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
```

Sugerencia 1: Para manejar múltiples nodos, el campo children puede contener una estructura **LinkedList** conteniendo **TrieNode**

~~Para trabajar con cadenas, utilizar la clase string del módulo **algo.py.**~~

~~uncadena = **String**(“esto es un string”)~~

~~Luego es posible acceder a los elementos de la cadena mediante un índice.~~

~~print(unacadena[1]))~~
~~>>> s~~

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

```
def insert(T,element):
    #La lista no esta creada
    if T.root.children == None:
    #creo la lista
        T.root.children = LinkedList()
    #funcion recursiva que insertara el resto de los caracteres
    insertR(T.root.children,element,0,T.root)

def insertR(L, palabra, caracter,parent):
    #caso base
    if caracter == len(palabra):
        return

    #debo averiguar si ya existe el caracter que quiero insertar
    if L == None:
        current = TrieNode()
        current.parent = parent
        current.children = LinkedList()
        current.key = palabra[caracter]
        add(L, current)
    else:
        current = L.head
        while current != None:
            #busco el caracter en la lista
            if current.value.key == palabra[caracter]:
                current = current.value
                break
            current = current.nextNode
        #si el caracter no existe en la lista, lo agrego
        if current == None:
            current = TrieNode()
            current.parent = parent
            current.children = LinkedList()
            current.key = palabra[caracter]
            add(L, current)
```

```
    #end of word
    if caracter == len(palabra)-1:
        current.isEndOfWord = True
        return

    #llamo a la funcion recursiva para el próximo caracter
    #mis parametros son: la prox lista, la palabra, el prox caracter y el current que será el padre
    insertR(current.children, palabra, caracter+1,current)

```

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.

```
78
79
80  def search(T,element):
81      #El árbol esta vacio
82      if T.root.children == None:
83          return False
84      else:
85          return searchR(T.root.children,element, 0,)
86
87  def searchR(L,element,caracter):
88      L = L.head
89      #busco el caracter
90      while L != None:
91          #si lo encuentro lo guardo en node y rompo el bucle
92          if L.value.key == element[caracter]:
93              node = L.value
94              break
95          L = L.nextNode
96      if L == None:
97          return False
98      #pregunto si el caracter que encontre es el ultimo de la palabra que busco
99      if caracter == (len(element)-1):
100         if node.isEndOfWord == True:
101             return True
102         else:
103             return False
104     else:
105         return searchR(node.children,element,caracter+1)
106
107
```

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).

O(m |Σ|) : m longitud de la palabra y |Σ| tamaño del abecedario

Si utilizo listas para crear estas estructuras, mi peor caso de la operación search() será de O(m |Σ|) (si ninguna cadena es sufijo de otra), ya que si o si tendré que recorrer todos los distintos elementos del vocabulario
Pero, si en lugar de utilizar listas, uso arreglos, los accesos serán en tiempo constantes, por lo que podremos reducir la complejidad temporal a O(m), pero vamos a ganar en complejidad espacial ya que mi arreglo sería de longitud 26 (por la cantidad de palabras en el abecedario español).

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.

```
121
122  def delete(T,element):
123      if T.root.children == None:
124          return False
125      else:
126          return deleteR(T.root.children,element, 0)
127
128  def deleteR(L,element,caracter):
129      current = L.head
130      while current != None:
131          if current.value.key == element[caracter]:
132              node = current.value
133              break
134          current = current.nextNode
135      if current == None: #no se encuentra la palabra (C1)
136          return False
137
138      if caracter == len(element)-1:
139      #El elemento está presente y es único ó tiene al menos un elemento incluido (C2y4).
140          if node.isEndOfWord == True and node.children.head == None:
141              return deleteCaso2y4(L,node) #(node = current.value)
142
143          #la palabra esta pero no tiene marcada la ultima letra como fin de palabra
144          elif node.isEndOfWord == False:
145              return False
146          #la palabra esta pero es prefijo de otra: desmarco el indicador de fin de palabra (C3)
147          elif node.isEndOfWord == True and node.children != None:
148              node.isEndOfWord = False
149              return True
150      else:
151          return deleteR(node.children,element,caracter+1)
152
```

```
155  #node.parent apunta a todo lo que tiene(children, key, etc)
156
157  def deleteCaso2y4(L,node): #node = L.value
158      newNode = node.parent
159      #pregunto si la lista tiene un elemento o mas:
160      if newNode.children.head.nextNode == None:
161          if newNode.children.head != None: #pregunto si no es la root
162          #tiene un solo elemento
163              deleteL(newNode.children, node)
164              deleteCaso2y4(L,newNode)
165          else: return True
166      else:
167          #tiene más de un elemento:
168          deleteL(newNode.children, node)
169      return True
170
171
```

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 buscoPatron(T,prefijo,n):
    long=len(prefijo)
    if T.root.children == None:
        return None
    else:
        current = T.root.children.head
        i = 0
        #Busco el patrón
        while current != None and i < long:
            if current.value.key == prefijo[i]:
                current = current.value.children.head
                i+=1
            elif i < long:
                current = current.nextNode
    if current == None:
        return None
    else:
        lista =LinkedList()
        buscoPalabras(0,current,prefijo,lista,n-1)
        return lista
```

```python
def buscoPalabras(cont,current,palabra,lista,n):
    if current == None:
        return lista

    if current.nextNode is None: #es hijo único:
        palabra += current.value.key
        cont+=1
        if cont == n and current.value.isEndOfWord == True: #si es un nodo hoja, agrego la palabra a la lista
            add(lista,palabra)
        elif cont == n and current.value.isEndOfWord != True:
            current.value.children.head = None
            cont = 0
        buscoPalabras(cont,current.value.children.head,palabra,lista,n) #paso a la siguiente lista

    else: #no es hijo unico
        palabra1 = palabra #guardo cadena hasta donde es hijo unico (ej, guardo "co": "corazon" y "como")
        while current is not None:
            palabra += current.value.key
            cont += 1
            if cont == n and current.value.isEndOfWord == True: #si es un fin de palabra, agrego la palabra a la lista
                add(lista,palabra)
            elif cont == n and current.value.isEndOfWord != True:
                current = current.value.children.head
                cont = 0
            buscoPalabras(cont,current.value.children.head,palabra,lista,n) #paso a la siguiente lista
            current = current.nextNode
            palabra = palabra1
            cont = len(palabra)-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.

Mi función arbolesIdenticos() tiene una complejidad de O(n^2) ya que dentro de un bucle while llamo a search la cual es de complejidad O(n). Además, usa una función recorrerTrieR() la cual tiene una complejidad de n^2 ya que dentro de un bucle while llamo a la función recursiva n veces.

```
227   #mi funcion recorrerTrieR tiene una complejidad de n^2 ya que dentro de un bucle while llamo a la funcion recursiva n veces.
228   def recorrerTrie(T1):
229       listaPalabras = LinkedList()
230       recorrerTrieR(T1.root.children.head,"",listaPalabras)
231       return listaPalabras
232
233   def recorrerTrieR(current,palabra,lista):
234       if current == None:
235           return lista
236
237       if current.nextNode is None: #es hijo único:
238           palabra += current.value.key
239           if current.value.isEndOfWord == True: #si es un nodo hoja, agrego la palabra a la lista
240               add(lista,palabra)
241           recorrerTrieR(current.value.children.head,palabra,lista) #paso a la siguiente lista
242
243       else: #no es hijo unico
244           palabra1 = palabra #guardo cadena hasta donde es hijo unico (ej, guardo "co": "corazon" y "como")
245           while current is not None:
246               palabra += current.value.key
247               if current.value.isEndOfWord == True: #si es un nodo hoja, agrego la palabra a la lista
248                   add(lista,palabra)
249               recorrerTrieR(current.value.children.head,palabra,lista) #paso a la siguiente lista
250               current = current.nextNode
251               palabra = palabra1
252
```

```
252
253   # arbolesIdenticos busca en T2 las palabras de T1 que anteriormente las agregue a una lista.
254   def arbolesIdenticos(T1,T2):
255       lista = recorrerTrie(T1)
256       print("Lista palabras T1:")
257       printLista(lista)
258       print("")
259       print("¿Árboles idénticos?")
260       current = lista.head
261       while current != None:
262           busco = search(T2,current.value)
263           if busco == False:
264               return False
265               break
266           else:
267               current = current.nextNode
268       return busco
```

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.

```
276  def cadenasInvertidas(T3):
277      listaPalabras = LinkedList()
278      palabraInvertida = False
279      return cadenasInvertidasR(T3,T3.root.children.head,"",listaPalabras,palabraInvertida)
280
281  def cadenasInvertidasR(T3,current,palabra,lista,palabraInvertida):
282      #CASOS BASE: deberia terminar de ejecutarse?????????
283      if palabraInvertida == True:
284          return print("True")
285
286      elif current == None:
287          return False
288
289      if current.nextNode is None: #es hijo único:
290          palabra += current.value.key
291          if current.value.isEndOfWord == True: #si es un nodo hoja, agrego la palabra a la lista
292              palabraInv = ""
293              m = len(palabra)-1
294              for i in range(0,len(palabra)):
295                  palabraInv += palabra[m-i]
296              palabraInvertida = search(T3,palabraInv)
297              if palabraInvertida != True:
298                  palabra = ""
299              else:
300                  print(palabraInv)
301                  print(palabra)
302
```

```
302
303          cadenasInvertidasR(T3,current.value.children.head,palabra,lista,palabraInvertida) #paso a la siguiente lista
304
305      else: #no es hijo unico
306          while current is not None:
307              palabra += current.value.key
308              if current.value.isEndOfWord == True: #si es un nodo hoja, agrego la palabra a la lista
309                  palabraInv = ""
310                  for i in range(len(palabra)-1,0):
311                      palabraInv += palabra[i]
312                  palabraInvertida = search(T3,palabraInv)
313                  if palabraInvertida != True:
314                      palabra = ""
315                  else:
316                      print(palabraInv)
317                      print(palabra)
318                      break
319              cadenasInvertidasR(T3,current.value.children.head,palabra,lista,palabraInvertida) #paso a la siguiente lista
320              current = current.nextNode
```

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"**.

```
329  def autoCompletar(T,prefijo):
330      long=len(prefijo)
331      if T.root.children == None:
332          return None
333      else:
334          current = T.root.children.head
335          i = 0
336          #Busco el patrón
337          while current != None and i < long:
338              if current.value.key == prefijo[i]:
339                  current = current.value.children.head
340                  i+=1
341              elif i < long:
342                  current = current.nextNode
343      if current == None:
344          return None
345
346      #completo palabra
347      palabra = ""
348      if current.nextNode != None:
349          return None
350
351      while current != None and current.nextNode == None: #mientras sea hijo unico
352          palabra += current.value.key
353          current = current.value.children.head
354
355      #si current == None entonces la palabra se completo totalmente.
356      if current == None:
357          return palabra
358
359      #si no es hijo unico corto y devuelvo hasta donde lo fue.
360      elif current.nextNode != None:
361          return palabra
```