

Algoritmos y Estructura de Datos II

Árboles N-arios: Trie

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.

```

1  class Trie:
2      root = None
3  class TrieNode:
4      parent = None
5      children = None
6      key = None
7      isEndOfWord = False
8
9  # ---- Ejercicio 1 ----
10
11 def insert(T, element):
12     #insertar un elemento en T, siendo T un Trie.
13     if element==None:
14         #No se ingresó palabra a insertar
15         return
16     if T.root==None:
17         #No hay Tree, se crea uno
18         newRoot=TrieNode()
19         newRoot.key=" "
20         T.root=newRoot
21     index=0; charIndex=0
22     insertElement(T.root,element,index,charIndex)
23
24 def insertElement(current,element,index,charIndex):
25     if charIndex==len(element):
26         #ya cubrí toda la palabra
27         current.isEndOfWord=True
28         return
29     if current.children==None:
30         #Compruebo que el children no sea ya una lista creada (para no resetearla)
31         current.children=[]
32     else:
33         #Para chequear si inserta
34         printList(current.children)
35         for i in range(len(current.children)):
36             if current.children[i].key==element[charIndex]:

```

```

36         if current.children[i].key==element[charIndex]:
37             #Comparo el caracter actual con los elementos del current.children, si
38             # son iguales vuelvo a llamar a la función
39             index=i
40             insertElement(current.children[i],element,index,charIndex+1)
41             return
42         #No son iguales
43         newNode=TrieNode()
44         newNode.key=element[charIndex]
45         newNode.parent=current
46         current.children.append(newNode)
47         index=len(current.children)-1
48         charIndex+=1
49         #Para chequear si inserta (2)
50         printList(current.children)
51         insertElement(current.children[index],element,index,charIndex)
52

```

```

# ---- Extras ----
def printList(L):
    print("[ ", end="")
    for i in range(0,len(L)):
        print(L[i].key, end=" ")
    print("] ", end="")
    print("")

```

```

53 def search(T,element):
54     #Verifica que un elemento se encuentre dentro del Trie.
55     if T.root==None:
56         #Arbol vacío
57         return False
58     if element==None:
59         #Palabra vacía
60         return False
61     index=0
62     check=False
63     check=searchWord(T.root.children,element,index,check)
64     return check
65
66 def searchWord(current,element,index,check):
67     if index<len(element):
68         #Evita que se rompa si busco una palabra mas chica que las que ya hay en el árbol
69         if current!=None:
70             for i in range(0,len(current)):
71                 if current[i].key==element[index]:
72                     #Comparo a ver si son iguales
73                     if (current[i].isEndOfWord==True) and (index==len(element)-1):
74                         #Ya cubrí la palabra
75                         check=True
76                         return check
77                     return searchWord(current[i].children,element,index+1,check)
78             return check
79         else:
80             return check
81     else:
82         return check
83

```

Ejercicio 2

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 $O(m)$.

Para que sea de complejidad $O(m)$ podría usar Arrays para su implementación.

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.

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**.

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.

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.

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”**.