UNCUYO
UNIVERSIDAD NACIONAL DE CUYO

FACULTAD DE INGENIERÍA

UNIVERSIDAD NACIONAL DE CUYO

FACULTAD DE INGENIERÍA

ALGORITMOS Y ESTRUCTURAS DE DATOS II

Trie

Yacante Daniel

Leg: 10341

Ejercicio 1

```python
"""
La función Insert() busca si el primer elemento de la cadena se encuentra en el nivel correspondiente, si lo esta, de forma recursiva
pasa los elementos restantes de la cadena, quitando el primero.
Si no se encuentra, en el nodo correspondiente se crea un hijo con el valor de key correspondiente y si quedan mas valores de la cadena se
los pasa de forma recursiva.
"""
def __insert(lista:list,element:str):
    nodo=__searchInList(lista,element)
    if nodo!=None:
        if len(element)>1:
            if nodo.children==None:
                nodo.children=[]
            __insert(nodo.children,element[1:])
        else:
            nodo.isEndOfWord=True
    else:
        newNode=TrieNode()
        newNode.key=element[0]
        if len(element)!=1:
            newNode.children=[]
            __insert(newNode.children,element[1:])
        else:
            newNode.isEndOfWord=True
        lista.append(newNode)
        return

def __searchInList(lista:List[TrieNode],element:str):
    for nodo in lista:
        if nodo.key==element[0]:
            return nodo
    return None

def insert(T:Trie,element:str):
    if T.root==None:
        T.root=[]
    __insert(T.root,element)
    return
```

```python
"""
La función search() realiza una búsqueda en anchura del primer carácter de la cadena en el nivel correspondiente, si el nodo final esta
marcado como fin de palabra regresa True para indicar que se encuentra la palabra en el Trie, cualquier otro caso regresa False.
"""

def __searchMatch(nl:List[TrieNode],element:str):
    nodo=__searchInList(nl,element)
    if nodo!=None:
        if len(element)>1:
            if nodo.children!=None:
                return __searchMatch(nodo.children,element[1:])
            else:
                return False
        elif nodo.isEndOfWord:
            return True
        else:
            return False
    else:
        return False

def search(T:Trie,element:str):
    if T.root!=None:
        resp= __searchMatch(T.root,element)
        return resp
    else:
        return False
```

Ejercicio 2

Para que search sea O(m) deberíamos tener una forma de encontrar a cada carácter en tiempo O(1), por lo tanto se podría usar una hashtable para indexar en la lista de hijos en que posición se encuentra tal carácter.

Ejercicio 3

```
"""
La función delete() tiene 3 posibles caminos...
    * La palabra no se encuentra: no hay nada para borrar
    * La palabra se encuentra:
        - Como parte de otra palabra: Se quita la marca de fin de palabra
        - Como palabra que no es parte de otra: Se borran nodos hasta llegar a la raíz o al proximo nodo marcado como fin de palabra

De forma recursiva busco la palabra, como si fuera la función search(), cuando se encuentra se ve si tiene la marca de fin de palabra,
si tiene hijos, simplemente se le quita la marca y se retorna. Si no tiene hijos, de la lista en la que se encuentra el nodo, se elimina
al nodo correspondiente y se regresa la "bandera" 'del' para indicar que hay que borrar nodos hasta encontrar la proxima marca de fin de palabra
Si se llega a la raíz, se borra de la raíz el nodo y se retorno True habiendo borrado la palabra por completo.
"""
def __del(nl:List[TrieNode],element:str):
    nodo=__searchInList(nl,element)
    if nodo!=None:
        if len(element)>1:
            if nodo.children!=None:
                resp=__del(nodo.children,element[1:])
                if resp=="del":
                    if nodo.isEndOfWord:
                        return True
                    else:
                        nl.remove(nodo)
                        return "del"
                else:
                    return resp
            else:
                return False
        elif nodo.isEndOfWord:
            if nodo.children!=None:
                nodo.isEndOfWord=False
                return True
            else:
                nl.remove(nodo)
                return "del"
        else:
            return False
    else:
        return False


def delete(T:Trie,element:str):
    if T.root!=None:
        resp=__del(T.root,element)
        if resp=="del":
            return True
        else:
            return resp
    else:
        return False
```

Ejercicio 4

```
"""
Ejercicio 4
Buscar palabras de longitud n que empiecen con un patron.
En principio planteo un algoritmo similar a search() para encontrar si el patron esta en el trie
luego en profundidad solo busco hasta n con una función similar a la de imprimir las palabras
"""
def __findWords(nl:List[TrieNode],cadenas:list,palabra:str,n):
    for node in nl:
        if node.isEndOfWord and n==0:
            cadenas.append(palabra+node.key)
        if node.children!=None and n>0:
            __findWords(node.children,cadenas,palabra+node.key,n-1)
    return

def __searchP(nl:List[TrieNode],pat:str,n):
    nodo=__searchInList(nl,pat)
    if nodo!=None:
        if len(pat)>1:
            if nodo.children!=None:
                return __searchP(nodo.children,pat[1:],n-1)
            else:
                return None
        elif nodo.children!=None:
            words=[]
            __findWords(nodo.children,words,"",n-1)
            return words
        else:
            return None
    else:
        return None

def searchP(T:Trie,pat:str,n:int):
    if T.root!=None:
        palabras=__searchP(T.root,pat,n-1)
        return [pat+w for w in palabras]
    else:
        return None
```

Ejercicio 5

```
"""
Ejercicio 5
Mi enfoque fue el de llamar a la función que regresa todas las palabras en el Trie, hecho esto con ambos Trie, ordeno las listas en orden
alfabético y comparo las mismas posiciones de las listas, si para la misma posición tengo palabras distintas, los Trie no son iguales.
El costo seria el O de buscar todas las palabras en ambos Trie, mas el O de ordenarlas mas el O de comparar.
Buscar todas las palabras: O(m|E|)
Ordenar las listas: O(nlogn)
Comparar las listas: O(n)
"""
def sameDoc(T1:Trie,T2:Trie):
    palT1=showTrieContent(T1)
    palT2=showTrieContent(T2)
    palT1.sort()
    palT2.sort()
    if len(palT1)==len(palT2):
        for i in range(len(palT1)):
            if palT1[i]!=palT2[i]:
                return False
        return True
    else:
        return False
```

Ejercicio 6

```
"""
Ejercicio 6
Primero obtengo todas las palabras del Trie, y luego para cada una obtengo la cadena inversa y la comparo con las demás palabras del Trie
La complejidad temporal sera la de encontrar todas las palabras y luego si n es la cantidad de palabras, O(n^2) en comparar una
cadena con las demás.
"""

def checkInvertidos(T:Trie):
    words=showTrieContent(T)
    HP=False
    for i in range(len(words)):
        for j in range(len(words)):
            if i!=j and words[i][::-1]==words[j]:
                HP=True
    return HP
```

Ejercicio 7

```
"""
La función autoCompletar() esta basada en la función search, primero busco si se encuentra el prefijo pasado, si no se encuentra regresa None,
que luego se cambia por la cadena vacía.
Si se encuentra el prefijo, paso a la función __findCommon() que va a iterar recursivamente hasta llegar a que un nodo tiene mas de 1 hijo,
indicando una derivación de las palabras, o hasta que no haya mas hijos. Si se llega a que hay mas de un hijo regresa la cadena vacía
y se va a ir añadiendo las key de los nodos superiores hasta llegar a donde se encontró el prefijo.
Si se llega a que no hay mas hijos, regresa None, que luego se cambia por la cadena vacía, en señal de que no hay palabras en común con ese prefijo
"""
def __findCommon(nl:List[TrieNode]):
    if len(nl)>1:
        return ""
    elif nl[0].children==None:
        return None
    else:
        resp=__findCommon(nl[0].children)
        if resp!=None:
            resp=nl[0].key+resp
        return resp

def __findPrefix(nl:List[Trie],prefix):
    nodo=__searchInList(nl,prefix)
    if nodo!=None:
        if len(prefix)>1:
            if nodo.children!=None:
                return __findPrefix(nodo.children,prefix[1:])
            else:
                return None
        elif nodo.children!=None:
            return __findCommon(nodo.children)
        else:
            return None
    else:
        return None


def autoCompletar(T:Trie,cadena:str):
    if T.root!=None:
        resp=__findPrefix(T.root,cadena)
        if resp==None:
            return ""
        else:
            return resp
    else:
        return False
```

Ejercicios 1, 2, 3, 7 en archivo trie.py

Ejercicio 4, 5, 6 en archivo ejercicios.py