GUÍA DE APRENDIZAJE PARA CONCURSANTES ICPC Y IOI: ÁRBOL DE RECUPERACIÓN DE INFORMACIÓN (*TRIE*)

1. Introducción

Dentro de las diferentes estructuras de datos que puedan existir existen algunas que trabajan con determinados tipos de datos especificos. Tal es el caso del árbol de recuperación de información (*Trie*) una estructura de datos de tipo árbol que permite la recuperación de información de un grupo de secuencias de caracteres. Sobre dicha estructura de dato aboradará la siguiente guía.

2. Conocimientos previos

2.1. Árbol

En ciencias de la computación y en informática, un árbol es un tipo abstracto de datos (TAD) ampliamente usado que imita la estructura jerárquica de un árbol, con un valor en la raíz y subárboles con un nodo padre, representado como un conjunto de nodos enlazados.

Una estructura de datos de árbol se puede definir de forma recursiva (localmente) como una colección de nodos (a partir de un nodo raíz), donde cada nodo es una estructura de datos con un valor, junto con una lista de referencias a los nodos (los hijos), con la condición de que ninguna referencia esté duplicada ni que ningún nodo apunte a la raíz.

2.2. Árbol de búsqueda binaria

Un árbol binario de búsqueda también llamado BST (acrónimo del inglés *Binary Search Tree*) es un tipo particular de árbol binario que presenta una estructura de datos en forma de árbol usada en informática. Es un árbol binario que cumple que el subárbol izquierdo de cualquier nodo (si no está vacío) contiene valores menores que el que contiene dicho nodo, y el subárbol derecho (si no está vacío) contiene valores mayores.

3. Desarrollo

Introducidos en 1959 independientemente por Rene de la Briandais y Edward Fredkin un*Trie* es una estructura de datos de tipo árbol que permite la recuperación de información (de ahí su nombre del inglés *reTRIEval*). La información almacenada en un *Trie* es un conjunto de claves, donde una clave es una secuencia de símbolos pertenecientes a un alfabeto. Las claves son almacenadas en las hojas del árbol y los nodos internos son pasarelas para guiar la búsqueda. El árbol se estructura de forma que cada letra de la clave se sitúa en un nodo de forma que los hijos de un nodo representan las distintas posibilidades de símbolos diferentes que pueden continuar al símbolo representado por el nodo padre. Por tanto la búsqueda en un *Trie* se hace de forma similar a como se hacen las búsquedas en un diccionario:

Se empieza en la raíz del árbol. Si el símbolo que estamos buscando es A entonces la búsqueda continúa en el subárbol asociado al símbolo A que cuelga de la raíz. Se sigue de forma análoga hasta llegar al nodo hoja. Entonces se compara la cadena asociada al nodo hoja y si coincide con la cadena de búsqueda entonces la búsqueda ha terminado en éxito, si no entonces el elemento no se encuentra en el árbol.

La estructura de datos *Trie* se define como una estructura de datos basada en árboles que se utiliza para almacenar una colección de cadenas y realizar operaciones de búsqueda eficientes en ellas.

Trie sigue una propiedad de que si dos cadenas tienen un prefijo común, entonces tendrán el mismo ancestro en el *Trie*.

Ahora ya sabemos que *Trie* tiene una estructura en forma de árbol. Por eso, es muy importante conocer sus propiedades. A continuación se muestran algunas propiedades importantes de la estructura de datos *Trie*:

- Hay un nodo raíz en cada *Trie*.
- Cada nodo de un *Trie* representa una cadena y cada arista representa un carácter.
- Cada nodo consta de *hashmaps* o una matriz de punteros, y cada índice representa un carácter y una bandera para indicar si alguna cadena termina en el nodo actual.
- La estructura de datos *Trie* puede contener cualquier número de caracteres, incluidos alfabetos, números y caracteres especiales. Por ejemplo para cadenas con caracteres a-z solo se necesitan 26 punteros para cada nodo, donde el índice 0 representa los caracteres *a* y el índice 25 representa los caracteres *z*.
- Cada ruta desde la raíz hasta cualquier nodo representa una palabra o cadena.

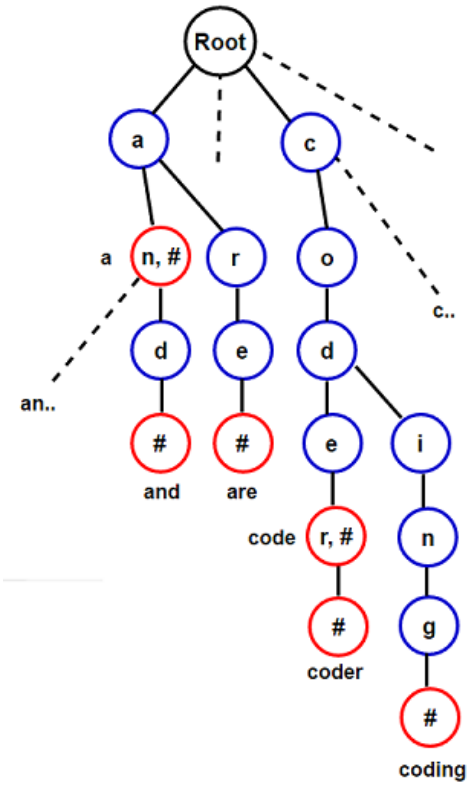
3.1. Representación de *Trie*

Hay varias formas de representar un Trie, que corresponden a diferentes compensaciones entre el uso de la memoria y la velocidad de las operaciones. La forma básica es la de un conjunto vinculado de nodos, donde cada nodo contiene una array de punteros secundarios, uno para cada símbolo en el alfabeto (por lo que para el alfabeto inglés, uno almacenaría 26 punteros secundarios y para el alfabeto de bytes, 256 punteros). El nodo *Trie* también mantiene un indicador que especifica si corresponde o no al final de la clave.

Como se ilustra en la siguiente figura, cada clave se representa en el *Trie* como un camino desde la raíz hasta el nodo interno o una hoja:

Veamos cómo se almacena una palabra *and* y *are* en la estructura de datos de *Trie*:

1. Almacene *and* en la estructura de datos *Trie*:
    - La palabra *and* comienza con *a*, por lo que inicilizaremos el nodo hijo que representa *a* con memoria en el nodo *Trie*, que representa el uso de *a*.
    - Después de colocar el primer carácter, para el segundo carácter nuevamente hay 26 posibilidades, así que desde el nodo *a*, nuevamente hay un nuevo nodo *n*, para almacenar y representar el segundo carácter.
    - El segundo carácter es *n*, así que de *a*, nos moveremos a *n*.



- Después de *n*, el tercer carácter es *d*,nuevamente hay un nuevo nodo *d* que tiene como padre el nodo *n*, para almacenar y representar el tercer carácter. Para dicho nodo se establece una marca que indica que desde la raíz del *Trie* hasta ese nodo hay una cadena.

2. Almacene *are* en la estructura de datos *Trie*:
   - La palabra *are* comienza con *a* y el nodo *a* en el nodo raíz ya se ha creado. Entonces, no es necesario volver a crarlo, solo muévase al nodo *a* en Trie.
   - Después de colocar el primer carácter, para el segundo carácter nuevamente hay 26 posibilidades de las cuales ya existe uno previamente *n*, así que desde el nodo *a*, nuevamente hay un nuevo nodo *r*, para almacenar y representar el segundo carácter.
   - El segundo carácter es *r*, así que de *a*, nos moveremos a *r*.
   - Después de *r*, el tercer carácter es *e*,nuevamente hay un nuevo nodo *e* que tiene como padre el nodo *r*, para almacenar y representar el tercer carácter. Para dicho nodo se establece una marca que indica que desde la raíz del *Trie* hasta ese nodo hay una cadena.

3.2. Operaciones

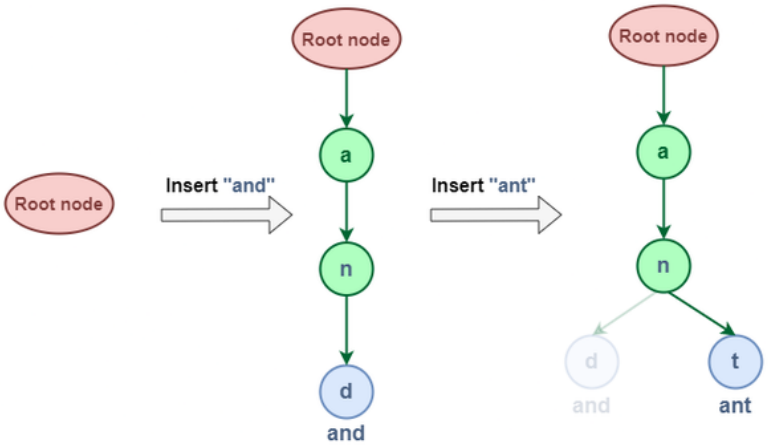
El *Trie* puede presentar varias operaciones en dependencia de la situación donde vaya ser aplicado, pero al menos las operaciones de *Inserción*, *Búsqueda* y *Eliminación* son las más utilizadas o

sirven de base de implementación para otras operaciones que puede ser soportada por la estructura.

3.2.1. Inserción

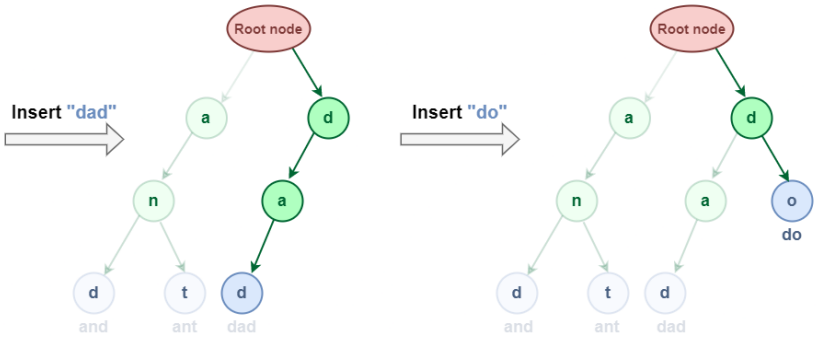
Se procede recorriendo el *Trie* de acuerdo con la string que se va a insertar, y luego agrega nuevos nodos para el sufijo de la string que no está contenido en el *Trie*. Esta operación se utiliza para insertar nuevas cadenas en la estructura de datos de Trie. Veamos cómo funciona esto:

Intentemos insertar *and* y *ant* en este Trie:



De la representación de inserción anterior, podemos ver que la palabra *and* y *ant* han compartido algún nodo común (es decir, *an*), esto se debe a la propiedad de la estructura de datos *Trie* de que si dos cadenas tienen un prefijo común entonces tendrán el mismo ancestro en el *Trie*.

Ahora intentemos insertar *dad* y *do*:



Algoritmo de inserción en la estructura de datos *Trie*:

1. Definir una función insert(TrieNode root, string word) que tomará dos parámetros uno para la raíz y otro para la cadena que queremos insertar en la estructura de datos *Trie*.
2. Ahora tome otro puntero *currentNode* e inicialícelo con el nodo raíz.
3. Itere sobre la longitud de la cadena dada y verifique si el valor es NULL o no en el arreglo de punteros en el carácter actual de la cadena.
   - Si es NULL entonces, crea un nuevo nodo y apunta el carácter actual a este nodo recién creado.
   - Mueva el *currentNode* al nodo recién creado.
4. Finalmente, incremente el número de palabras del último nodo actual, esto implica que hay una cadena que termina en nodo actual.

3.2.2. Búsqueda

La operación de búsqueda en *Trie* se realiza de manera similar a la operación de inserción, pero la única diferencia es que cada vez que encontramos que el arreglo de punteros en el nodo actual no apunta al carácter actual de la palabra, devuelve falso en lugar de crear un nuevo nodo. por ese carácter actual de la palabra.

Esta operación se utiliza para buscar si una cadena está presente en la estructura de datos Trie o no. Hay dos enfoques de búsqueda en la estructura de datos de *Trie*.

1. Encuentra si existe alguna palabra que comience con el prefijo dado en *Trie*
2. Encuentra si la palabra dada existe en *Trie*. Es similar a la búsqueda de prefijos, pero además, debemos verificar si la palabra termina en el último carácter de la palabra o no.

Hay un patrón de búsqueda similar en ambos enfoques. El primer paso para buscar una palabra dada en Trie es convertir la palabra en caracteres y luego comparar cada carácter con el nodo trie del nodo raíz. Si el carácter actual está presente en el nodo, avance a sus hijos. Repita este proceso hasta encontrar todos los caracteres.

3.3. Eliminación

Es un poco complicado La idea es eliminar la clave de forma ascendente utilizando recursión. Se debe tener especial cuidado al eliminar la clave, ya que puede ser el prefijo de otra clave, o su prefijo puede ser otra clave en *Trie*. Hay tres casos cuando se elimina una palabra de *Trie*.

1. **La palabra eliminada es un prefijo de otras palabras en *Trie*:** Como se muestra en la siguiente figura, la palabra eliminada *an* comparte un prefijo completo con otra palabra *and* y *and*. Una solución fácil para realizar una operación de eliminación en este caso es simplemente disminuir el recuento de palabras en 1 en el nodo final de la palabra.
2. **La palabra eliminada comparte un prefijo común con otras palabras en *Trie*:** Como se muestra en la siguiente figura, la palabra eliminada *and* tiene algunos prefijos comunes con

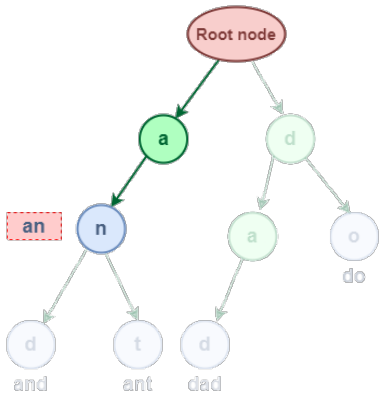


Figura 1: La palabra eliminada es un prefijo de otras palabras en *Trie*

otras palabras *ant*. Comparten el prefijo *an*. La solución para este caso es eliminar todos los nodos desde el final del prefijo hasta el último carácter de la palabra dada.

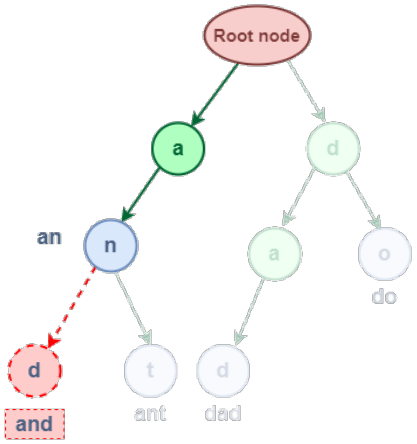


Figura 2: La palabra eliminada comparte un prefijo común con otras palabras en *Trie*

3. **La palabra eliminada no comparte ningún prefijo común con otras palabras en *Trie*:** Como se muestra en la siguiente figura, la palabra *geek* no comparte ningún prefijo común con ninguna otra palabra. La solución para este caso es simplemente eliminar todos los nodos.

Los *Trie* puede ser visto con un árbol de búsqueda y presenta un grupo de ventajas sobre los árboles de búsqueda binaria como son:

- Búsqueda de claves más rápida. La búsqueda de una clave de longitud M tendrá en el peor de los casos un coste de O(M). Un árbol de búsqueda binaria tiene un coste de O ($M \log N$), siendo N el número de elementos del árbol, ya que la búsqueda depende de la profundidad del árbol, logarítmica con el número de claves.

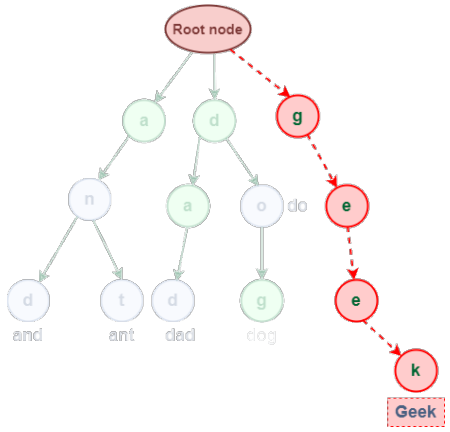


Figura 3: La palabra eliminada no comparte ningún prefijo común con otras palabras en *Trie*

- Menos espacio requerido para almacenar gran cantidad de cadenas pequeñas, puesto que las claves no se almacenan explícitamente.
- Mejor funcionamiento para el algoritmo de búsqueda del prefijo más largo

4. Implementación

4.1. C++

```
struct TrieNode {
   int wordCount;
   unordered_map<char, TrieNode*> children;
   TrieNode() { this->wordCount = 0; children.clear(); }
};

struct Trie {
   TrieNode *root;
   Trie(){root = new TrieNode();}

   void printSorted() {
      string s = "";
      for (unordered_map<char, TrieNode*>::iterator it =
         root->children.begin(); it != root->children.end(); it++) {
         preorder(it->second, s + it->first);
      }
   }

   void preorder(TrieNode *node, string s) {
      for (unordered_map<char, TrieNode*>::iterator it =
         node->children.begin(); it != root->children.end(); it++) {
         preorder(it->second, s + it->first);
```

```
            }
    if(node->wordCount > 0) {
       for (int i = 0; i < node->wordCount; i++) cout << s << ENDL;
    }
}

void insertString(string s) {
    TrieNode *v = root;
    for (auto ch : s) {
       TrieNode *next;
       if (v->children.find(ch) == v->children.end()) {
          next = new TrieNode(); v->children.insert(pair<char, TrieNode*>(ch
              , next));
       }
       else { next = v->children[ch];}
       v = next;
     }
     v->wordCount++;
}

bool searchWord(string key) {
    TrieNode *currentNode = root;
    for (char ch : key) {
       unordered_map<char, TrieNode*>::iterator it=
              currentNode->children.find(ch);
       if(it!=currentNode->children.end()) currentNode = it->second;
       else return false;
    }
    return (currentNode->wordCount > 0);
}

bool searchPrefix(string key) {
    TrieNode *currentNode = root;
    for (char ch : key) {
       unordered_map<char, TrieNode*>::iterator it =
         currentNode->children.find(ch);
       if (it != currentNode->children.end()) currentNode = it->second;
       else return false;
    }
    return true;
}

void deleteKey(string s){ deleteKey(root, s, 0);}

TrieNode* deleteKey(TrieNode *root, string word, int depth) {
    if (root == NULL) return NULL;
    if (depth == word.length()) {
       root->wordCount--;
       if (isEmpty(root)) root = NULL;
       return root;
```

```
        }
        char index = word[depth];

        TrieNode * t = deleteKey(root->children[index], word, depth + 1);
        if (t == NULL){
            unordered_map<char, TrieNode*>::iterator it = root->children.find(
                index);
            if(it!=root->children.end()) root->children.erase(it);
        }
        if (isEmpty(root) && root->wordCount == 0) root = NULL;
        return root;
    }

    bool isEmpty(TrieNode *node){
        return node->wordCount==0 && node->children.size()==0;
    }
};
```

4.2. Java

```
private class Trie {
    private class TrieNode {
        Map<Character, TrieNode> children;
        int wordCount;
        public TrieNode() {
            children = new TreeMap<>();wordCount = 0;
        }
    }

    private TrieNode root;

    public Trie() { root = new TrieNode();}

    public void printSorted() {
        String s = "";
        for (Character ch : root.children.keySet()){
            preorder(root.children.get(ch),s+ch);
        }
    }

    private void preorder(TrieNode node, String s) {
        for (Character ch : node.children.keySet())
            preorder(node.children.get(ch), s + ch);

        if (node.wordCount > 0)
            for(int i=0;i<node.wordCount;i++) System.out.println(s);
    }
```

```
    public void insertString(String s) {
        TrieNode v = root;
        for (char ch : s.toCharArray()) {
            TrieNode next = v.children.get(ch);
            if (next == null) v.children.put(ch, next = new TrieNode());
            v = next;
        }
        v.wordCount++;
    }

    public boolean searchWord(String key) {
        TrieNode currentNode = root;
        for (char ch : key.toCharArray()) {
            TrieNode next = currentNode.children.get(ch);
            if (next == null) return false;
            currentNode = next;
        }
        return (currentNode.wordCount > 0);
    }

    public boolean searchPrefix(String key) {
        TrieNode currentNode = root;
        for (char ch : key.toCharArray()) {
            TrieNode next = currentNode.children.get(ch);
            if (next == null) return false;
            currentNode = next;
        }
        return true;
    }

    public void deleteKey(String word){ deleteKey(root, word, 0); }

    private TrieNode deleteKey(TrieNode root, String word, int depth) {
        if (root == null) return null;
        if(depth == word.length()) {
            root.wordCount--;
            if (isEmpty(root)) root = null;
            return root;
        }
        char index = word.charAt(depth);
        TrieNode t = deleteKey(root.children.get(index), word, depth + 1);
        if (t == null) root.children.remove(index);
        if (isEmpty(root) && root.wordCount == 0) root = null;
        return root;
    }

    private boolean isEmpty(TrieNode node) {
        return node.wordCount == 0 && node.children.size() == 0;
    }
}
```

5. Aplicaciones

Entre las aplicaciones que podemos encontrar al *Trie* está la sustitución de otras estructuras de datos:

- **Como representación de diccionarios:** Una aplicación frecuente de los *Tries* es el almacenamiento de diccionarios, como los que se encuentran en los teléfonos móviles. Estas aplicaciones se aprovechan de la capacidad de los *Tries* para hacer búsquedas, inserciones y borrados rápidos. Sin embargo, si sólo se necesita el almacenamiento de las palabras (p.ej. no se necesita almacenar información auxiliar de las palabras del diccionario) un autómata finito determinista acíclico mínimo usa menos espacio que un *Trie*. Autocompletar (o completar palabras) es una función en la que una aplicación predice el resto de una palabra que el usuario está escribiendo.
- **Corrector ortográfico:** El corrector ortográfico marca las palabras de un documento que pueden no estar escritas correctamente. Los correctores ortográficos se usan comúnmente en procesadores de texto, clientes de correo electrónico, motores de búsqueda, etc.
- **Como reemplazo de otras estructuras de datos:** *Trie* tiene varias ventajas sobre árbol de búsqueda binaria. También puede reemplazar una tabla hash ya que la búsqueda es generalmente más rápida en *Trie*, incluso en el peor de los casos. Además, no hay colisiones de diferentes claves en un *Trie*, y un *Trie* puede proporcionar un orden alfabético de las entradas por clave.
- **Ordenación lexicográfico de un juego de llaves:** Ordenación lexicográfico de un juego de llaves se puede lograr con un algoritmo simple basado en *Trie*. Inicialmente insertamos todas las claves en un *Trie* y luego imprimimos todas las claves en el *Trie* realizando recorrido de pedido anticipado (primer recorrido en profundidad), lo que da como resultado un orden lexicográficamente creciente.
- **Coincidencia de prefijo más largo:**Los enrutadores utilizan el prefijo más largo algoritmo de coincidencia en redes de protocolo de Internet (IP) para seleccionar una entrada de una tabla de reenvío.

Es muy útil para conseguir búsquedas eficientes en repositorios de datos muy voluminosos. La forma en la que se almacena la información permite hacer búsquedas eficientes de cadenas que comparten prefijos. También son útiles en la implementación de algoritmos de correspondencia aproximada, como los usados en el software de corrección ortográfica.

6. Complejidad

La complejidad temporal de una estructura de datos Trie para la operaciónes de inserción/eliminación/búsqueda es solo O(N), dónde N es la longitud de la clave. La complejidad espacial de

dichas operaciones la inserción su complejidad es O($N \times M$) donde N es la longitud de la cadena almacenar y M es la cantidad de cadenas almacenadas, mientras para las otras dos operaciones es O(1).

La complejidad espacial de una estructura de datos Trie es O($N \times M \times C$), dónde N es el número total de strings, M es la longitud máxima de la string, y C es el tamaño del alfabeto. El problema de almacenamiento se puede aliviar si solo asignamos memoria para los alfabetos en uso y no desperdiciamos espacio almacenando punteros nulos.

7. Ejercicios propuestos

A continuación una lista de ejercicios que se pueden resolver con el uso de esta estructura de datos:

- DMOJ - Contactos
- DMOJ - Lista de teléfonos