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

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
def insert(T, element):

    if T.root is None:
        node = TrieNode()
        T.root = node

    current = T.root.children       ## Elijo el nodo "Head" de la lista, si es que existe
    Flag = False

    if current is None:
        new_node = TrieNode()
        new_node.parent = T.root
        T.root.children = new_node
        current = new_node

    for i in range(len(element)):

        if current.key == None:
            current.key = element[i]
            Flag = True

        else:
            if SearchL(current, element[i]) == None and Flag == False:

                new_node = TrieNode()       ## Se complica al insertar mas de dos palablar porque empiezan a pisar el campo nextNode y no se conectan de forma efectiva
                new_node.nextNode = current.nextNode        ## Al agregar esta linea logre insertar mas de 2 palabras a la primera fila
                current.nextNode = new_node
                new_node.key = element[i]           ## Me falta asignar bien los parents aca
                Flag = True
                # Experimental
                new_node.parent = current.parent
                ##
                current = new_node

            elif SearchL(current, element[i]) != None:
                current = SearchL(current, element[i])       ## Debo buscar otra manera de comparar el elemento ya que si el search es "None", tira error
                ## Al insertar la palabra mamarracho falla por la iteracion numero 5, ya que al repetir el valor del nodo, falla en...
                if current.children != None:  ## ... Esta parte

                    current = current.children      ## Logre Insertar una parabla que tiene como prefijo otra, creo que se puede hacer mas eficiente
                else:
                    Flag = True ## Con Esta linea logre agregar palabras mas largas
                    continue

            else:

                new_node = TrieNode()       ## Codigo parecido al del inicio la diferencia es que en la primera iteracion se va a conectar a la raiz
                new_node.parent = current
                current.children = new_node
                new_node.key = element[i]
                current = new_node

            if element[i] == element[len(element)-1] and i == len(element)-1:       ## Verifico si es el ultimo elemento de la palabra
                if current.key != element[i]:
                    current.parent.isEndOfWord = True
                else:
                    current.isEndOfWord = True

def SearchL(Node, element):

    while Node != None:
        if Node.key == element:
            return Node
        else:
            Node = Node.nextNode
    return None

def search(T, element):

    current = T.root.children
    return searchR(current, element)

def searchR(current, element):

    if current == None:
        return False

    pos = SearchL(current, element[0])
    aux = pos != None

    if aux == False:
        return False

    if len(element) == 1 and pos.isEndOfWord == True:
        return True
    element = element[1:]

    return searchR(pos.children, element)
```

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).

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.

```python
def delete(T, element):

    current = T.root.children

    if search(T, element) == False:
        return False

    EndWord = toEndofWord(current, element)


    # Funciona Caso 0: Palabra dentro de palabra

    if current.children != None and current.isEndOfWord is True:
        current.isEndOfWord = False
        return True

    # Funciona Caso 1: Nodo final solitario

    if current.children is None and current.isEndOfWord is True:
        if current.nextNode is None:

            current.parent.children = None
        elif current.nextNode is not None:
            aux = current
            current.parent.children = aux.nextNode
```

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**.

```
151  def toEndofWord(List, element):
152
153      current = SearchL(List, element[0])
154
155      if current != None and current.children != None:
156          current = current.children
157      if current.key == element[len(element)-1] and current.isEndOfWord == True:
158          return current
159      element = element[1:]
160      return toEndofWord(current, element)
161
162
163  def PrintChain(T, p, n):
164
165      if T.root == None:
166          return None
167
168      last_node = toEndofPattern(T.root.children, p)
169
170      if last_node == None:
171          return None
172
173      n = n - len(p)
174      ListaPalabras = []
175      current = last_node
176      aux = n
```

```
176      aux = n
177
178
179      ## He posicionado mal los marcadores, imaginate que te estas moviendo por algo parecido a una matriz (No es una matriz, ojo)
180      ## Tengo que cambiarlos por algo mas facil de manejar que un while, intentar con un "for"
181      ## El caso que se me complica es revisar los nodos en forma de "Escalera"
182      while n > 0:
183          while aux > 0:
184              if current.nextNode != None:
185                  current = current.children
186
187
188              aux -= 1
189          CheckEnd(last_node, ListaPalabras)
190
191          if last_node.children != None:
192              last_node = last_node.children
193              current = last_node
194
195          n -= 1
196      CheckEnd(current, ListaPalabras)
197
198      #CheckEnd(last_node, ListaPalabras)
199      return ListaPalabras
200
```

```python
def CheckEnd(List, ListaPalabras):      ## Checkea que alguno de los elementos de la lista tenga EndOfWord = True
    if List == None:
        return
    if List.isEndOfWord == True:
        aux = ''
        palabra = AlmacenarPalabra(List, aux)
        ListaPalabras.append(palabra)

    CheckEnd(List.nextNode, ListaPalabras)
    return ListaPalabras

def AlmacenarPalabra(List, aux):        ## Almacena una palabra accediendo a los keys de los parents

    if List.key == None:
        return aux
    aux = List.key + aux
    return AlmacenarPalabra(List.parent, aux)

def toEndofPattern(List, element):

    current = SearchL(List, element[0])

    if current != None:
        current = current.children
    else:
        return None
    if current.key == element[len(element)-1]:
        return current
    element = element[1:]
    return toEndofPattern(current, element)
```

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"**.