Alumno: Manuel Matías QUESADA RICCIERI

Parte 1

Ejercicio 1

```python
def insert(T, element):
    #Si el árbol está vacío se crea una nueva lista para el nodo root
    if T.root == None:
        S = []
        T.root = S
    #Si el árbol no está vacío se utiliza la lista ya existente
    else:
        S = T.root
    return insertR(T.root, S, element)

def insertR(lastNode, S, element):
    #Revisa cada nodo de la lista S buscando un nodo con la misma key que el elemento que se está insertando
    for node in S:
        if node.key == element[0]:
            #Si se encuentra un nodo con la misma key, lo seleccionamos como nodo y continuamos la recursion
            trieNode = node
            break
    else:
        #Si no se encuentra un nodo con la misma key, se crea un nuevo nodo y se agrega a la lista S
        trieNode = TrieNode()
        trieNode.key = element[0]
        trieNode.parent = lastNode

        if len(element) == 1:
            trieNode.children = []
            trieNode.isEndOfWord = True # Agregamos esta línea para establecer isEndOfWord en True para los nodos hoja
        else:
            trieNode.children = None
        S.append(trieNode)

    if len(element) != 1:
        #Si todavía hay elementos en la palabra, se llama recursivamente a la función insertR con los elementos restantes y la lista de
        #hijos del nodo actual
        element = element[1:]
        if trieNode.children == None:
            # Si el nodo actual es una hoja, se crea un nuevo nodo vacío para sus hijos
            trieNode.children = []
        insertR(trieNode, trieNode.children, element)
```

```python
def search(T, element):
    return searchR(T.root, element)

def searchR(S, element):
    #Si la lista S es None el elemento no se encuentra en la lista
    if S == None:
        return False

    #Iteramos sobre los elementos de la lista S
    for i in range(len(S)):
        #Si encontramos un elemento cuya key es igual a la primera letra del elemento buscado, almacenamos su índice en la variable 'index' y
        #salimos del bucle con 'break'
        if S[i].key == element[0]:
            index = i
            break
    else:
        #Si no encontramos ningún elemento cuya key sea igual a la primera letra del elemento buscado, significa que el elemento no se encuentra
        #en el trie
        return False

    #Si todavía quedan letras por buscar en el elemento continuamos buscando en la lista de hijos
    if len(element) != 1:
        element = element[1:]
        return searchR(S[index].children, element)
    else:
        #Si ya no quedan letras por buscar en el elemento y el nodo actual es un nodo final de palabra entonces hemos encontrado el elemento
        #buscado
        if S[index].isEndOfWord == True:
            return True
        else:
            return False
```

Ejercicio 2

La versión de la operación search() para Trie que cumpla con la complejidad O(m) que yo propondría sería usando arrays en vez de LinkedList.
Con esto conseguimos acceder directamente a la letra que estamos buscando sin necesidad de hacer un recorrido por el árbol.

Ejercicio 3

```python
def delete(T, element):
    #Comprobamos si el elemento a eliminar se encuentra en el árbol
    if search(T, element) == False:
        return False

    #Si el elemento existe comenzamos la eliminación
    currentNode = T.root
    parentNode = None
    index = None

    #Recorremos los nodos del árbol para llegar a la ultima letra del elemento
    for i in range(len(element)):
        #Para cada nodo, buscamos el índice del hijo que contiene la letra correspondiente al elemento.
        for j in range(len(currentNode)):
            if currentNode[j].key == element[i]:
                index = j
                break

        #Si el nodo actual no tiene hijos y todavía no hemos alcanzado la última letra del elemento, eliminamos el nodo actual y devolvemos True.
        if currentNode[index].children is None and i < len(element) - 1:
            currentNode.pop(index)
            return True
        #Si el nodo actual tiene hijos, avanzamos al siguiente nodo y seguimos recorriendo el árbol.
        else:
            parentNode = currentNode
            currentNode = currentNode[index].children

    #Comprobamos si el nodo que contiene la última letra del elemento tiene hijos.
    #Si no tiene hijos, eliminamos el nodo del árbol y devolvemos True.
    if len(currentNode) == 0:
        parentNode.pop(index)
        return True
    #Si el nodo tiene hijos, simplemente marcamos su bandera 'isEndOfWord' como False y devolvemos True.
    else:
        currentNode[index].isEndOfWord = False
        return True
```

Parte 2

Ejercicio 4

```python
def getWords(T):
    #Creo una lista para almacenar los elementos
    words = []

    #Si el root del árbol no es nulo comienzo a recorrer sus nodos
    if T.root != None:
        #Creo una pila con todos los hijos de la raiz y su prefijo en la pila
        stack = [(node, '') for node in T.root]

        #Mientras las pila no esté vacía extrae el siguiente nodo y su prefijo de la pila
        while stack:
            node, prefix = stack.pop()
            #Creamos una nueva palabra al agregar la clave del nodo al prefijo
            word = prefix + node.key

            #Si este nodo es el final de una palabra, la agregamos a la lista de palabras encontradas
            if node.isEndOfWord:
                words.append(word)

            #Si el nodo tiene hijos, los agregamos a la pila junto con el prefijo actualizado.
            if node.children != None:
                for child in reversed(node.children):
                    stack.append((child, word))
    return words
```

```python
def searchWords(T, patron, long):
    #Obtiene la lista de palabras dentro del árbol
    mainlist = getWords(T)

    #Recorre la lista de palabras
    for i in range(len(mainlist)):
        #Si la longitud de la palabra es igual a la longitud deseada y el primer carácter de la palabra es igual al patrón, se imprime la palabra
        if len(mainlist[i]) == long and mainlist[i][0] == patron:
            print(mainlist[i])
    return
```

Ejercicio 5

```python
def sameWords(Trie1, Trie2):
    #Obtengo todas las palabras de ambos árboles Trie y las ordeno por orden alfabético
    listTrie1 = sorted(getWords(Trie1))
    listTrie2 = sorted(getWords(Trie2))

    #Compruebo si la Trie1 es una sublista de Trie2
    if set(listTrie1).issubset(set(listTrie2)):
        return True

    #Compruebo si la Trie2 es una sublista de Trie1
    elif set(listTrie2).issubset(set(listTrie1)):
        return True

    #Compruebo si Trie1 y Trie2 son iguales
    elif listTrie1 == listTrie2:
        return True

    return False
```

El costo computacional del algoritmo es de O(n1 + n2 + n log n)
El costo computacional de la función getWords es O(n) ya que en todos los casos recorrerá todo el árbol en busca de todas las palabras del mismo
Al llamar a getWords() dentro de sameWords para llenar las listas 1 y 2 obtenemos un costo computacional O(n1 + n2), donde las n son los números totales de nodos en los Tries.
Luego, dentro de sameWords() se llevan a cabo 2 ordenaciones que tienen un costo de O(n log n), donde n es la longitud de la lista de palabras
Entonces en conclusión obtendremos un costo computacional de O(n1 + n2 + n log n)

Ejercicio 6

```python
def twiceStringInverted(T):
    #Obtengo todas las palabras que se encuentran en el Trie
    wordslist = getWords(T)

    for i in range(len(wordslist)):
        #Invierto la palabra que tome de la lista y la comparo a resto de las palabras
        reversedword = "".join(reversed(wordslist[i]))
        for j in range(i+1, len(wordslist)):
            #Si hay una palabra que, sea igual a la palabra que ya teniamos invertida, significa que en el documento existen dos cadenas invertidas
            if reversedword == wordslist[j]:
                return True
    return False
```

Ejercicio 7

```python
def searchWordsList(T, patron):
    #Obtiene la lista de palabras dentro del árbol
    mainlist = getWords(T)

    #Crea un array vacío para almacenar las palabras que cumplan las condiciones
    wordsList = []

    # Recorre la lista de palabras
    for i in range(len(mainlist)):
        #Añade todas las palabras que empiecen por dicho patron a la lista
        if mainlist[i].startswith(patron):
            wordsList.append(mainlist[i])

    # Retorna el array con las palabras encontradas
    return wordsList

def autoCompletar(T, element):
    #Obtiene la lista de palabras dentro del árbol que empiecen con el elemento introducido
    wordsList = searchWordsList(T, element)

    #Si la lista está vacía, devuelve una cadena vacía
    if len(wordsList) == 0:
        return ""

    #Si la lista solo tiene una palabra, devuelve la parte de la palabra despues del elemento dado
    elif len(wordsList) == 1:
        return wordsList[0][len(element):]

    #Si la lista tiene mas de una palabra, busca el punto en el que las palabras de la lista difieren y devuelve esa parte de la primera palabra
    else:
        for i in range(len(element), len(wordsList[0])):
            #Comprueba si el carácter actual en todas las palabras de la lista es el mismo que el de la primera palabra
            if any(palabra[i] != wordsList[0][i] for palabra in wordsList):
                #Si encuentra una diferencia, devuelve la parte de la primera palabra desde el inicio hasta ese punto
                return wordsList[0][len(element):i]

        #Si no hay diferencias, devuelve la parte de la primera palabra después del elemento dado
        return wordsList[0][len(element):]
```