Martin Farres 13495

TP – Tries

Code:

```
# ---------------------------------- Insert ------------------------------------

def insert(T, element):
    if T.root == None:
        newRoot = TrieNode()
        T.root = newRoot
    currentNode = T.root
    for i in range(0, len(element)):
        childrenList = currentNode.children
        newNode = False
        if childrenList != None:
            newNode = linkedlist.getNodeTrie(
                childrenList, linkedlist.searchTrie(childrenList, element[i]))
        else:
            childrenList = linkedlist.LinkedList()
            currentNode.children = childrenList
        if newNode == False:
            newNode = TrieNode()
            newNode.parent, newNode.key = currentNode, element[i]
            linkedlist.add(childrenList, newNode)
        if i == len(element)-1:
            newNode.isEndOfWord = True
        currentNode = newNode


# ---------------------------------- Search ------------------------------------
def search(T, element):
    currentNode = T.root
    return searchR(currentNode, element)

def searchR(currentNode, element):
    childrenList = currentNode.children
    newNode = linkedlist.getNodeTrie(
        childrenList, linkedlist.searchTrie(childrenList, element[0]))
    if newNode == False:
        return False
    if len(element) == 1 and newNode.isEndOfWord == True:
        return True
    element = element[1:]
    return searchR(newNode, element)
```

```
# ---------------------------------- Delete -------------------------------------

def delete(T, element):
    currentNode = T.root
    nodeToDelete = [None, None]
    # Searching and saving the last letter that is the end of a word inside the
element
    for i in range(0, len(element)):
        childrenList = currentNode.children
        newNode = linkedlist.getNodeTrie(
            childrenList, linkedlist.searchTrie(childrenList, element[i]))
        # Saving the first node to delete if no other is found
        if i == 0:
            nodeToDelete[0], nodeToDelete[1] = newNode, i
        if newNode == False:
            return False
        # Saving node to delete if it isEndOfWord and not the last
        if newNode.isEndOfWord == True and i != len(element):
            nodeToDelete[0], nodeToDelete[1] = newNode, i
        currentNode = newNode
    # Deleting the Node
    if newNode.children == None:  # Last Node of our elemnt has no children, we
delete the whole element
        if nodeToDelete[1] == 0:
            linkedlist.deleteTrie(T.root.children, element[0])
            nodeToDelete[0].children = None
        else:
            linkedlist.deleteTrie(
                nodeToDelete[0].children, element[nodeToDelete[1]+1])
            nodeToDelete[0].children = None
    else:
        newNode.isEndOfWord = False
    return True


# ----------------------- Search by Pattern ----------------------------------
def searchByPattern(T, p, n):
    currentNode = searchPatternR(T.root, p)
    if currentNode == False:
        return False
    newTree = Trie()
    newTree.root = currentNode
    listWords = printTrie(newTree, p)
    listWords = [i for i in listWords if len(i) == n]
    return listWords
```

Martin Farres 13495

```
def searchPatternR(currentNode, element):
    childrenList = currentNode.children
    newNode = linkedlist.getNodeTrie(
        childrenList, linkedlist.searchTrie(childrenList, element[0]))
    if newNode == False:
        return False
    if len(element) == 1:
        return newNode
    element = element[1:]
    return searchPatternR(newNode, element)

# ----------------------- Print Words in Trie ---------------------------------


def printTrie(T, pattern=""):
    listWords = []
    letters = "" + pattern
    node = T.root
    printTrieR(node.children.head, listWords, letters)
    return listWords


def printTrieR(listNode, listWords,  letters):
    if listNode == None:
        return
    letters = letters + listNode.value.key
    if listNode.value.isEndOfWord == True:
        listWords.append(letters)
    if listNode.value.children != None:
        printTrieR(listNode.value.children.head, listWords, letters)
        letters = letters[:-1]
    return printTrieR(listNode.nextNode, listWords, letters)


# ---------------------------- Tries Iguales ---------------------------------

def sameTries(T, P):
    Tlist = printTrie(T)
    Plist = printTrie(P)
    if Tlist == Plist:
        return True
    return False
```

```
# ---------------------------- Palabras Invertidas ----------------------------

def reversWords(T):
    allWords = printTrie(T)
    currentNode = allWords.head
    nextNode = allWords.head.nextNode

    while nextNode != None and currentNode != None:
        if (currentNode.value == nextNode.value[::-1]):
            return True
        nextNode = nextNode.nextNode
        if (nextNode == None):
            currentNode = currentNode.nextNode
            nextNode = currentNode
    return False


# ---------------------------- Auto Completar Palabras ------------------------

def autofillWords(T, p):
    currentNode = searchPatternR(T, p)
    if currentNode == False:
        return ""
    newTree = Trie()
    newTree.root = currentNode
    listWords = printTrie(newTree, p)
    if len(listWords) == 1:
        return listWords[0]
    return ""
```

Exercise 2:

Suponiendo que conocemos todos los caracteres posibles que puedan ser usados en el trie. Supongamos que nuestro trie solo acepta las 26 letras del abecedario español. Si generamos un array de 0 a 25, donde cada elemento del array equivale a una letra, siendo a = 0 y así sucesivamente hasta llegar a z = 25.

De esta manera, para buscar cada carácter por nivel en el árbol seria de O(1). Entonces al hacer esta búsqueda m cantidades de veces según el largo de la palabra, obtenemos que la complejidad de la función search() es de O(m).

Martin Farres 13495