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.

```python
class Trie:
  root = None
class TrieNode:
  parent = None
  children = None
  key = None
  isEndOfWord = False

# ---- Ejercicio 1 ----

def insert(T, element):
  #insertar un elemento en T, siendo T un Trie.
  if element==None:
    #No se ingresó palabra a insertar
    return
  if T.root==None:
    #No hay Tree, se crea uno
    newRoot=TrieNode()
    newRoot.key=" "
    T.root=newRoot
  index=0; charIndex=0
  insertElement(T.root,element,index,charIndex)

def insertElement(current,element,index,charIndex):
  if charIndex==len(element):
    #ya cubrí toda la palabra
    current.isEndOfWord=True
    return
  if current.children==None:
    #Compruebo que el children no sea ya una lista creada (para no resetearla)
    current.children=[]
  else:
    #Para checkear si inserta
    printList(current.children)
  for i in range(len(current.children)):
    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 checkear 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 |Σ|). 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.

```
84  def delete(T,element):
85    element=element.upper()
86    if search(T,element)==False:
87      #No se encuentra la palabra dentro del árbol asi que retorna Falso
88      print("Element not Found")
89      return False
90    charIndex=0
91    deleteElement(T.root,element,charIndex)
92    return True
93
94
95  def deleteElement(current,element,charIndex):
96    #Current = Node
97    for i in range(0,len(current.children)):
98      if current.children[i].key==element[charIndex]:
99        if current.children[i].isEndOfWord==True and charIndex==len(element)-1:
100         if current.children[i].children!=None:
101           #Si llego al fin de palabra pero la palabra a eliminar tiene hijos
102           current.children[i].isEndOfWord= False
103         else:
104           if len(current.children)>1:
105             #si la lista children tiene más de un nodo entonces no tengo que borrar el
106             # nodo current porque desvinculo el otro elemento de la lista children
107             current.children.pop(i)
108             return
109           else:
110             return unlink(current,element,charIndex)
111         return
112       deleteElement(current.children[i],element,charIndex+1)
113       return
114   return
```

```
115
116 def unlink(current,element,charIndex):
117   #Current = Node
118   for i in range (0,len(current.children)):
119     if current.children[i].key==element[charIndex]:
120       if len(current.children)>1:
121         current.children.pop(i)
122         return
123       current.children.pop(i)
124       if current.isEndOfWord==True:
125         return
126   unlink(current.parent,element,charIndex-1)
```

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
def Ejercicio4(T,p,n):
    #dado un árbol Trie T, un patrón p y un entero n, escribe todas la palabras del árbol
    # que empiezan por p y sean de longitud n
    if T.root==None or p==None or n==None:
        return
    #Para que no haya problemas hacemos que la cadena sea en mayusculas
    p=p.upper()
    ListTree=[]
    ListTree=ListOfTreeWords(T.root.children,ListTree,"")
    correctwords=[]
    #Creamos otra lista igual
    correctwords=ListTree.copy()
    #comparar las palabras con p y n e ir sacandolas en la lista resultante
    for word in ListTree:
        for i in range(0,len(p)):
            if word[i]!=p[i]:
                correctwords.remove(word)
                break
            else:
                if len(word)!=n:
                    correctwords.remove(word)
                    break
    return correctwords
```

```python
def ListOfTreeWords(currentL,L,word):
    #Funcion para poner cada palabra de un Trie en una lista (cada elemento de la lista es una palabra)
    #Le paso T.root.children (currentL)
    for current in currentL:
        if current.children!=None:
            if len(current.children)>1:
                for child in current.children:
                    word=word+current.key
                    L=ListOfTreeWords([child],L,word)
                    word=""
            else:
                word=word+current.key
                if current.isEndOfWord==True:
                    L.append(word)
                L=ListOfTreeWords(current.children,L,word)
                word=""
        else:
            word=word+current.key
            L.append(word)
    return L
```

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.

```
153  #Ejercicio 5
154  def CompareTrees(T1,T2):
155      #analizar si todas las palabras del árbol Trie T1 se encuentran en el árbol T2.
156      if T1.root==None or T2.root==None:
157          #Alguno de los árboles no existe
158          return False
159      # Pongo todas las palabras de los trie en 2 listas distintas
160      T1List=[]
161      T1List=ListOfTreeWords(T1.root.children,T1List,"")
162      T2List=[]
163      T2List=ListOfTreeWords(T2.root.children,T2List,"")
164      #printListNormal(T1List)
165      #printListNormal(T2List)
166      #ordeno las listas
167      T1List.sort()
168      T2List.sort()
169      if T1List==T2List:
170          #Comparo las listas para ver si son iguales(implica que tienen las mismas palabras)
171          return True
172      else:
173          return False
```

```
def printListNormal(L):
    print("[ ", end="")
    for i in range(0,len(L)):
        print(L[i], end=" ")
    print("] ", end="")
    print("")
```

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.

```
176  def Ejercicio6(T):
177      #Devuelve True si existen en el documento T dos cadenas invertidas. (ejemplo abcd y dcba)
178      L1=[]
179      L2=[]
180      #Coloco las palabras del Trie en dos listas
181      L1=ListOfTreeWords(T.root.children,L1,"")
182      L2=ListOfTreeWords(T.root.children,L2,"")
183      for i in range(0,len(L2)):
184          #Doy vuelta las palabras de la segunda lista
185          L2[i]=L2[i][::-1]
186      #printListNormal(L1)
187      #printListNormal(L2)
188      #comparo todas las palabras de la primera lista con todas de la segunda
189      for word1 in L1:
190          for word2 in L2:
191              if word1==word2:
192                  return True
193      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"**.