Franco Yudica 13922

Algoritmos y estructuras de datos II

TP Trie

Franco Yudica 13922

Parte 1 3

Franco Yudica 13922

Parte 1

Ejercicio 1: Implementación de insert y search

```python
def insert(self, word):

    # Otherwise
    char_index = 0
    self._insert_recursive(self.root, word, char_index)

def _insert_recursive(self, current, word, char_index):

    # Means all the nodes were added
    if char_index >= len(word):
        return

    # Tries to find if the char was already added
    for child in current.children:

        # If the child has the same value (char)
        if child.key == word[char_index]:
            self._insert_recursive(child, word, char_index + 1)
            return

    # In this case, there are no children created. So it should add nodes untill it ends
    # There is no recursion needed
    word_length = len(word)
    parent = current
    while char_index < word_length:

        # Creates the node
        node = TrieNode()
        node.key = word[char_index]

        # Links the nodes
        node.parent = parent
        parent.children.append(node)
        parent = node
        char_index += 1

    parent.is_end = True
```

Franco Yudica 13922

```python
def search(self, word):

    def _search_recursive(current, word, char_index):

        current_char = word[char_index]
        found = False

        # Tries to find a child with the same char
        for child in current.children:
            if child.key == current_char:

                # When the last char of the word was reached
                if char_index + 1 == len(word):
                    return child.is_end

                found = _search_recursive(child, word, char_index + 1)
                break

        return found

    return _search_recursive(self.root, word, 0)
```

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(m).

Supongamos que trabajamos con el abecedario en inglés y con caracteres en minúsculas entonces hay 26 posibles caracteres para formar palabras.
La siguiente idea se basa fuertemente en el código ascii de los caracteres del abecedario. Sabemos que la tabla de ascii empieza con “a” cuyo valor es 97 y termina, en orden alfabético con “z” cuyo valor es 122.
Si nosotros en vez de utilizar linked list para almacenar los hijos de un nodo utilizamos un array, pero un array con un tamaño de 26, podemos sacar provecho de la siguiente manera:
. Se pasa el carácter “a”, cuyo ascii es 97, y se le resta 97 obteniendo 0
. Se pasa el carácter “”z”, cuyo ascii es 122, y se le resta 97, obteniendo 25

Es decir que para todos los caracteres vamos a poder acceder al índice de almacenamiento de esta manera, resultando en una complejidad de O(1), luego, si la longitud de la palabra buscada es m, la nueva versión de search va a tener una complejidad O(m)

Franco Yudica 13922

Ejercicio 3: Implementación delete

```python
def delete(self, word):

    # The word is not in the trie
    if not self.search(word):
        return

    def _delete_recursive(current, word, char_index):

        current_char = word[char_index]

        # Tries to find a child with the same char
        for child in current.children:
            if child.key == current_char:
                break

        # Child is the one that contains the current character. This code assumes that the word is contained

        # The end of the word is reached
        if char_index + 1 == len(word):

            # When child has children, just changes the end flag
            if len(child.children) != 0:
                child.is_end = False
                return

            # Here we can remove the word
            # Traverse upwards to remove the excess
            node = child
            while node is not None:

                parent = node.parent
                # Unlinks the node, resulting in the removal of the excess of the word
                if parent.is_end:

                    parent.children.remove(node)
                    node.parent = None
                    break

                node = parent

            return

        # Continues with the recursion until it finds the end
        _delete_recursive(child, word, char_index + 1)


    _delete_recursive(self.root, word, 0)
```

Franco Yudica 13922

Parte 2

Ejercicio 4: Implementar un algoritmo que dado un árbol Trie T, un patrón p y un entero n, escriba todas las palabras del árbol que empiezan por p y sean de longitud n.

```python
def starts_with(self, char, length):
    """
    Returns a list of all the words of length that starts with char
    """

    def find_words(current, words, current_str, max_length):
        # Recursive function for searching the word
        # When the length is reached
        if len(current_str) == max_length:
            if current.is_end:
                # Adds the word and quits recursion
                words.append(current_str)
            return

        # For every child
        for child in current.children:
            find_words(child, words, current_str + child.key, max_length)

    # Checks there if exists any word starting with char
    for child in self.root.children:
        if child.key == char:
            break
    else:
        return

    words = []

    find_words(child, words, child.key, length)
    return words
```

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 Trie pertenecen al mismo documento cuando:

- Ambos Trie sean iguales (esto se debe cumplir)
- Si la implementación está basada en LinkedList, considerar el caso donde las palabras hayan sido insertadas en un orden diferente.

En otras palabras, analizar si todas las palabras de T1 se encuentran en T2.

Franco Yudica 13922

```python
def is_same_file(self, other):
    # Returns True if all the words of self, are inside the other Trie

    def is_contained(current_self, current_other):

        # When it reaches the end
        if len(current_self.children) == 0:
            return True

        for self_child in current_self.children:

            contained_sub_tree = False
            # If the key is contained in the other tree, just breaks
            for other_child in current_other.children:
                if self_child.key == other_child.key:
                    contained_sub_tree = is_contained(self_child, other_child)
                    break
            else:
                # When the self_child key isn't contained
                return False

            # If one single key is not contained in the sub_tree
            if not contained_sub_tree:
                return False

        return True

    return is_contained(self.root, other.root)
```

El costo computacional en el peor de los casos se da cuando se recorre todo el Trie 1. Por lo tanto, si la altura del árbol Trie 1 es de m, tenemos que el costo computacional del algoritmo resulta O(m|Σ) donde |Σ| es la cantidad de caracteres del abecedario

Franco Yudica 13922

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 abcd y dcka no son invertidas ya que difieren en un carácter.

```python
def get_reversed_pairs(self):
    # Returns all the list of reversed words
    words = self.words()

    def reversed_word(word):
        # Given a word returns the reversed version
        new_reversed = ""
        for char in reversed(word):
            new_reversed += char
        return new_reversed

    words_pairs = []
    for word in words:
        new_reversed = reversed_word(word)
        if new_reversed in words:
            words_pairs.append((word, new_reversed))

    return words_pairs
```

Nótese que en lugar de simplemente verificar si existen palabras invertidas y retornar True o False, me pareció interesante retornar los pares en una lista.
La implementación anterior utiliza la función words() que retorna la lista de todas las palabras contenidas en el Trie

```python
def words(self) -> list:

    def find_words(current, words, current_str):

        # If we find the end of a word, adds the word
        if current.is_end:
            words.append(current_str)

        # Continues adding the rest of the word
        for child in current.children:
            find_words(child, words, current_str + child.key)

    words = []

    for child in self.root.children:
        find_words(child, words, child.key)

    return words
```

Franco Yudica 13922

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 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”.

Franco Yudica 13922

```python
def autocompelte(trie, prefix):
    """ Given a Trie and a prefix, returns all the autocompleted words that start with the prefix"""

    """ Utility functions"""
    def _find_last_node_recursive(current, prefix, word_index):
        # Returns the node that contains the last character of the word
        # 'H' 'e' 'l' 'l' 'o'
        # Returns the noode that has the key 'o'
        if word_index == len(prefix):
            return current

        char = prefix[word_index]

        # Tries to find the child that contains the char
        for child in current.children:
            if child.key == char:
                return _find_last_node_recursive(child, prefix, word_index + 1)

        # Here, the node wasn't found, meaning the word isn't contained in the Trie

    """ Utility functions"""
    def find_words(current, words, current_str):
        # Given a node, appends the sub_tree words
        if current.is_end:
            words.append(current_str)

        # Translates recusion for all the children
        for child in current.children:
            find_words(child, words, current_str + child.key)


    """1. Finds the last node"""
    last_node = _find_last_node_recursive(trie.root, prefix, 0)

    # When the word wasn't found
    if last_node is None:
        return

    """2. Finds all the words that start with the prefix"""
    # Otherwise, traverses donwards
    words = []
    find_words(last_node, words, prefix)

    return words
```

La implementación toma como parámetro Trie y un prefijo.
Lo primero que hago es encontrar el nodo del Trie que tiene como key a el último patrón del prefijo. Luego busco las palabras que empiezan con el prefijo a partir de last_node, esto lo puedo hacer mediante la función find_words, la cuál tiene como salida una lista de palabras. En nuestro caso, esa lista de palabras es la que contiene todas las palabras autocompletadas. Luego retorno las palabras
Entiendo que la consigna no pide retornar todas las palabras, pero me pareció que la resolución dada resulta más completa y se puede aplicar en más situaciones, pues así es como funcionan las terminales