Ejercicio 1

Crear un modulo 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 insertTrie(T, element):
   if T.root == None:
       T.root = TrieNode()
       T.root.isEndOfWord=False
       T.root.key=""
       T.root.parent=None
   insertTrieR(T,T.root,element,0)

def insertTrieR(T,node,element,i):
   if i == len(element):
       node.isEndOfWord=True
       return
   currentNode = node.children
   if currentNode == None:
      return insertOnlyDown(node,element,i)
   while currentNode != None:
       if currentNode.key == element[i]:
           return insertTrieR(T,currentNode,element,i+1)
       if currentNode.nextNode ==None:
           newNode = createTrieNode(element[i])
           currentNode.nextNode =newNode
           newNode.parent = currentNode.parent
           return insertOnlyDown(newNode,element,i+1)
       currentNode=currentNode.nextNode
```

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 SearchTrie(T,element):
```

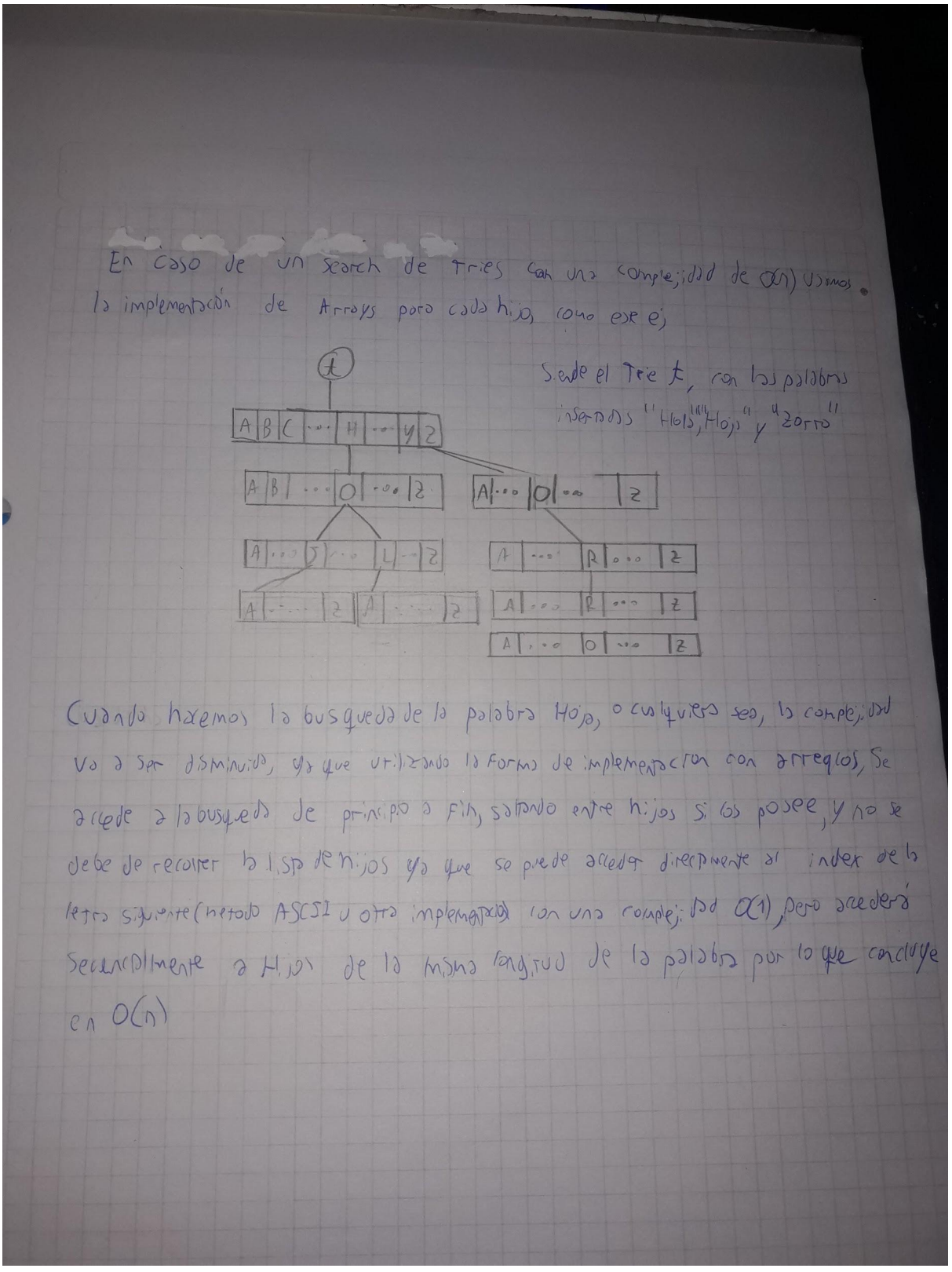
```
      if T.root==None:
            return None
      else:
            return SearchTrieR(T,T.root.children,element,0)

def SearchTrieR(T,node,element,c):
      if node.nextNode==None:
            if node.key!=element[c]:
                  return False
            else:
                  if len(element)-1==c:
                        if node.isEndOfWord:
                              return True
                        else:
                              return False
                  else:
                        if node.children==None:
                              return False
                        else:
                              return
SearchTrieR(T,node.children,element,c+1)

      else:
            if node.key!=element[c]:
                  return SearchTrieR(T,node.nextNode,element,c)
            else:
                  if len(element)-1==c:
                        if node.isEndOfWord:
                              return True
                        else:
                              return False
                  else:
                        if node.children==None:
                              return False
                        else:
                              return
SearchTrieR(T,node.children,element,c+1)
```

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(n).



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 T.root==None:
        return None
    else:
        node=SearchNode(T,element)
        if SearchTrie(T,element):
            return deleteR(T,node,element,0)
        else:
            return False
def deleteR(T,node,element,c):
    parentN=node.parent
    if node.isEndOfWord and node.children!=None:
        node.isEndOfWord=False
        return True
    if parentN==T.root or parentN.isEndOfWord or parentN.children!=node or node.nextNode!=None:
        linksDeleteR(node)
        return True
    linksDeleteR(node)
    return deleteR(T,parentN,element,c)

def linksDeleteR(node):
        if node.parent.children==node:
            if node.nextNode!=None:
                node.parent.children=node.nextNode
                node.nextNode=None
            else:
                node.parent.children=None
        else:
                nodoaux=node.parent.children
                while nodoaux.nextNode!=None:
                    if nodoaux.nextNode==node:
                        nodoaux.nextNode=node.nextNode
                        node.nextNode=None
                        break
                    nodoaux=nodoaux.nextNode

        node.parent=None
```

```
                node.isEndOfWord=None
```

Parte 2

Ejercicio 4

Escribir una algoritmo que dado un árbol **Trie T**, una palabra **p** y un entero **n**, escriba todas las palabras del árbol que empiezan por **p** y sean de longitud **n**.

```
def searchPtoN(T,p,n):
    if T==None or n<1 or (len(p)>n):
        print(len(p))
        return False
    L=LinkedList()
    searchPtoNRec(T,T.root.children,p,n,p,False,L)
    return L


def searchPtoNRec(T,node,p,n,con,i,L):
    cont=0
    nodea=node
    while nodea.children!=None:
        if len(p)==0:
            i=True
            break
        elif cont==len(p):

            i=True
            break
        elif p[cont]==nodea.key:
            cont+=1
            nodea=nodea.children
        elif p[cont]!=nodea.key:
            if nodea.nextNode==None:
                break
            else:

searchPtoNRec(T,nodea.nextNode,substr(p,cont,len(p)),n-cont,con,False,L)
                break

    if i:
        countsonsPtoN(T,nodea,con,n-1,cont,L)
```

```
def countsonsPtoN(T,node,p,n,cont,L):
    conta=cont
    while node!=None:
        if node.nextNode!=None:
            countsonsPtoN(T,node.nextNode,p,n,conta,L)
        p=p+node.key
        if (conta<n):
            conta+=1
            node=node.children
        elif conta==n:
            if node.isEndOfWord:
                add(L,p)
                return
            else:
                return
        else:
            return
```

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 2 Trie pertenecen al mismo documento cuando:

1. Ambos Trie sean iguales
2. El Trie A contiene un subconjunto de las palabras del Trie B.
3. Si la implementación está basada en LinkedList, considerar el caso donde las hayan sido insertadas en un orden diferente.

Analizar el costo computacional.

```
def isTrie1inTrie2(T1,T2):
    cond1=True
    cond2=True
    L1=searchWordsTrie(T1)
    L2=searchWordsTrie(T2)
    c1=L1.head
    c2=L2.head
    while c1!=None:
        if SearchTrie(T2,c1.value_L):
            c1=c1.nextNode
        else:
            cond1=False
            break
    while c2 !=None:
        if SearchTrie(T1,c2.value_L):
```

```
                c2=c2.nextNode
        else:
                cond2=False
                break
    return (cond1 or cond2)
```

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 **a<u>b</u>cd** y **dc<u>k</u>a** no son invertidas ya que difieren en un carácter.

```
def haveInverses(T):

    L1=searchWordsTrie(T)

    cnode=L1.head

    while cnode!=None:

        S=inverseString(cnode.value_L)

        print (cnode.value_L,"inv",S)

        if search(L1,S) is not None:

            return True

        cnode=cnode.nextNode

    return False


def inverseString(S):

    returnString=""

    for i in range (len(S)-1,-1,-1):

        returnString=returnString+S[i]

    return returnString
```

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 de 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 autoCompleteR(T,node,p,i):
	cont=0
	nodea=node
	while nodea.children!=None:
		if len(p)==0:
			i=True
			break
		elif cont==len(p):
			i=True
			break
		elif p[cont]==nodea.key:
			cont+=1
			nodea=nodea.children
		elif p[cont]!=nodea.key:
			if nodea.nextNode==None:
				break
			else:
				return
autoCompleteR(T,nodea.nextNode,substr(p,cont,len(p)),False)
				break

	if i:
		returnString=""
		while nodea!=None:
			print( returnString)
			if nodea.nextNode!=None:
				return returnString
			else:
				returnString=returnString+nodea.key
				nodea=nodea.children
		return returnString
	else:
		return None
```