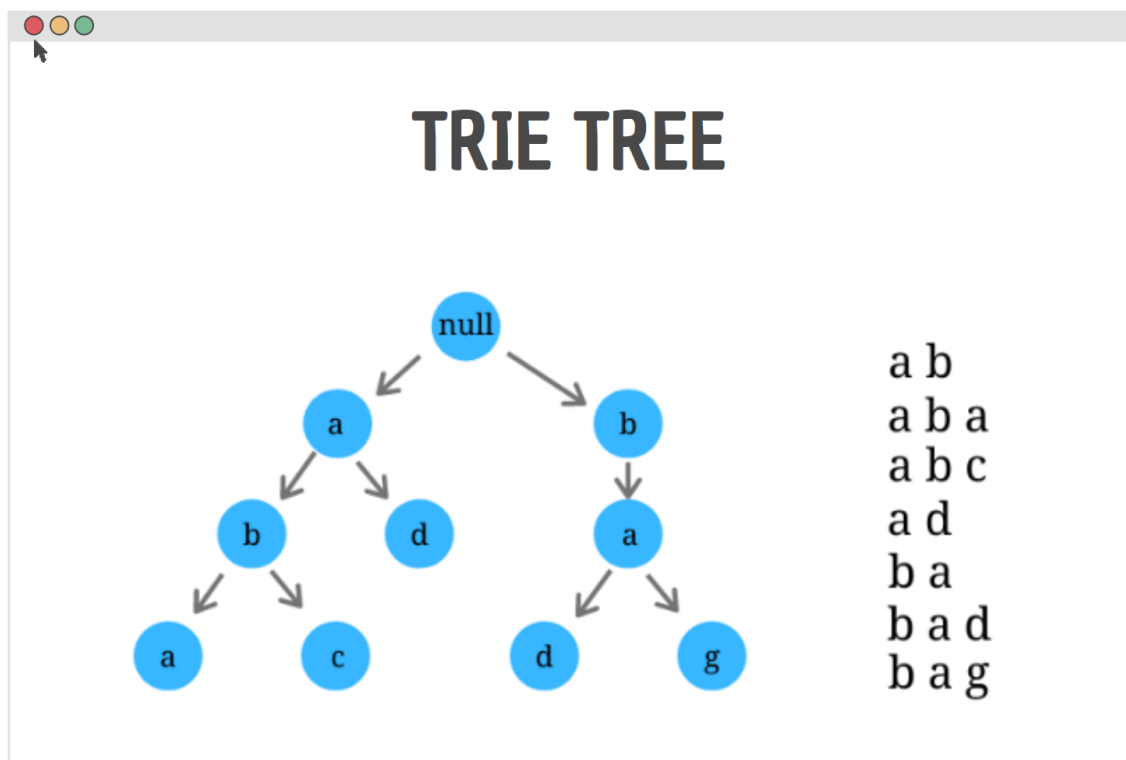


ESTRUCTURAS DE DATOS Y ALGORITMOS con Java



Profesor(a):
Edson Luque Mamani

Estudiantes:
Jorge Luis Mamani Huarsaya
Victor Narciso Mamani Anahua

27 de junio, 2024

Implementación de un Trie Tree en Java

Un Trie es una estructura de datos especializada que se utiliza para almacenar un conjunto de cadenas, generalmente para realizar búsquedas rápidas de palabras. Se detallará el funcionamiento de cada método y la importancia de los campos establecidos en la clase.

Estructura de la Clase

A continuación se presenta la implementación de la clase `Trie`:

```
1 import java.util.HashMap;
2 import java.util.Map;
3
4 public class Trie implements ITrie {
5     private TrieNode root;
6
7     public Trie() {
8         // El nodo raíz que representa el caracter nulo
9         root = new TrieNode('\0');
10    }
11
12    private static class TrieNode {
13        int count = 0;
14        char ch;
15        Map<Character, TrieNode> children;
16        boolean isEndOfWord;
17
18        public TrieNode(char ch) {
19            children = new HashMap<>();
20            isEndOfWord = false;
21            this.ch = ch;
22        }
23    }
24 }
```

Campos de la Clase

- **root**: Es la raíz del Trie, un nodo especial que no representa ningún carácter ('`\0`').
- **TrieNode**: Clase estática interna que representa un nodo en el Trie.
 - **count**: Cuenta las veces que una palabra termina en este nodo.
 - **ch**: Caracter que representa el nodo.
 - **children**: Mapa de hijos, donde la clave es un carácter y el valor es otro nodo `TrieNode`.
 - **isEndOfWord**: Indica si este nodo es el final de una palabra.

Métodos de la Clase

La clase `Trie` implementa los métodos definidos en la interfaz `ITrie`. A continuación se detallan cada uno de estos métodos.

Inserción de Palabras

```
1 @Override
2 public void insert(String word) {
3     TrieNode current = root;
4     for (char ch : word.toCharArray())
5         current = current.children.computeIfAbsent(ch, c -> new TrieNode(c));
6
7     current.isEndOfWord = true;
8     current.count++;
9 }
```

Este método inserta una palabra en el Trie. Recorre cada carácter de la palabra y lo inserta en el Trie si no está presente. Utiliza el método `computeIfAbsent` de `HashMap` para agregar un nuevo nodo si el carácter no existe en los hijos actuales. Al final de la palabra, marca el nodo como el final de una palabra (`isEndOfWord`) y aumenta el contador (`count`).

Búsqueda de Palabras

```
1 @Override
2 public boolean contains(String word) {
3     TrieNode current = root;
4     for (char ch : word.toCharArray()) {
5         current = current.children.get(ch);
6         if (current == null)
7             return false;
8     }
9     return current.isEndOfWord;
10 }
```

Este método verifica si una palabra está contenida en el Trie. Recorre cada carácter de la palabra y verifica si existe en el Trie utilizando el método `get` de `HashMap`. Si al final de la palabra, el nodo actual es el final de una palabra, retorna `true`, de lo contrario, `false`.

Obtención de Palabras

```
1 @Override
2 public String get(String word) {
3     TrieNode current = root;
4     for (char ch : word.toCharArray()) {
5         current = current.children.get(ch);
```

```
6     if (current == null)
7         return null;
8     }
9     return current.isEndOfWord ? word : null;
10 }
```

Este método retorna la palabra si está contenida en el Trie. Sigue la misma lógica que el método `contains`, pero retorna la palabra en lugar de un valor booleano.

Eliminación de Palabras

```
1  @Override
2  public boolean remove(String word) {
3      return delete(root, word, 0);
4  }
5
6  private boolean delete(TrieNode current, String word, int index) {
7      if (index == word.length()) {
8          if (!current.isEndOfWord)
9              return false;
10
11         current.isEndOfWord = false;
12         return current.children.isEmpty();
13     }
14
15     char ch = word.charAt(index);
16     TrieNode node = current.children.get(ch);
17     if (node == null)
18         return false;
19
20     boolean shouldDeleteCurrentNode = delete(node, word, index + 1);
21     if (shouldDeleteCurrentNode) {
22         current.children.remove(ch);
23         return current.children.isEmpty();
24     }
25     return false;
26 }
```

El método `remove` elimina una palabra del Trie. Utiliza un método recursivo `delete` que elimina los nodos si ya no son necesarios.

El Método `delete`

El método `delete` es el núcleo del proceso de eliminación. A continuación se explica paso a paso su funcionamiento:

- **Caso Base:** Si el índice ha alcanzado la longitud de la palabra (`index == word.length()`), se verifica si el nodo actual marca el final de una palabra (`current.isEndOfWord`). Si no

es así, significa que la palabra no está en el Trie y se retorna **false**. Si lo es, se desmarca como el final de una palabra y se retorna **true** si el nodo no tiene hijos, indicando que este nodo puede ser eliminado.

- **Recursión:** Se obtiene el siguiente carácter de la palabra y el nodo hijo correspondiente. Si el nodo hijo no existe, se retorna **false**, indicando que la palabra no está en el Trie.
- **Eliminación Condicional:** Se llama recursivamente al método **delete** con el nodo hijo, la palabra y el índice incrementado. Si esta llamada retorna **true**, significa que el nodo hijo puede ser eliminado. Por lo tanto, se elimina el nodo hijo del mapa de hijos del nodo actual y se retorna **true** si el nodo actual no tiene más hijos.

Ranking de frecuencia

```
1 public List<String> findTopKFrequentWords(int k) {
2     PriorityQueue<Map.Entry<String, Integer>> minHeap = new
↪ PriorityQueue<>((a, b) -> a.getValue() - b.getValue());
3     Map<String, Integer> wordFrequencyMap = new HashMap<>();
4     findTopKFrequentWordsHelper(root, "", wordFrequencyMap);
5     for (Map.Entry<String, Integer> entry : wordFrequencyMap.entrySet()) {
6         minHeap.offer(entry);
7         if (minHeap.size() > k) {
8             minHeap.poll();
9         }
10    }
11    List<String> result = new ArrayList<>();
12    while (!minHeap.isEmpty()) {
13        result.add(minHeap.poll().getKey());
14    }
15    Collections.reverse(result);
16    return result;
17 }
18 private void findTopKFrequentWordsHelper(TrieNode node, String word,
↪ Map<String, Integer> wordFrequencyMap) {
19     if (node == null) {
20         return;
21     }
22     if (node.isEndOfWord) {
23         wordFrequencyMap.put(word, node.count);
24     }
25     for (Map.Entry<Character, TrieNode> entry : node.children.entrySet()) {
26         findTopKFrequentWordsHelper(entry.getValue(), word + entry.getKey(),
↪ wordFrequencyMap);
27     }
28 }
```

El Método findTopKFrequentWords

- **Descripción General** : La función `findTopKFrequentWords` toma un parámetro entero `k` y devuelve una lista de las `k` palabras más frecuentes en el trie. Para lograr esto, utiliza una cola de prioridad (min-heap) para mantener un seguimiento de las palabras más frecuentes.
- **HashMap de Frecuencias** : Primero, crea un `HashMap` llamado `wordFrequencyMap` para almacenar la frecuencia de cada palabra. Luego, llama a la función auxiliar `findTopKFrequentWordsHelper` para llenar este mapa.
- **Llenado de la Cola de Prioridad** : Después de llenar el mapa de frecuencias, la función itera sobre las entradas del mapa e inserta cada entrada en la cola de prioridad. Si el tamaño de la cola excede `k`, se elimina la entrada con la frecuencia más baja. Esto asegura que la cola de prioridad contenga sólo las `k` entradas más frecuentes.
- **Generación del Resultado** : Finalmente, la función extrae las entradas de la cola de prioridad, las agrega a una lista, y revierte el orden de la lista antes de devolverla, ya que la cola de prioridad saca las palabras en orden ascendente de frecuencia.

Reemplazar palabras en un texto

```
1 public String replaceWordsInText(String text, Map<String, String>
2   replacements) {
3     StringBuilder result = new StringBuilder();
4     String[] words = text.split("\\s+");
5     for (String word : words) {
6       if (contains(word)) {
7         result.append(replacements.getOrDefault(word, word)).append(" ");
8       } else {
9         result.append(word).append(" ");
10      }
11    }
12    return result.toString().trim();
13 }
```

El Método replaceWordsInText

- **Descripción General** : La función `replaceWordsInText` toma un texto y un mapa de reemplazos, y devuelve el texto con ciertas palabras reemplazadas según el mapa.
- **Inicialización del Resultado** : Se utiliza un `StringBuilder` llamado `result` para construir el texto resultante de manera eficiente. El texto se divide en palabras utilizando el método `split` con el delimitador de espacios en blanco ("`s+`").
- **Iteración y Reemplazo** : Se itera sobre cada palabra en el arreglo `words`. Para cada palabra, se verifica si debe ser reemplazada. Si la palabra está en el mapa de reemplazos, se añade el reemplazo al `StringBuilder`. Si no, se añade la palabra original.
- **Devolución del Resultado** : Finalmente, se convierte el `StringBuilder` a una cadena, se eliminan los espacios al principio y al final, y se devuelve el resultado.

Busqueda de palabras por un prefijo

```
1 private void collectWords(TrieNode node, String prefix, List<String> words)
2 {
3     if (node.isEndOfWord) {
4         words.add(prefix);
5     }
6     for (Map.Entry<Character, TrieNode> entry : node.children.entrySet()) {
7         collectWords(entry.getValue(), prefix + entry.getKey(), words);
8     }
9 }
10 private TrieNode searchNode(String prefix) {
11     TrieNode node = root;
12     for (char c : prefix.toCharArray()) {
13         node = node.children.get(c);
14         if (node == null) {
15             return null;
16         }
17     }
18     return node;
19 }
20 public List<String> searchByPrefix(String prefix) {
21     List<String> words = new ArrayList<>();
22     TrieNode node = searchNode(prefix);
23     if (node != null) {
24         collectWords(node, prefix, words);
25     }
26     return words;
27 }
```

El Método searchByPrefix

- **Descripción General** : La función `searchByPrefix` busca todas las palabras en el trie que comienzan con un prefijo dado. Primero, utiliza `searchNode` para encontrar el nodo del trie correspondiente al final del prefijo. Luego, utiliza `collectWords` para recoger todas las palabras que tienen el prefijo especificado.
- **Búsqueda del Nodo** : La función `searchNode` recibe un prefijo y recorre el trie siguiendo los caracteres del prefijo. Si en algún momento el nodo no existe, la función retorna `null`. De lo contrario, retorna el nodo donde termina el prefijo.
- **Recogida de Palabras** : La función `collectWords` es recursiva y recolecta todas las palabras que se encuentran bajo un nodo dado en el trie. Si el nodo actual marca el final de una palabra (`isEndOfWord` es `true`), añade el prefijo correspondiente a la lista de palabras.
- **Ejecución y Resultado** : En `searchByPrefix`, si el nodo del prefijo no es `null`, se llama a `collectWords` para obtener todas las palabras bajo ese nodo. Finalmente, la función devuelve la lista de palabras encontradas.

Impresión del Trie

```
1 public void printTrie() {
2     printTrie(root, "", true);
3 }
4
5 private void printTrie(TrieNode node, String prefix, boolean isTail) {
6     if (node != null) {
7         System.out.println(prefix + (isTail ? " " : " ") +
8                               (node.ch != '\\0' ? node.ch : "") +
9                               (node.isEndOfWord ? "(" + node.count + ")" : ""));
10
11         int children = node.children.size();
12         int i = 0;
13         for (Map.Entry<Character, TrieNode> entry : node.children.entrySet()) {
14             printTrie(entry.getValue(), prefix + (isTail ? " " : " "), ++i ==
15 ↪ children);
16         }
17     }
18 }
```

Este método imprime el Trie de manera visual utilizando caracteres especiales. Muestra la jerarquía de los nodos y la cuenta de palabras que terminan en cada nodo.

El Método printTrie

El método `printTrie` es una representación visual del Trie. Utiliza caracteres como y `()` para mostrar la estructura jerárquica de los nodos.

- **Lógica de Impresión:** El método recursivo `printTrie` toma un nodo, un prefijo y un booleano que indica si el nodo es la cola (el último hijo) de su padre. Imprime el prefijo seguido del carácter del nodo y, si es el final de una palabra, el contador de esa palabra.
- **Recorrido Recursivo:** Recorre recursivamente todos los hijos del nodo actual, ajustando el prefijo según si el nodo actual es la cola o no, y llamándose a sí mismo para cada hijo.

Prueba del programa

Para poder probar el programa necesitamos usar algún emulador de terminal, en este caso vamos a utilizar **Wezterm**.

```
1 # Compilar el programa
2 javac TestTree.java
3
4 # Ejecutar el programa
5 java TestTree
```



```
user@victus:~/trie-tree/Trie_Tree$ java TestTrie
Bienvenido, construye tu propio arbol digital :)
```

```
-----
insert      ->    Insert a word
remove      ->    Remove a word
contains    ->    Verify if contains a word
show        ->    Show the trie tree
prefix      ->    Search words by prefix
replace     ->    Replace words in text
topk        ->    Find top k frequent words
quit        ->    Exit program
-----
```

```
trie[string]> insert victor
```

```
trie[string]> insert jorge
```

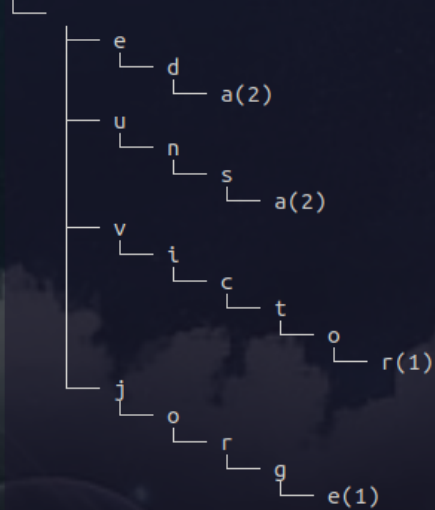
```
trie[string]> insert unsa
```

```
trie[string]> insert eda
```

```
trie[string]> insert unsa
```

```
trie[string]> insert eda
```

```
trie[string]> show
```



```
trie[string]> insert juan
```

```
trie[string]> insert educacion
```

Figura 1: Trie Tree



Figura 2: Trie Tree

```
trie[string]> show
├── e
│   └── d
│       ├── a(2)
│       └── u
│           ├── c
│           │   ├── a
│           │   │   ├── c
│           │   │   │   ├── i
│           │   │   │   │   ├── o
│           │   │   │   │   │   └── n(1)
│           └── c
│               ├── a
│               │   ├── c
│               │   │   ├── i
│               │   │   │   ├── o
│               │   │   │   │   └── n(1)
│               └── u
│                   ├── n
│                   │   ├── s
│                   │   │   └── a(2)
│                   └── j
│                       ├── u
│                       │   ├── a
│                       │   │   └── n(1)
│                       └── o
│                           ├── r
│                           │   ├── g
│                           │   │   └── e(1)
└── u
    ├── n
    │   ├── s
    │   │   └── a(2)
    └── j
        ├── u
        │   ├── a
        │   │   └── n(1)
        └── o
            ├── r
            │   ├── g
            │   │   └── e(1)

trie[string]> prefix j
Words with prefix "j": [juan, jorge]

trie[string]> topk 2
Top 2 frequent words: [unsa, eda]

trie[string]> replace unsa casa jua diego
Enter text:
Ayer jua salio de la unsa a las 9 de la noche
Replaced text: Ayer diego salio de la casa a las 9 de la noche

trie[string]> quit
user@victus:~/trie-tree/Trie_Tree$
```

Figura 3: Trie Tree