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):
  if T.root == None:
    T.root = TrieNode()
    T.root.children = LinkedList()
    insertR(T.root.children,None,element,0)
  else:
    insertR(T.root.children,T.root,element,0)

def insertR(lista,parent,element,i):
  if i >= len(element):
    parent.isEndOfWord = True
  else:
    ch = element[i]
    nodoLista = searchElementInList(lista,ch)

    if nodoLista == None:
      newTrieNode = TrieNode(parent,LinkedList(),ch,False)
      add(lista,newTrieNode)
      insertR(newTrieNode.children,newTrieNode,element,i+1)
    else:
      node = nodoLista.value
      #node es el trieNode que se almacena en el campo value del
nodo de la lista
      insertR(node.children,node,element,i+1)
```

```
def searchElementInList(L,ch):
  current = L.head
  while current != None:
    if current.value.key == ch:
      return current
    current = current.nextNode
  return
```

`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.

```
def search(T,element):
  if T.root == None or T.root.children == None:
    return False
  else:
    #(lista,nodo,string,strIndex)
    #print("b: ", searchR(T.root.children,element,0))
    return searchR(T.root.children,element,0)

def searchR(list,element,i):
  if i >= len(element):
      return
  else:
      ch = element[i]
      node = searchElementInList(list,ch)
      if node == None:
        return False
      else:
        trieNode = node.value
        if i == len(element)-1:
          return trieNode.isEndOfWord
        else:
          return searchR(trieNode.children,element,i+1)
```

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).

Si usamos arreglos en lugar de una lista enlazada no tendremos que recorrer en el peor caso las 26 posiciones para encontrar el carácter que buscamos sino que al arreglo nos permite acceder en O(1) a dicho carácter por lo tanto la complejidad temporal sera de O(m), m = tamaño de la palabra

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(T,element):
  if search(T,element) == True:
    return deleteR(T,T.root.children,element,0)
  else:
    return False
```

```
def deleteR(T,list,element,i):
  ch = element[i]
  node = searchElementInList(list,ch)
  trieNode = node.value
  if i == len(element)-1:
    if trieNode.children.head == None:
      deleteNode(list,ch)
      if length(list) >= 1:
        return True
      else:
          while trieNode.parent.parent != None and not
trieNode.parent.isEndOfWord:

deleteNode(trieNode.parent.parent.children,trieNode.parent.k
ey)
            trieNode = trieNode.parent
          deleteNode(T.root.children,trieNode.parent.key)
          return True
    else:
      trieNode.isEndOfWord = False
      return True
  else:
    return deleteR(T,trieNode.children,element,i+1)
```

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**.

```
def imprimePalabras(T,p,n):
    trieNode = searchPrefijo(T,p)
    long = n-len(p)
    if trieNode == None or len(p) > n:
        return None
    else:
        imprimePalabrasR(trieNode.children.head,p,long,0)

def imprimePalabrasR(node,string,long,cont):
    if node == None:
        return
    else:
        if cont <= long-1:
            if node.nextNode == None:
                if cont == long-1:
                    if node.value.isEndOfWord == True:
                        string += node.value.key
                        print(string)
                        return
                else:
                    if node.value.isEndOfWord != True:
                        string += node.value.key
                    imprimePalabrasR(node.value.children.head,string,long,cont+1)
            else: #si la lista tiene mas de un nodo
                str = string
                while node != None:
                    if cont == long-1:
                        if node.value.isEndOfWord == True:
                            string += node.value.key
                            print(string)
                    else:
                        if node.value.isEndOfWord != True:
                            string += node.value.key
                        imprimePalabrasR(node.value.children.head,string,long,cont+1)
                    node = node.nextNode
                    string = str
```

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.

```python
def sonIguales(T1,T2):
    lista = recorreTrie(T1)
    #printList1(lista)
    current = lista.head
    while current != None:
        if search(T2,current.value) == False:
            print("Los trie no son iguales")
            return False
        current = current.nextNode
    print("Los trie son iguales")
    return True

def recorreTrie(T1):
    lista = LinkedList()
    recorreTrieR(T1.root.children.head,"",lista)
    return lista

def recorreTrieR(node,palabra,lista):
    if node == None: #llego al final del trie
        return lista

    if node.nextNode == None: #solo tiene un hijo (la lista solo tiene un nodo)
        palabra += node.value.key
        if node.value.isEndOfWord == True:
            add(lista,palabra)
        recorreTrieR(node.value.children.head,palabra,lista)
    else: #la lista tiene mas de un nodo
        string = palabra #guardo el prefijo que comparten las palabras
        while node != None:
            palabra += node.value.key
            if node.value.isEndOfWord == True:
                add(lista,palabra)
            recorreTrieR(node.value.children.head,palabra,lista)
            node = node.nextNode
            palabra = string
```

Python

costo computacional: O(n^2) ya que dentro de un bucle while (O(n)) llamo a la función search que es de complejidad O(n)

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 cadInvertidas(T):
  lista = recorreTrie(T)
  current = lista.head
  while current != None:
    if search(T,invierteCadena(current.value)) == True:
      print("hay cadenas invertidas", current.value)
      return True
    current = current.nextNode
  print("no hay cadenas invertidas")
  return False


def invierteCadena(cadena):
  palabra = ""
  for i in range(len(cadena)-1,-1,-1):
    palabra += cadena[i]
  return palabra
```

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"**.

```
def autocompletar(T,cadena):
  node = searchPrefijo(T,cadena) #me devuelve el nodo.key = ultimo caracter de la cadena
  if node == None:
    return "" #cadena vacia
  else:

    return autocompletarR(node.children.head,"")

def autocompletarR(node,palabra):
  if node == None: #llego al final del trie
    return palabra
  if node.nextNode == None: #la lista solo tiene un nodo
    palabra += node.value.key
    return autocompletarR(node.value.children.head,palabra)
  else:
    return palabra
```