Universidad Nacional de Cuyo
Facultad de Ingeniería
Licenciatura en Ciencias de la Computación

Trabajo Práctico Nº 3

Algoritmos y Estructuras de Datos

Árboles N-arios: Trie

2024

Gonzalo Padilla Lumelli

Abril 2024

Parte 1

Importante: Los ejercicios de esta primera parte tienen como objetivo codificar las diferentes funciones básicas necesarias para implementar un Trie.

A partir de estructuras definidas como:

```
class Trie:
    root = None

class TrieNode:
    parent = None
    children = None
    key = None
    isEndOfWord = None
```

Ejercicio 1

Crear un módulo de nombre **trie.py** que implemente las siguientes especificaciones de las operaciones elementales para el **TAD Trie**.

insert(T, element)
Descripción: insert un elemento en T, siendo T un Trie.
Entrada: El Trie sobre la cual se quiere agregar el elemento (Trie) y el valor del elemento (palabra) a agregar.
Salida: No hay salida definida.

search(T, element)
Descripción: Verifica que un elemento se encuentre dentro del Trie.
Entrada: El Trie sobre la cual se quiere buscar el elemento (Trie) y el valor del elemento (palabra).
Salida: Devuelve False o True según se encuentre el elemento.

Solución

```
def insert(T, element):
    if not element or not T:
        return None

    if not T.root:
        T.root = TrieNode()
        T.root.children = [None] * ALPHABET_SIZE

    node = T.root
    element = element.lower()
    for i, char in enumerate(element):
        indexChild = ord(char) - ord('a')
        nextNode = node.children[indexChild]
        if not nextNode:
            node.children[indexChild] = TrieNode()
            nextNode = node.children[indexChild]
            nextNode.children = [None] * ALPHABET_SIZE
            nextNode.key = char
            nextNode.parent = node
        if i == len(element) - 1:
            nextNode.isEndOfWord = True
        node = nextNode

    return None
```

```
def search(T, element):
    return bool(_findLastNodeOfWord(T, element))

def _findLastNodeOfWord(T, element):
    if not T or not T.root or not element:
        return False

    node = T.root
    element = element.lower()
    for i, char in enumerate(element):
        node = node.children[ord(char) - ord('a')]
        if (not node or
            node.key != char or
            (i == len(element) - 1 and not node.isEndOfWord)):
            return False

    return node
```

Ejercicio 2 (no code)

Sabiendo que el orden de complejidad para el peor caso de la operación search() es de $O(m|\Sigma|)$. Proponga una versión de la operación search() cuya complejidad sea $O(m)$.

Solución

La implementación de search() en el ejercicio 1 ya es de orden O(m), ya que en cada nodo, no es necesario buscar al siguiente nodo hijo recorriendo una lista de largo $|\Sigma|$, sino que se accede a él directamente mediante acceso indexado en un arreglo de longitud $|\Sigma|$. Si se busca el nodo con la letra 'a', simplemente se accede al elemento 0 del arreglo. Si se quiere el nodo con la letra 'b', se accede al índice 1, y así para cualquier letra. Para calcular el índice correspondiente a cada letra, se resta 97 al valor asociado a la misma en ASCII (la letra debe ser minúsucula).

Ejercicio 3

delete(T, element)

Descripción: Elimina un elemento se encuentre dentro del Trie.
Entrada: El Trie sobre la cual se quiere eliminar el elemento (Trie) y el valor del elemento (palabra) a eliminar.
Salida: Devuelve False o True según se haya eliminado el elemento.

Solución

```python
def delete(T, element):
    if not T or not T.root or not element:
        return False

    node = _findLastNodeOfWord(T, element) # final de palabra, si existe

    if not node:
        return False # Si no existe, no hacemos nada

    node.isEndOfWord = False

    # Mientras no tenga hijos ni sea fin de otra palabra, borramos nodo:
    while (not any(child for child in node.children) and not node.isEndOfWord):
        node.parent.children[ord(node.key)-ord('a')] = None # borramos el nodo
        node = node.parent # pasamos a su padre

    return True
```

Parte 2

Ejercicio 4

Implementar un algoritmo que dado un árbol Trie T, un patrón p (prefijo) y un entero n, escriba todas las palabras del árbol que empiezan por p y sean de longitud n.

Solución

```python
def printWordsWithPrefixAndLength(t, prefix, length):
    if not t or not t.root or not prefix or not length:
        return

    # Si el prefijo es mayor al largo de
    # palabra, retornamos
    if length < len(prefix):
        return

    # Encontrar node final del prefijo
    node = t.root
    prefix = prefix.lower()
    for c in prefix:
        node = node.children[ord(c) - ord('a')]
        if not node or node.key != c:
            return

    # Imprime palabras a partir de un nodo,
    # que tengan el largo indicado
    def printWordsFromNodeWithLength(node, word):
        if not node:
            return

        # Si llegamos al largo indicado,
        # imprimimos si es final de palabra
        # Si no, cortamos
        if len(word) == length:
            if node.isEndOfWord:
                print(word)
            return

        # Si todavía no llegamos al largo indicado
        # recorremos cada hijo del nodo
        for child in node.children:
            if child:
                printWordsFromNodeWithLength(child, word + child.key)

    # Imprimimos palabras del largo indicado
    # partiendo desde el final del prefijo
    printWordsFromNodeWithLength(node, prefix)
```

Ejercicio 5

Implementar un algoritmo que dado los Trie T1 y T2 devuelva True si estos pertenecen al mismo documento y False en caso contrario. Se considera que un Trie pertenece al mismo documento cuando:

1. Ambos Trie sean iguales (esto se debe cumplir)
2. Si la implementación está basada en LinkedList, considerar el caso donde las palabras hayan sido insertadas en un orden diferente.

En otras palabras, analizar si todas las palabras de T1 se encuentran en T2.

Analizar el costo computacional.

Solución

```python
def areEqual(t1, t2):
    if not t1 or not t2:
        return False

    # Caso trivial, referencias al mismo dato en memoria
    if t1 == t2:
        return True

    # Compara nodo a nodo, recursivamente
    def compareTrees(node1, node2):
        # Si ambos son None
        if not node1 and not node2:
            return True


        # Si uno es None y el otro no
        if not node1 or not node2:
            return False


        # Si difieren en alguna forma
        if (node1.key != node2.key
            or node1.isEndOfWord != node2.isEndOfWord
            or len(node1.children) != len(node2.children)):
            return False

        # Comparar hijos
        for i in range(len(node1.children)):
            # Si alguno difiere, ya podemos retornar falso
            if not compareTrees(node1.children[i], node2.children[i]):
                return False

        # Se cumplió todo, retornamos que son iguales
        return True

    # Comparamos ambos árboles a partir de la raíz
    return compareTrees(t1.root, t2.root)
```

Ejercicio 6

Implemente un algoritmo que dado el Trie T devuelva True si existen en el documento T dos cadenas invertidas. Dos cadenas son invertidas si se leen de izquierda a derecha y contiene los mismos caracteres que si se lee de derecha a izquierda, ej: abcd y dcba son cadenas invertidas, gfdsa y asdfg son cadenas invertidas, sin embargo abcd y dcka no son invertidas ya que difieren en un carácter.

Solución

```python
def hasInvertedWords(t):
    if not t or not t.root:
        return None

    # Recorremos el trie
    def traverseTrie(word, node):
        if not node:
            return False

        # Vamos calculando la palabra actual
        if node.key:
            word = word + node.key

        # Si es final de palabra, buscamos su inverso en el trie
        # Si está, retornamos verdadero, si no, revisamos los hijos
        if node.isEndOfWord and search(t, word[::-1]):
            return True

        # Retornamos si se cumple para alguno de sus hijos
        return any([traverseTrie(word, child) for child in node.children])

    return traverseTrie("", t.root)
```

Ejercicio 7

Un corrector ortográfico interactivo utiliza un Trie para representar las palabras de su diccionario. Queremos añadir una función de auto-completar (al estilo de la tecla TAB en Linux): cuando estamos a medio escribir una palabra, si sólo existe una forma correcta de continuarla entonces debemos indicarlo.

Implementar la función autoCompletar(Trie, cadena) dentro del módulo trie.py, que dado el árbol Trie T y la cadena devuelve la forma de auto-completar la palabra. Por ejemplo, para la llamada autoCompletar(T, 'groen') devolvería "land", ya que podemos tener "groenlandia" o "groenlandés" (en este ejemplo la palabra groenlandia y groenlandés pertenecen al documento que representa el Trie). Si hay varias formas o ninguna, devolvería la cadena vacía. Por ejemplo, autoCompletar(T, ma') devolvería "" (cadena vacia) si T presenta las cadenas "madera" y "mama"

Solución

```python
def autoCompletar(t, cadena):
    if not t or cadena is None:
        return None

    result = ""

    # Buscamos el último nodo de la cadena
    # Retornamos string vacío "" si no está
    node = t.root
    cadena = cadena.lower()
    for char in cadena:
        node = node.children[ord(char) - ord('a')]
        if (not node or node.key != char):
            return result

    while True:
        # Obtenemos los hijos que existen
        notNoneChildren = [child for child in node.children if child]

        # Si hay distinto de 1, o el nodo actual es final de palabra
        # retornamos el resultado actual
        if len(notNoneChildren) != 1 or node.isEndOfWord:
            break

        # Si no, sumamos la siguiente letra al resultado
        onlyChild = notNoneChildren[0]
        result += onlyChild.key
        node = onlyChild

    return result
```