Trabajo Práctico: 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

```
18  def insert(T,element):
19      if T != None:
20          #Caso 1: El árbol está vacío
21          if T.root == None:
22              T.root = trienode()
23              T.root.isendofword = True
24              T.root.children = linkedlist.LinkedList()
25              linkedlist.add(T.root.children,trienode())
26              T.root.children.head.value.parent = T.root
27              insertword(T.root.children.head,element,0)
28              return T
```

```
        #Caso 2: El árbol no está vacio
        else:
            end = False
            i = 0
            list = T.root.children
            currentnode = list.head
            while end == False:
                if currentnode.value.key == element[i]:
                    i += 1
                    #La palabra está incluida dentro de otra
                    if i == len(element):
                        currentnode.value.isendofword = True
                        return T

                    list = currentnode.value.children
                    #Se agrega una nueva lista en el children del nodo
                    if list == None:
                        currentnode.value.children = linkedlist.LinkedList()
                        list = currentnode.value.children
                        linkedlist.add(list,trienode())
                        list.head.value.parent = currentnode
                        end = True
                    else:
                        currentnode = list.head
                elif currentnode.nextNode == None:
                    #Se necesitan nodos extras en la lista
                    linkedlist.add(list,trienode())
                    list.head.value.parent = currentnode.value.parent
                    end = True
                else:
                    currentnode = currentnode.nextNode

            insertword(list.head,element,i)
            return T
```

```
#Agregar la palabra en el árbol
def insertword(currentnode,word,i):
    for i in range(i,len(word)):
        currentnode.value.key = word[i]
        if i == len(word)-1:
            currentnode.value.isendofword = True
            return
        currentnode.value.children = linkedlist.LinkedList()
        linkedlist.add(currentnode.value.children,trienode())
        currentnode.value.children.head.value.parent = currentnode
        currentnode = currentnode.value.children.head
```

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.

```
84  def search(T,element):
85      if T.root != None and len(element) != 0:
86          return searchR(T.root.children.head,element,0)
87      else:
88          return
89
90  def searchR(currentword,word,index):
91      if index == len(word) or currentword == None:
92          return False
93
94      if currentword.value.key == word[index]:
95          if currentword.value.isendofword == True and index == len(word)-1:
96              return True
97          elif currentword.value.children == None:
98              return False
99          else:
100             return searchR(currentword.value.children.head,word,index+1)
101     else:
102         return searchR(currentword.nextNode,word,index)
```

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).

Respuesta: Una versión del search con complejidad de O(m) sería una implementación del trie pero con arrays o listas de Python ya que cuando queremos buscar una letra en el respectivo array o lista solamente accedemos a el utilizando una key. Por lo tanto acceder a esa letra nos queda de O(1) y recorrer la palabra 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.

```
125   def delete(T,element):
126       #Buscamos la última letra de la palabra si es que existe
127       node = searchNode(T.root.children.head,element,0)
128       if node != None:
129           #Caso donde la palabra esta incluida en otra
130           if node.value.children != None:
131               node.value.isendofword = False
132               return True
133           node.value.isendofword = False
134           return deleteR(node,T)
```

```
137 def deleteR(currentnode,T):
138     #Caso donde la palabra a eliminar tiene una palabra en su cadena
139     if currentnode.value.isendofword == True:
140         return True
141 
142     parent = currentnode.value.parent
143     #Si el parent es justo la raíz del arbol eliminamos la letra y hacemos none al children de la raíz SI LA LISTA QUEDÓ VACÍA
144     if parent == T.root:
145         linkedlist.delete(parent.children,currentnode.value)
146         if parent.children.head == None:
147             parent.children = None
148             return True
149         return True
150 
151     list = currentnode.value.parent.value.children
152     linkedlist.delete(list,currentnode.value)
153     if list.head != None:
154         return True
155 
156     parent.value.children = None
157     return deleteR(parent,T)
```

```
110   def searchNode(currentword,word,index):
111       if index == len(word) or currentword == None:
112           return
113 
114       if currentword.value.key == word[index]:
115           if currentword.value.isendofword == True and index == len(word)-1:
116               return currentword
117           elif currentword.value.children == None:
118               return
119           else:
120               return searchNode(currentword.value.children.head,word,index+1)
121       else:
122           return searchNode(currentword.nextNode,word,index)
```

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 wordswithpattern(Trie,p,n):
    pattern = searchpattern(Trie.root.children.head,p,0)
    if pattern != None:
        #El patrón p tiene la misma longitud que n
        if len(p) == n: return
        #El patrón p no tiene children
        if pattern.value.children == None: return
        word = p
        wordswithpatternR(pattern.value.children.head,n,len(p)+1,word)


def wordswithpatternR(currentnode,n,i,word):
    if currentnode == None:
        return
    word = word + currentnode.value.key
    if i == n and currentnode.value.isendofword == True:
        print(word)
        print(" ")
        return
    if currentnode.value.children != None:
        wordswithpatternR(currentnode.value.children.head,n,i+1,word)
    wordswithpatternR(currentnode.nextNode,n,i,word[:-1])
```

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.

Respuesta: El orden de complejidad del algoritmo es de O(m*n + s(Σ)). Primero siendo m el tamaño del T1 y n el tamaño del T2, luego tenemos s(Σ) que es recorrer cada palabra del trie.

```python
def sametrie(t1,t2):
    t1words = []
    word = ""
    getallwordR(t1.root.children.head,word,t1words)
    for i in range(0,len(t1words)):
        if search(t2,t1words[0]) == False:
            return False
    return True

#Función que retorna una lista con todas las palabras del trie
def getallwordR(currentnode,word,list):
    if currentnode == None:
        return
    word = word + currentnode.value.key
    if currentnode.value.isendofword == True:
        list.append(word)

    if currentnode.value.children != None:
        getallwordR(currentnode.value.children.head,word,list)

    getallwordR(currentnode.nextNode,word[:-1],list)
```

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.

```python
def inverted(T):
    twords = []
    word = ""
    getallwordR(T.root.children.head,word,twords)
    newlist = []
    for i in range(0,len(twords)):
        wordinverted = twords[i]
        newlist.append(wordinverted[::-1])

    finalist = []
    for i in range(0,len(newlist)):
        if search(T,newlist[i]) == True:
            finalist.append(newlist[i])

    if len(finalist) != 0:
        print("Se encuentran al menos dos cadenas invertidas en el trie")
        print(finalist)
        return
    print("No se encontraron cadenas invertidas")
```

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"**.

```
240  def searchpattern(currentnode,p,index):
241      if currentnode == None:
242          return
243
244      if currentnode.value.key == p[index]:
245          if index == len(p)-1:
246              return currentnode
247          if currentnode.value.children != None:
248              return searchpattern(currentnode.value.children.head,p,index+1)
249          return
250      return searchpattern(currentnode.nextNode,p,index)
251
252
253  def autoCompletar(Trie,cadena):
254      pattern = searchpattern(Trie.root.children.head,cadena,0)
255      if pattern == None:
256          print("No se encontraron palabras")
257          return
258      if pattern.value.children == None:
259          print("")
260          return
261      word = cadena
262      list = []
263      getallwordR(pattern.value.children.head,word,list)
264      print(list)
```