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) el valor del elemento (palabra) a agregar.
Salida: No hay salida definida

```
def insert(T, string):
    if T.root is None:
        newNode = TrieNode()
        #print("Insertado",string[0])
        newNode.key = string[0]
        newNode.children = [None,None]
        T.root = newNode
        if len(string) == 1:
            T.root.isEndOfWord = True
    add_trie(T.root, string)

def add_trie(current,string):
    if current is not None:
        if current.key == string[0]:
            string=string[1:]
            if not string:
                current.isEndOfWord = True
                return
            elif current.children[0] != None:
                #print("Hijo de",current.key)
```

```
                return add_trie(current.children[0],string)
            else:
                newNode = TrieNode()
                newNode.parent=current
                #print("Insertado como Hijo",string[0])
                newNode.key = string[0]
                newNode.children = [None,None]
                current.children[0] = newNode
                if len(string) == 1:
                    newNode.isEndOfWord = True
            return add_trie(current.children[0],string)
        if current.children[1] != None:
            #print(current.key,"Hermano de",current.children[1].key)
            return add_trie(current.children[1],string)
        else:
            newNode = TrieNode()
            newNode.parent=current
            #print("Insertado como hermano",string[0])
            newNode.key = string[0]
            newNode.children = [None,None]
            current.children[1] = newNode
            if len(string) == 1:
                newNode.isEndOfWord = True
        #print(current.key,"Hermano de",current.children[1].key)
        return add_trie(current.children[1],string)
```

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) el valor del elemento (palabra)
Salida: Devuelve **False o True** según se encuentre el elemento.

```
def find(current,string):
    if current is None :
        if len(string) > 0:
            return False
        return True
    elif len(string) == 0:
        if current.isEndOfWord:
            return True
        return False

    if current.key == string[0]:
        #print("Buscar hijo",current.key)
        return find(current.children[0],string[1:])
    else:
        #print("Buscar hermano",current.key)
        return find(current.children[1],string)

def search(T,string):
    if T.root != None:
        return find(T.root,string)
    return False
```

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).

Si implementamos para las primeros caracteres de las palabras un hash table entonces tendríamos una key de la palabra que nos permitiría acceder al primer carácter de la palabra que buscamos dando ha si O(1) + O(m) siendo este la complejidad de recorrer la palabra de tamaño 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.

```
def delete_node(current):
    if current!=None:
        #print(current.key)
        if current.isEndOfWord == False :
            if current.children[1] is None:
                #print("elimina",current.children[0].key)
                current.children[0]=None
        if current.parent is not None:
            if current.parent.children[1] is not None and
current.parent.children[0] is not None:
                if current.key == current.parent.children[1].key:
                    current=current.parent
```

```
                    #print("elimina 1 de ",current.key)
                    current.children[1]=None
                    return
                if current.key == current.parent.children[0].key:
                    current=current.parent
                    if current.parent is not None:
                        aux=current.parent
                        if current.key == aux.children[0].key:
                            aux.children[0]=current.children[1]
                        elif current.key == aux.children[1].key:
                            aux.children[1]=current.children[1]
                    return
            #print("SUBE")
            return delete_node(current.parent)

def delete_word(current,element):
    if current is None and len(element) > 0:
        return False
    elif current.key == element[0] and len(element) == 1:
        delete_node(current)
        return True
    if current.key == element[0]:
        return delete_word(current.children[0],element[1:])
    else:
        return delete_word(current.children[1],element)

def delete(T,element):
    Flag=False
    if T.root != None and len(element)>0:
        Flag = delete_word(T.root,element)
        if T.root.key == element[0] and Flag==True:
            if T.root.children==[None,None] is None:
                T.root=None
            elif T.root.children[0] is None and T.root.children[1] is not None :
                T.root=T.root.children[1]
    return Flag
```

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**.

```
def collect_prefj(node,prfj,n, prefix, words):
    if node is None:
        return
    # Agregar la clave actual al prefijo
    new_prefix = prefix + node.key if node.key else prefix
    # Si es el final de una palabra, agregar al resultado
    if node.isEndOfWord:
        if new_prefix[0] == prfj[0] and len(new_prefix) == n:
            words.append(new_prefix)
    # Recorrer ambos hijos: [0] es el hijo directo y [1] es el hermano
    collect_prefj(node.children[0],prfj,n, new_prefix, words)
    collect_prefj(node.children[1],prfj,n, prefix, words)  # note que se usa el prefijo sin añadir la clave
actual

def prefj(T,prfj,n):
    words = []
    collect_prefj(T.root,prfj,n,"", words)
    return words

print("Ejercio_4------------------------")
B=prefj(T,"j",4)
print("prefijo de j:",B)
```

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.

```
def is_subset(T2,L1):
    for i in L1:
        if not search(T2,i) :
            return False
    return True

def compare_tries(T1, T2):
    if T1.root is not None and T2.root is not None:
        #Compara dos Tries y determina si son iguales o si uno es subconjunto del otro.
        List1=get_all_words(T1)
        if len(List1)>0 or T1.root is not None:
            return is_subset(T2,List1)
    return False
```

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.

```
def generate_reverse(word):
    return word[::-1] #Funcional que invierte una lista

def search_inverted(T,words):
    for list in words:
        l1=generate_reverse(list)
        if search(T,l1):
            return True
    return False

def is_inverted_pair(T):
    if T.root is None:
        return False
    list = get_all_words(T)
    return search_inverted(T,list)

print("Ejercio_6-----------------------")
insert(T1,"afar")
all_words = get_all_words(T1)
print("Trie 1:",all_words)
print(is_inverted_pair(T1))
```

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"**.

```
def completar(current,element,words):
    if current is not None:
        if len(element) > 0:
            if current.key==element[0]:
                #print("0",current.key)
                return completar(current.children[0],element[1:],words)
            else:
                #print("1",current.key)
                return completar(current.children[1],element,words)
        else:
            #print("estoy en ",current.key)
            if current.isEndOfWord != False:
                if current.children[0] is None and current.children[1] is None:
```

```
                new_words = words + current.key
                return new_words
            return
        else:
            new_words = words + current.key if current.key else words
            #print(new_words)
            if current.children[0] is not None and current.children[1] is None:
                #print("entra 0")
                return completar(current.children[0],element,new_words)
            elif current.children[0] is None and current.children[1] is not None:
                #print("entra 1")
                return completar(current.children[1],element,new_words)
    return words

def autoCompletar(T,cadena):
    if T.root!= None:
        return completar(T.root,cadena,"")
    return ""

print("Ejercio_7------------------------")
A=autoCompletar(T,"ju")
print("ju:",A)
```