UNCUYO - Licenciatura en ciencias de la computación
Augusto Robles 11737

Algoritmos y estructuras de datos II

Árboles N-arios:Trie

Parte 1

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

A partir de estructuras definidas como :

```
class Trie:
      root = None

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

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.

```python
# EJERCICIO 1

def insert(T,element):
    if element == '':
        return
    index = 0
    insert_recursive(T.root, element, index)

def insert_recursive(node, element, index):
    if index > len(element) - 1:
        return
    parent = node
    if node.children != []:
        for node in node.children:
            if element[index] == node.key:
                insert_recursive(node, element, index + 1)
                return

    while index <= len(element) - 1:
        newNode = TrieNode()
        newNode.key = element[index]
        newNode.parent = parent
        parent.children.append(newNode)
        parent = newNode
        index += 1

    newNode.isEndOfWord = True
    return
```

```python
def search(T,element):
    if element == '':
        return False
    index = 0
    return search_recursive(T.root, element, index)

def search_recursive(node, element, index):
    if index > len(element) - 1:
        if node.isEndOfWord == True:
            return True
        else:
            return False

    if node.children != []:
        for node in node.children:
            if element[index] == node.key:
                return search_recursive(node, element, index + 1)
    return False
```

Ejercicio 2

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

Suponiendo que trabajamos con el abecedario inglés, letras minúsculas, tenemos 26 caracteres posibles para formar palabras.

La idea de un search() cuya complejidad sea O(m) se basa fuertemente en la utilización del código ascii de cada carácter, siendo el código ascii de a=97 y el de z=122. En vez de utilizar una linked list para almacenar los hijos de un nodo del Trie, utilizamos un array de tamaño 26.

Se pasa “a” cuyo codigo ascii es 97, se le resta 97, obtenemos el indice 0.
Se pasa “z” cuyo código ascii es 122, se le resta 97, obtenemos el indice 25.

Es decir que para todos los caracteres podremos acceder al indice de almacenamiento de esta manera, resultando en complejidad O(1), luego si la longitud de la palabra buscada es m, la nueva versión del search() tendrá una complejidad de O(m).

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.

```python
# EJERCICIO 3

def delete(T,element):
  if search(T,element) == False:
    return False

  node = T.root
  for ch in element:  ### Recorre cada letra del elemento (palabra) ###
    index = traverse_list(node.children,ch)
    if index == None:  ### No se encuentra la letra en la lista ###
      return False
    else:  ### Se encuentra la letra en la lista ###
      node = node.children[index]  ### Sigue al proximo nodo ###
  if node.children != []:  ### Si la palabra a eliminar es parte de una palabra mas larga (hola,holanda) ###
    node.isEndOfWord == False
    return True
  else:
    delete_node(T,node)
    return True

def delete_node(T,node):
  temp = node
  node = node.parent
  node.children.remove(temp)
  if node.isEndOfWord == True or node == T.root or len(node.children) > 0:
    return True
  else:
    delete_node(T,node)

def traverse_list(list, ch):
    for index, i in enumerate(list):
      if i.key == ch:
        return index
    return None
```

Parte 2

Ejercicio 4

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

```python
# EJERCICIO 4

def startsWith(T, p, n):
    trieWords = print_trie_words(T)
    patternWords = []
    for word in trieWords:
        if p == word[0:len(p)]:
            # Si no se especifica longitud
            if n == None:
                patternWords.append(word)
            # Si se especifica longitud
            if len(word) == n:
                patternWords.append(word)
    return patternWords
```

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** pertenecen al mismo documento cuando:

1. Ambos Trie sean iguales (esto se debe cumplir)
2. ~~El Trie T1 contiene un subconjunto de las palabras del Trie T2~~
3. 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.

```python
# EJERCICIO 5

def sameTries(T1,T2):
    T1List = print_trie_words(T1)
    T2List = print_trie_words(T2)
    if T1List == T2List:
        return True
    return False
```

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

```python
# EJERCICIO 6

def has_inverted_words(T):
    trieWords = print_trie_words(T)
    for word in trieWords:
        reversedWord = "".join(reversed(word))
        if reversedWord in trieWords:
            return True
    return False
```

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 ***"pal"*** 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 "" si **T** presenta las cadenas **"madera"** y **"mama"**.

```python
# EJERCICIO 7

def auto_complete(T,string):
    patternWords = startsWith(T,string,None)
    if len(patternWords) == 1:
        return patternWords[0].replace(string,'')
    else:
        return ''
```

```python
# Funciones auxiliares

def print_trie_words(T):
    if T.root!=None:
        content=[]
        print_trie_recursive(T.root.children,content,"")
        return content
    else:
        return

def print_trie_recursive(children,wordsList,prefix):
    for node in children:
        if node.isEndOfWord:
            wordsList.append(prefix + node.key)
        if node.children!=None:
            print_trie_recursive(node.children, wordsList, prefix + node.key)
    return
```