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

```python
import linkedlist
import string


class Trie:
    root = None

    def __init__(self):
        self.root = TrieNode(None, None)

    def insert(self, word):
        currentNode = self.root
        for i, char in enumerate(word.upper()):
            #Árbol sin raíz (es decir, creamos un árbol desde 0)
            if self.root == None:
                self.root = TrieNode(None, None)
                childNode = TrieNode(self.root, char)
                linkedlist.add(self.root.children, childNode)
                currentNode = childNode
            #Rama sin hijos (creamos una nueva familia para el nodo)
            elif currentNode.children.head == None:
                childNode = TrieNode(currentNode, char)
                linkedlist.add(currentNode.children, childNode)
                currentNode = childNode
            #Rama con hijos (agregamos un hijo a la familia)
            else:
                isUnsorted = True
                position = 0
                childNodeAux = currentNode.children.head
                while isUnsorted:
                    if (childNodeAux == None):
                        childNode = TrieNode(currentNode, char)
                        linkedlist.insert(currentNode.children, childNode, position)
                        isUnsorted = False
                        currentNode = childNode
                    else:
                        if(char < childNodeAux.value.key):
                            childNode = TrieNode(currentNode, char)
                            linkedlist.insert(currentNode.children, childNode, position)
                            isUnsorted = False
                            currentNode = childNode
                        if(char == childNodeAux.value.key):
                            currentNode = childNodeAux.value
                            isUnsorted = False
                        if(char > childNodeAux.value.key):
                            position += 1
                            childNodeAux = childNodeAux.nextNode
            if i == len(word) - 1:
                currentNode.isEndOfWord = True

```

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.

```
51      def search(self, word):
52          currentNode = self.root
53          if (currentNode == None):
54              return None
55          currentNode = currentNode.children.head
56          charFoundCounter = 0
57          charFoundBool = False
58          for i, char in enumerate(word.upper()):
59              if currentNode.value.key == char:
60                  charFoundCounter += 1;
61                  charFoundBool = True
62              elif charFoundCounter != 0:
63                  charFoundCounter = 0
64                  charFoundBool = False
65              if(charFoundBool):
66                  currentNode = currentNode.value.children.head
67              else:
68                  while(currentNode != None and not charFoundBool):
69                      if(currentNode.value.key == char):
70                          charFoundCounter += 1;
71                          charFoundBool = True
72                      else:
73                          currentNode = currentNode.nextNode
74                  if(currentNode == None and charFoundBool == False):
75                      return False
76              if charFoundCounter == i + 1:
77                  return True
```

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).

En vez de que los hijos de cada nodo sean linkedlist, tendrían que ser arrays. Teniendo la desventaja de que, al ser una estructura estática (no se pueden agregar ni quitar espacios del que se definió al inicializar), tenemos que incluir todas las letras del alfabeto para cada hijo de cada nodo.

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.

```
78
79      def delete(self, word):
80          currentNode = self.root
81          currentNode = currentNode.children.head
82          #Caso 1: árbol vacío
83          if currentNode == None:
84              return False
85          fatherNode = searchInBrothers(currentNode, word[0])
86          #Caso 2: la palabra no se encuentra en el árbol
87          if fatherNode == None:
88              return False
89
90          currentNode = fatherNode
91          wordsFound = 0
92          for i, char in enumerate(word.upper()):
93              currentNode = searchInBrothers(currentNode, char)
94              if(currentNode == None):
95                  return False
96
97
98              #Se encontró un nodo que es final de palabra (puede ser de la que se quiere eliminar u otra)
99              if currentNode.value.isEndOfWord:
100                 wordsFound += 1
101                 if(i < len(word) - 1):
102                     lastWordNode = currentNode
103                     lastWord = word[0:i+1]
104             #Llegamos al final de palabra
105             if(i == len(word) - 1):
106                 #En el recorrido hasta la última letra, sólo se encontró una palabra
107                 if(wordsFound == 1 and currentNode.value.children.head == None):
108                     currentNode = currentNode.value.parent
109                     while(currentNode != None):
110                         if(linkedlist.length(currentNode.children) == 1):
111                             currentNode.children = None
112                         else:
113                             linkedlist.delete(currentNode.children, searchInBrothers(currentNode.children.head, word[i].upper()).value)
114                             return True
115                         currentNode = currentNode.parent
116                         i -= 1
117
118
119                     if(currentNode.value.children.head == None):
120                         fatherNode = None
121                     else:
122                         currentNode.value.isEndOfWord = False
123                 else:
124                     currentNode.value.isEndOfWord = False
125                     return True
126             currentNode = currentNode.value.children.head
127         return False
```

```
162 def searchInBrothers(currentNode, char):
163     while currentNode != None:
164             if currentNode.value.key == char.upper():
165                 return currentNode
166             currentNode = currentNode.nextNode
167     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**.

```
231  def searchPattern(T, pattern, length):
232      words = findWords(T, pattern)
233      currentNode = words.head
234      wordsOfInterest = linkedlist.LinkedList()
235      while currentNode != None:
236
237          currentNode.value = pattern.upper() + currentNode.value
238          k = 0
239          for i in enumerate(currentNode.value):
240              k += 1
241          if(k == length):
242              linkedlist.add(wordsOfInterest, currentNode.value)
243          currentNode = currentNode.nextNode
244
245      return wordsOfInterest
246
```

```
183  def findWords(T, pattern):
184      listaPalabras = linkedlist.LinkedList()
185      currentNode = T.root.children.head
186      palabra = ""
187      for i, char in enumerate(pattern):
188          currentNode = searchInBrothers(currentNode, char.upper())
189          if currentNode != None:
190              if(i == len(pattern)-1):
191                  if currentNode.value.key == char.upper():
192                      break
193              currentNode = currentNode.value.children.head
194          else:
195              return None
196
197
198      findWordsRec(currentNode, listaPalabras, palabra)
199      return listaPalabras
200
```

```
220  def findWordsRec(TrieNode, palabras, palabra):
221      currentNode = TrieNode.value.children.head
222      while currentNode != None:
223          letra = currentNode.value.key
224          palabra = palabra + letra
225          if(currentNode.value.isEndOfWord):
226              linkedlist.add(palabras, palabra)
227          findWordsRec(currentNode, palabras, palabra)
228          palabra = palabra[0:len(palabra)-1]
229          currentNode = currentNode.nextNode
230
```

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.

```
248  def areEqual(T1, T2):
249      wordList1 = findAllWords(T1)
250      wordList2 = findAllWords(T2)
251
252      currentNode1 = wordList1.head
253      currentNode2 = wordList2.head
254
255      while currentNode1 != None and currentNode2 != None:
256          if(currentNode1.value != currentNode2.value):
257              return False
258          currentNode1 = currentNode1.nextNode
259          currentNode2 = currentNode2.nextNode
260
261      return True
```

```
201  def findAllWords(T):
202      listaPalabras = linkedlist.LinkedList()
203      currentNode = T.root.children.head
204      palabra = ""
205      findAllWordsRecursive(currentNode, listaPalabras, palabra)
206      return listaPalabras
207
208  def findAllWordsRecursive(currentNode, palabras, palabra):
209      if(currentNode == None):
210          return
211      while currentNode != None:
212          letra = currentNode.value.key
213          palabra = palabra + letra
214          if(currentNode.value.isEndOfWord):
215              linkedlist.add(palabras, palabra)
216          findAllWordsRecursive(currentNode.value.children.head, palabras, palabra)
217          palabra = palabra[0:len(palabra)-1]
218          currentNode = currentNode.nextNode
219
```

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.

```
263  def findReverseWord(T):
264      wordList = findAllWords(T)
265      currentNode = wordList.head
266      currentNodeAux = wordList.head.nextNode
267
268      while currentNodeAux != None and currentNode != None:
269          if(currentNode.value == currentNodeAux.value[::-1]):
270              return True
271          currentNodeAux = currentNodeAux.nextNode
272          if(currentNodeAux == None):
273              currentNode = currentNode.nextNode
274              currentNodeAux = currentNode
275      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"**.

```
183  def findWords(T, pattern):
184      listaPalabras = linkedlist.LinkedList()
185      currentNode = T.root.children.head
186      palabra = ""
187      for i, char in enumerate(pattern):
188          currentNode = searchInBrothers(currentNode, char.upper())
189          if currentNode != None:
190              if(i == len(pattern)-1):
191                  if currentNode.value.key == char.upper():
192                      break
193              currentNode = currentNode.value.children.head
194          else:
195              return None
196
197
198      findWordsRec(currentNode, listaPalabras, palabra)
199      return listaPalabras
200
```