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):
    longitud = len(element)
    position = 0
    return insertR(T,element,longitud,position)

def insertR(T,element,Longitud,position):
    #Caso base
    if position == longitud:
        currentNode = T.children.head
        while currentNode != None:
            if currentNode.value.key == element[position-1]:
                currentNode.value.parent = T.children
                currentNode.value.isEndOfWord = True
                break
            currentNode = currentNode.nextNode
        return
    #1er paso
    if T.children != None:
        currentNode = T.children.head
    else:
        T.children = LinkedList()
        newTnode = TrieNode()
        newTnode.key = element[position]
        add(T.children, newTnode)
        currentNode = T.children.head
    #2do paso
    while currentNode != None:
        if currentNode.value.key == element[position]:
            if longitud - position > 1:
                currentNode.value.parent = T.children
                T = currentNode.value
master*  ⊗ 0 ⚠ 0
```

```
            if longitud - position > 1:
                currentNode.value.parent = T.children
                T = currentNode.value
            else:
                currentNode.value.parent = T.children
            break
        currentNode = currentNode.nextNode
    if currentNode == None:
        newTnode = TrieNode()
        newTnode.key = element[position]
        add(T.children, newTnode)
        T.children.head.value.parent = T
        T = T.children.head.value
    return insertR(T,element,longitud,position+1)
"""-------------------------------------------------------------
```

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 search(T,element):
    if T.children == None:
        return False
    longitud = len(element)
    position = 0
    contadorLetras = 0
    return searchR(T,element,longitud,position,contadorLetras)

def searchR(T,element,longitud,position,contadorLetras):
    if contadorLetras == longitud:
        return True
    else:
        currentNode = T.children.head
        while currentNode != None:
            if currentNode.value.key == element[position]:
                contadorLetras += 1
                T = currentNode.value
                break
            currentNode = currentNode.nextNode
        if currentNode == None:
            return False
        else:
            return searchR(T,element,longitud,position+1,contadorLetras)
```

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).

Una versión de la operación search() donde la complejidad sería de O(m) es usar listas o arreglos de python del 0 al 26 donde el alfabeto se encuentre en la lista/array en orden. De esta manera accederemos a las letras de la palabra de una manera constante ya que sabríamos en qué posición se encuentran las letras.

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.

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**.

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 **ab<u>c</u>d** y **dc<u>k</u>a** 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"**.