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.

```
        #Funcion que llamamos para insertar un caracter en los hijos de un nodo
        def insert_child(node, value):
            find = -1
            for i in range(len(node.children)):
                if node.children[i].key == value:
                    find = i

            if find != -1:
                return node.children[i]
            else:
                newNode = TrieNode()
                newNode.key = value
                newNode.children = []
                newNode.parent = node
                node.children.append(newNode)
                return newNode


        def insert(T,element):
            #Ponemos una raiz en caso de que nuestro trie no tenga una
            if T.root is None:
                rootNode = TrieNode()
                rootNode.children = []
                T.root = rootNode

            #Recorremos los caracteres del string y llamamos a la funcion que inserta los nodos
            current = T.root
            for character in element:
                current = insert_child(current, character)

            #Declaramos el ultimo nodo como endOfWord
            current.isEndOfWord = True

        #Comprobamos si un caracter esta en los hijos de nuestro nodo
        #Devuelve el hijo si lo encuentra, y false si no
        def search_node(node, value):
            for child in node.children:
                if child.key == value:
                    return child

            return None

        #Buscamos si un string esta en nuestro arbol
        def search(T,element):
            if T.root is None:
                return False
            else:
                current = T.root
                 #Para cada elemento, actualizamos el caracter y buscamos coincidencias en los hijos de
un nodo
                    #Si encontramos coincidencias volvemos a llamar a la funcion con el "nodo
coincidencia"
                  #Al terminar, devuelve el isEndOfWord del nodo actual, si la palabra se encuentra
```

```
deberia devolver true
            for character in element:
                current = search_node(current, character)

                if current == False:
                    return False

            return current.isEndOfWord
```

Ejercicio 2 (no code)

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).

Primero recordemos: |Σ| representa el tamaño de nuestro alfabeto. Si queremos reducir el orden a O(m), tendriamos que poder acceder a los hijos en O(1).
Para tener una operacion search() cuya complejidad sea O(m), deberiamos cambiar la estructura en la que almacenamos los hijos para poder acceder a estos en O(1); en nuestro caso, un **diccionario o hash table** nos permitiria reemplazar la linkedlist o lista que estamos usando para guardar **children**.

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.

```
#Borramos un string de nuestro trie
def delete(T,element):
   #Desvinculamos un child de su parent
   def delete_node(node):
       parent = node.parent
       parent.children.remove(node)
       return parent

   #Chequeamos que el trie exista
   if T.root is None:
       return False

   #Chequeamos que la palabra este dentro del arbol
   if not search(T,element):
       return False

   else:
```

```
        #Vamos hacia el ultimo nodo del string
        current = T.root
        for character in element:
            current = search_node(current, character)

        #Si el ultimo nodo tiene hijos, solo desmarcamos isEndOfWord y devolvemos true
        if current.children:
            current.isEndOfWord = False
            return True

        #Borramos los nodos parent si no tienen mas hijos hasta llegar a la raiz del trie
        while current != T.root:
            current = delete_node(current)

            if current.children or current.isEndOfWord:
                break

        return True
```

Parte 2

Ejercicio 4

Implementar un algoritmo que dado un árbol **Trie T**, un patrón **p (prefijo)** y un entero **n**, escriba todas las palabras del árbol que empiezan por **p** y sean de longitud **n**.

Primero deberiamos revisar que el **trie** tenga elementos y que el **prefijo** exista dentro de nuestro arbol (usando search()). Luego pasamos a buscar las palabras que tengan este prefijo.
Luego deberiamos ir armando strings con los **key** de los nodo que vamos “explorando”, y solo deberiamos tomar en cuenta las palabras que cumplan lo siguiente:
len(palabra) == n y **isEndOfWord == True**

```
#Buscamos todas las palabras que empiecen con un prefijo p y tengan longitud n
def search_with_p(T, p, n):
    #Chequeamos que el trie no esté vacío
    if T.root is None:
        return result

    #Chequeamos que el prefijo exista en el trie
    if not search(T, p):
        return False

    result = []

    #Funcion recursiva que nos permite armar strings para encontrar palabras con las condiciones
necesarias
    def recursive_search(node, palabra_actual):
        #Si la longitud supera n, no seguimos buscando hijos
```

```
        if len(palabra_actual) > n:
            return

        if len(palabra_actual) == n and node.isEndOfWord:
            result.append(palabra_actual)
            return

        #Exploramos los hijos
        for child in node.children:
            recursive_search(child, palabra_actual + child.key)

    #Nos desplazamos hasta el ultimo nodo del prefijo
    current = T.root
    for character in p:
        current = search_node(current, character)

    recursive_search(current, p)
    return result
```

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** pertenece 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.

```
#Busca y devuelve las palabras que hay en un Trie

def words_in_trie(T):

    result = []



    def word_recursive(palabra_actual, node):

        #No agregamos el key de la raiz porque no tiene

        if node != T.root:

            palabra_actual += node.key
```

```
        #Si llegamos al final de la palabra, la agregamos a la lista
        if node.isEndOfWord:
            result.append(palabra_actual)

        #Llamamos recursivamente desde cada uno de ellos
        for child in node.children:
            word_recursive(palabra_actual, child)

        return

    word_recursive("",T.root)
    return result

#Funcion que compara si tenemos los mismos strings en dos tries diferentes
def same_document_tries(T1,T2):
    #Obtenemos las listas de palabras de cada arbol
    words1 = words_in_trie(T1)
    words2 = words_in_trie(T2)

    #Comparamos si cada palabra del primer trie existe en el segundo trie
    #Devolvemos true o false si corresponde
    for child_of_1 in words1:
        flag_find = False
        for child_of_2 in words2:

            if child_of_1 == child_of_2:
                flag_find = True
```

```
        if not flag_find:
            return False

    return True
```

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 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 "" (cadena vacia) si **T** presenta las cadenas **"madera"** y **"mama"**.

```
#Dado un string ingresado, buscamos "fillers" o palabras que completen nuestro string

#El string que completa nuestra palabra tiene un "camino unico" (no tienen dos o mas hijos, solo uno)

def complete_word(T, string):

    #Nos desplazamos hasta el ultimo caracter del string ingresado

    current = T.root

    for character in string:

        current = search_node(current,character)



    #Analizamos si cada nodo a partir del ultimo del string tiene un solo hijo y vamos armando nuestro
filler
```

```
    def recursive_fill(palabra_actual,node):

        if len(node.children) == 1:

            node = node.children[0]

            palabra_actual += node.key

            return recursive_fill(palabra_actual,node)

        else:

            return palabra_actual




    result = recursive_fill("",current)

    return result
```