

TP TRIE

Tomás Rando - 14004 - LCC

Para la realización del TP hice dos implementaciones. Una con listas de python y otra con listas LinkedList. En la segunda mitad del tp se encuentran las implementaciones con LinkedList y en GitHub se encuentran los códigos correspondientes junto a un archivo main.py que contiene algunos casos de prueba para ambas implementaciones.

Punto 1)

```
#Busca en una lista si existe algún elemento con la key == element, si es así devuelve el índice
def searchInL(L, element):
    if L == None:
        return None
    length = len(L)
    for i in range(0, length):
        if L[i].key == element:
            return i
    return None
```

insert(T, element)

```
def insert(T, element):
    if T.root == None:
        L = linkedlist.LinkedList()
        T.root = L
    else:
        L = T.root
    return insertR(T.root, L, element)

def insertR(LastNode, L, element):
    condition = searchInL(L, element[0])

    if condition == None:
        TNode = TrieNode()
        TNode.key = element[0]
        TNode.parent = LastNode
        if len(element) == 1:
            TNode.children = None
        else:
            newList = linkedlist.LinkedList()
            TNode.children = newList
            linkedlist.add(L, TNode)
    else:
        TNode = condition

    if len(element) != 1:
        element = element[1:]
        if condition == None:
            insertR(TNode, newList, element)
        else:
            if condition.children == None:
                newList = linkedlist.LinkedList()
                condition.children = newList
            insertR(condition, condition.children, element)
    return

else:
    TNode.isEndOfWord = True
    return
```

search(T, element)

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

def searchR(L, element):
    if L == None:
        return False

    condition = searchInL(L, element[0])

    if condition == None:
        return False

    if len(element) != 1:
        element = element[1:]
        return searchR(L[condition].children, element)
    else:
        if L[condition].isEndOfWord == True:
            return True
        else:
            return False
```

Punto 2)

Una versión del search() cuya complejidad sea $O(m)$ sería una versión del Trie implementado mediante arrays en vez de listas enlazadas (utilizando posiciones conocidas de antemano para insertar las letras). De esta manera, el acceso a cada letra sería de $O(1)$ y solo importaría recorrer la palabra, que se realizaría en $O(m)$, siendo m la longitud de la palabra

Punto 3)

delete(T, element)

```
def searchForLastNode(T, element):
    return searchForLastNodeR(T.root, element)

def searchForLastNodeR(L, element):
    condition = searchInL(L, element[0])
    if condition == None:
        return False

    if len(element) != 1:
        element = element[1:]
        return searchForLastNodeR(L[condition].children, element)
    else:
        if L[condition].isEndOfWord == True:
            return L[condition]
        else:
            return False
```

```

def delete(T, element):
    condition = search(T, element)
    if condition == False:
        return False
    finalNode = searchForLastNode(T, element)
    finalNode.isEndOfWord = False

    return deleteR(T, finalNode, element)

def deleteR(T, finalNode, element):
    if finalNode.children != None:
        finalNode.isEndOfWord = False
        return True
    else:
        if finalNode.isEndOfWord == False:
            if finalNode.parent == T.root:
                T.root.remove(finalNode)
                return True
            parentNode = finalNode.parent
            parentList = parentNode.children
            parentList.remove(finalNode)

            if len(parentList) == 0:
                parentNode.children = None
                return deleteR(T, finalNode.parent, element)
        else:
            return True

```

Punto 4)

```

def searchLastNodePattern(T, element):
    return searchLastNodePatternR(T.root, element)

def searchLastNodePatternR(L, element):
    condition = searchInL(L, element[0])
    if condition == None:
        return False

    if len(element) != 1:
        element = element[1:]
        return searchLastNodePatternR(L[condition].children, element)
    else:
        return L[condition]

```

```

def imprimir(T, p, n):
    node = searchLastNodePattern(T, p)
    if node == False:
        return None
    auxiliarList = []
    lenWord = len(p)
    lista = imprimirR(node, p, n - lenWord + 1, 1, auxiliarList)

    if lenWord == n:
        lista.insert(0, p)
    print(lista)

def imprimirR(node, p, n, cont, auxiliarList):
    if cont != 1:
        auxiliarList[0] = auxiliarList[0] + node.key

    if n == cont:
        if node.isEndOfWord == False:
            auxiliarList.pop(0)
        return auxiliarList

    if node.children == None:
        auxiliarList.pop(0)
        return auxiliarList

    if cont != 1:
        aux = auxiliarList[0]

    j = 0
    for i in node.children:
        j = j + 1
        if cont == 1:
            auxiliarList.insert(0, p)

        if (j != 1) and (cont != 1):
            auxiliarList.insert(0, aux)

        imprimirR(i, p, n, cont + 1, auxiliarList)
    return auxiliarList

```

Punto 5)

```
def getWords(T):
    if T.root == None:
        return None
    else:
        auxiliarList = []
        for i in T.root:
            auxiliarList = getWordsR(i, auxiliarList, 1)
        return auxiliarList

def getWordsR(node, auxiliarList, cont):
    if cont != 1:
        auxiliarList[0] = auxiliarList[0] + node.key

    if node.isEndOfWord == True:
        if node.children == None:
            return auxiliarList
        else:
            auxiliarList.insert(1, auxiliarList[0])

    if node.children == None:
        return auxiliarList

    if cont != 1:
        aux = auxiliarList[0]
    j = 0
    for i in node.children:
        j = j + 1
        if cont == 1:
            auxiliarList.insert(0, node.key)
        if (j != 1) and (cont != 1):
            auxiliarList.insert(0, aux)
        auxiliarList = getWordsR(i, auxiliarList, cont + 1)
    return auxiliarList
```

```
def equal(T1, T2):
    list1 = getWords(T1)
    list2 = getWords(T2)

    if (list1 == None) or (list2 == None):
        return False

    if (len(list1) != len(list2)):
        return False

    areEqual = False
    for i in list1:
        elemento = i
        for j in list2:
            if elemento == j:
                areEqual = True
        if areEqual == False:
            return areEqual
        areEqual = False
    return True
```

La complejidad es de $O(m^2)$, siendo m la cantidad de palabras de los Trie, ya que se comparan las dos listas al final usando dos bucles anidados que recorren la longitud de las 2 listas.

Punto 6)

```
def getWordsBackwards(T):
    if T.root == None:
        return None
    else:
        auxiliarList = []
        for i in T.root:
            auxiliarList = getWordsBackwardsR(i, auxiliarList, 1)
        return auxiliarList

def getWordsBackwardsR(node, auxiliarList, cont):

    if cont != 1:
        auxiliarList[0] = node.key + auxiliarList[0]

    if node.isEndOfWord == True:
        if node.children == None:
            return auxiliarList
        else:
            auxiliarList.insert(1, auxiliarList[0])

    if node.children == None:
        return auxiliarList

    if cont != 1:
        aux = auxiliarList[0]
        j = 0
        for i in node.children:
            j = j + 1
            if cont == 1:
                auxiliarList.insert(0, node.key)
            if (j != 1) and (cont != 1):
                auxiliarList.insert(0, aux)
            auxiliarList = getWordsBackwardsR(i, auxiliarList, cont + 1)
        return auxiliarList
```

```
def invertidas(T):
    if T.root == None:
        return False

    list1 = getWords(T)
    list2 = getWordsBackwards(T)
    length = len(list1)
    for i in range(0, length):
        element = list1[i]
        for j in range(0, length):
            if element == list2[j]:
                return True
    return False
```

Punto 7)

```
def autoComplete(Trie, cadena):
    node = searchLastNodePattern(Trie, cadena)
    word = ""
    return autoCompleteR(node, word)

def autoCompleteR(node, word):
    if node.children == None:
        return word
    if len(node.children) != 1:
        return word
    else:
        word = word + node.children[0].key
        return autoCompleteR(node.children[0], word)
```

IMPLEMENTACIONES CON LINKEDLIST

Punto 1)

```
#Busca en una LinkedList un nodo con una key == element.
#Si la encuentre devuelve el TNode, caso contrario devuelve None
def searchInL(L, element):
    if L.head == None:
        return None
    currentNode = L.head
    while currentNode != None:
        if currentNode.value.key == element:
            return currentNode.value
        currentNode = currentNode.nextNode
    return None
```

```

def insert(T, element):
    if T.root == None:
        L = linkedlist.LinkedList()
        T.root = L
    else:
        L = T.root
    return insertR(T.root, L, element)

def insertR(LastNode, L, element):
    condition = searchInL(L, element[0])

    if condition == None:
        TNode = TrieNode()
        TNode.key = element[0]
        TNode.parent = LastNode
        if len(element) == 1:
            TNode.children = None
        else:
            newList = linkedlist.LinkedList()
            TNode.children = newList
            linkedlist.addatend(L, TNode)
    else:
        TNode = condition

    if len(element) != 1:
        element = element[1:]
        if condition == None:
            insertR(TNode, newList, element)
        else:
            if condition.children == None:
                newList = linkedlist.LinkedList()
                condition.children = newList
            insertR(condition, condition.children, element)
    return

    else:
        TNode.isEndOfWord = True
        return

```

```

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

def searchR(L, element):
    if L == None:
        return False

    condition = searchInL(L, element[0])

    if condition == None:
        return False

    if len(element) != 1:
        element = element[1:]
        return searchR(condition.children, element)
    else:
        if condition.isEndOfWord == True:
            return True
        else:
            return False

```


Punto 3)

*#Función que busca un elemento en el Trie, si existe
#devuelve el último nodo correspondiente al elemento. Caso contrario, devuelve False*

```
def searchForLastNode(T, element):  
    return searchForLastNodeR(T.root, element)  
  
def searchForLastNodeR(L, element):  
    condition = searchInL(L, element[0])  
  
    if condition == None:  
        return False  
  
    if len(element) != 1:  
        element = element[1:]  
        return searchForLastNodeR(condition.children, element)  
    else:  
        if condition.isEndOfWord == True:  
            return condition  
        else:  
            return False
```

```
def delete(T, element):  
    condition = search(T, element)  
    if condition == False:  
        return False  
    finalNode = searchForLastNode(T, element)  
    finalNode.isEndOfWord = False  
  
    return deleteR(T, finalNode, element)  
  
def deleteR(T, finalNode, element):  
  
    if finalNode.children != None:  
        finalNode.isEndOfWord = False  
        return True  
    else:  
        if finalNode.isEndOfWord == False:  
            if finalNode.parent == T.root:  
                linkedlist.delete(T.root, finalNode)  
                return True  
            parentNode = finalNode.parent  
            parentList = parentNode.children  
            linkedlist.delete(parentList, finalNode)  
  
            if linkedlist.length(parentList) == 0:  
                parentNode.children = None  
            return deleteR(T, finalNode.parent, element)  
        else:  
            return True
```

Punto 4)

```
def searchLastNodePattern(T, element):
    return searchLastNodePatternR(T.root, element)

def searchLastNodePatternR(L, element):
    condition = searchInL(L, element[0])
    if condition == None:
        return False

    if len(element) != 1:
        element = element[1:]
        return searchLastNodePatternR(condition.children, element)
    else:
        return condition
```

```
def imprimir(T, p, n):
    node = searchLastNodePattern(T, p)
    if node == False:
        return None
    auxiliarList = linkedlist.LinkedList()
    lenWord = len(p)
    lista = imprimirR(node, p, n - lenWord + 1, 1, auxiliarList)

    if lenWord == n:
        linkedlist.add(auxiliarList, p)
        linkedlist.printlist(lista)

def imprimirR(node, p, n, cont, auxiliarList):
    if cont != 1:
        auxiliarList.head.value = auxiliarList.head.value + node.key

    if n == cont:
        if node.isEndOfWord == False:
            linkedlist.pop(auxiliarList)
        return auxiliarList

    if node.children == None:
        linkedlist.pop(auxiliarList)
        return auxiliarList

    if cont != 1:
        aux = auxiliarList.head.value

    j = 0
    currentNode = node.children.head
    while currentNode != None:
        j = j + 1
        if cont == 1:
            linkedlist.add(auxiliarList, p)
        if (j != 1) and (cont != 1):
            linkedlist.add(auxiliarList, aux)

        imprimirR(currentNode.value, p, n, cont + 1, auxiliarList)
        currentNode = currentNode.nextNode
    return auxiliarList
```

Punto 5)

```
def getWords(T):
    if T.root == None:
        return None
    else:
        auxiliarList = linkedlist.LinkedList()
        currentNode = T.root.head
        while currentNode != None:
            auxiliarList = getWordsR(currentNode.value, auxiliarList, 1)
            currentNode = currentNode.nextNode
        return auxiliarList

def getWordsR(node, auxiliarList, cont):

    if cont != 1:
        auxiliarList.head.value = auxiliarList.head.value + node.key

    if node.isEndOfWord == True:
        if node.children == None:
            return auxiliarList
        else:
            linkedlist.add(auxiliarList, auxiliarList.head.value)

    if node.children == None:
        return auxiliarList

    if cont != 1:
        aux = auxiliarList.head.value

    j = 0
    currentNode = node.children.head
    while currentNode != None:
        j = j + 1
        if cont == 1:
            linkedlist.add(auxiliarList, node.key)
        if (j != 1) and (cont != 1):
            linkedlist.add(auxiliarList, aux)
        auxiliarList = getWordsR(currentNode.value, auxiliarList, cont + 1)
        currentNode = currentNode.nextNode
    return auxiliarList

def equal(T1, T2):
    list1 = getWords(T1)
    list2 = getWords(T2)

    if (list1.head == None) or (list2.head == None):
        return False

    if (linkedlist.length(list1)) != (linkedlist.length(list2)):
        return False

    areEqual = False
    currentNode1 = list1.head
    while currentNode1 != None:
        elemento = currentNode1.value
        currentNode2 = list2.head

        while currentNode2 != None:
            if elemento == currentNode2.value:
                areEqual = True
                currentNode2 = currentNode2.nextNode
            if areEqual == False:
                return areEqual
            areEqual = False
            currentNode1 = currentNode1.nextNode
    return True
```

Punto 6)

```
def getWordsBackwards(T):
    if T.root == None:
        return None
    else:
        auxiliarList = linkedlist.LinkedList()
        currentNode = T.root.head
        while currentNode != None:
            auxiliarList = getWordsBackwardsR(currentNode.value, auxiliarList, 1)
            currentNode = currentNode.nextNode
        return auxiliarList

def getWordsBackwardsR(node, auxiliarList, cont):

    if cont != 1:
        auxiliarList.head.value = node.key + auxiliarList.head.value

    if node.isEndOfWord == True:
        if node.children == None:
            return auxiliarList
        else:
            linkedlist.add(auxiliarList, auxiliarList.head.value)

    if node.children == None:
        return auxiliarList

    if cont != 1:
        aux = auxiliarList.head.value
        j = 0
        currentNode = node.children.head
        while currentNode != None:
            j = j + 1
            if cont == 1:
                linkedlist.add(auxiliarList, node.key)
            if (j != 1) and (cont != 1):
                linkedlist.add(auxiliarList, aux)
            auxiliarList = getWordsBackwardsR(currentNode.value, auxiliarList, cont + 1)
            currentNode = currentNode.nextNode
        return auxiliarList

def invertidas(T):
    if T.root == None:
        return False

    list1 = getWords(T)
    list2 = getWordsBackwards(T)
    currentNode = list1.head
    while currentNode != None:
        element = currentNode.value
        currentNode2 = list2.head
        while currentNode2 != None:
            if element == currentNode2.value:
                return True
            currentNode2 = currentNode2.nextNode
        currentNode = currentNode.nextNode
    return False
```

Punto 7)

```
def autoCompleteR(Trie, cadena):
    node = searchLastNodePattern(Trie, cadena)
    word = ""
    return autoCompleteR(node, word)

def autoCompleteR(node, word):
    if node.children == None:
        return word
    if node.children.head.nextNode != None:
        return word
    else:
        word = word + node.children.head.value.key
        return autoCompleteR(node.children.head.value, word)
```

Link al código: <https://github.com/Kilxz/algoritmos2/tree/main/practicas/tp-trie/code>