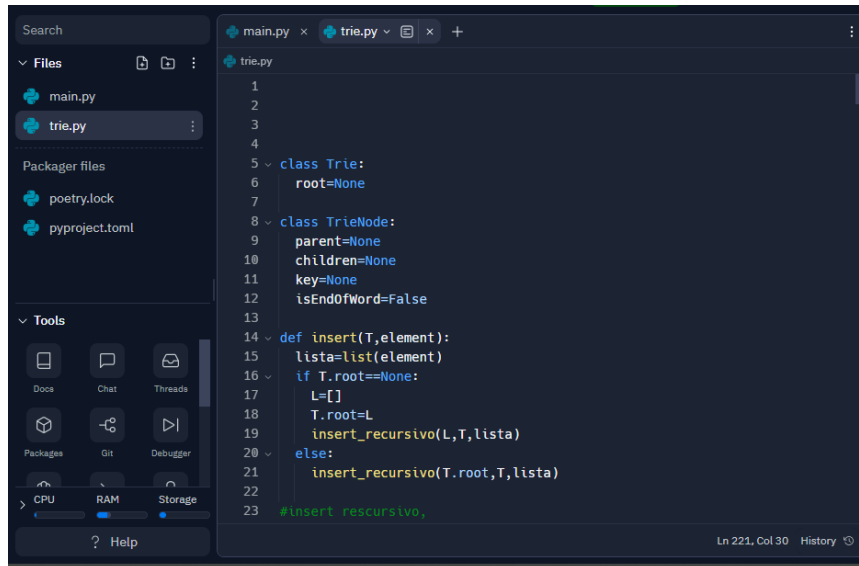


Algoritmos y Estructura de Datos II

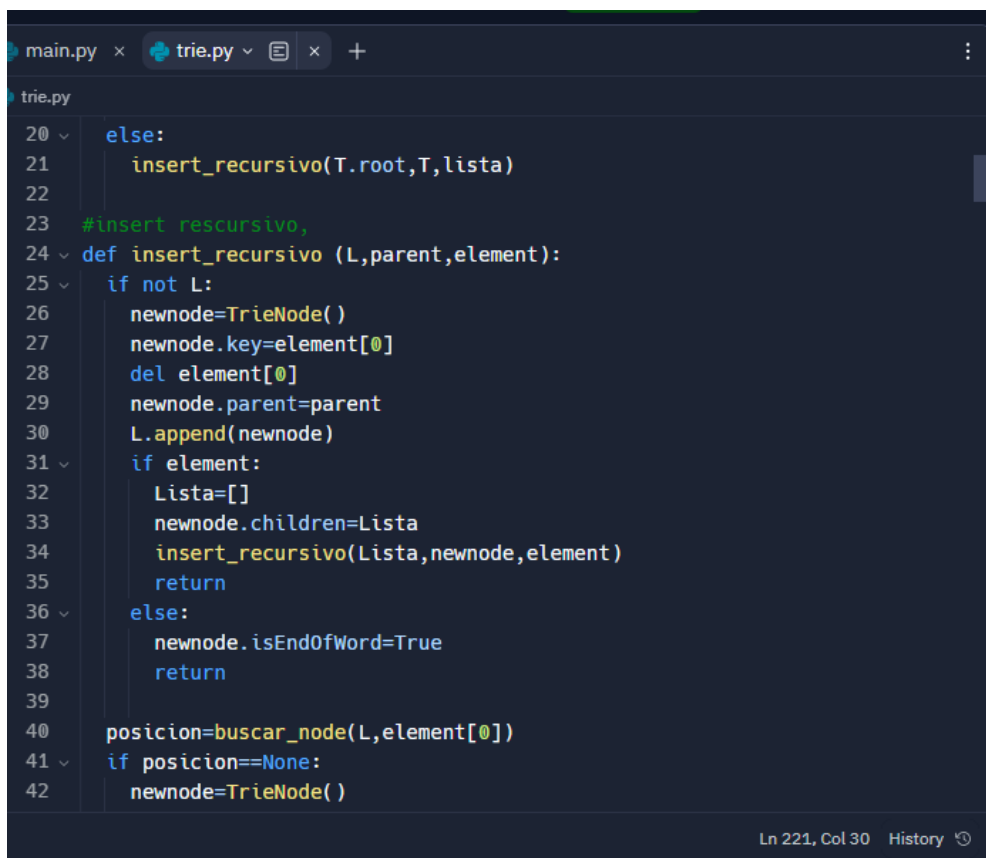
Trie

Coppede Santos Ignacio

Ejercicio 1



```
1
2
3
4
5 class Trie:
6     root=None
7
8 class TrieNode:
9     parent=None
10    children=None
11    key=None
12    isEndOfWord=False
13
14 def insert(T,element):
15     lista=list(element)
16     if T.root==None:
17         L=[]
18         T.root=L
19         insert_recursivo(L,T,lista)
20     else:
21         insert_recursivo(T.root,T,lista)
22
23 #insert recursivo,
```



```
20 else:
21     insert_recursivo(T.root,T,lista)
22
23 #insert recursivo,
24 def insert_recursivo (L,parent,element):
25     if not L:
26         newnode=TrieNode()
27         newnode.key=element[0]
28         del element[0]
29         newnode.parent=parent
30         L.append(newnode)
31         if element:
32             Lista=[]
33             newnode.children=Lista
34             insert_recursivo(Lista,newnode,element)
35             return
36         else:
37             newnode.isEndOfWord=True
38             return
39
40 posicion=buscar_node(L,element[0])
41 if posicion==None:
42     newnode=TrieNode()
```

```
main.py x trie.py x +
trie.py
39
40 posicion=buscar_node(L,element[0])
41 if posicion==None:
42     newnode=TrieNode()
43     newnode.key=element[0]
44     del element[0]
45     newnode.parent=parent
46     L.append(newnode)
47 if element:
48     Lista=[]
49     newnode.children=Lista
50     insert_recursivo(Lista,newnode,element)
51     return
52 else:
53     newnode.isEndOfWord=True
54     return
55 else:
56 if len(element)==1:
57     L[posicion].isEndOfWord=True
58 elif L[posicion].isEndOfWord==True:
59     Lis=[]
60     L[posicion].children=Lis
61     del element[0]
```

Ln 221, Col 30 History

```
main.py x trie.py x +
trie.py
58 elif L[posicion].isEndOfWord==True:
59     Lis=[]
60     L[posicion].children=Lis
61     del element[0]
62     insert_recursivo(L[posicion].children,L[posicion],element)
63 else:
64     del element[0]
65     insert_recursivo(L[posicion].children,L[posicion],element)
66
67
68
69 ###buscar la key de un nodo, dentro de una lista de nodos. devuelve la
posicion en la lista
70 def buscar_node(L,key):
71     for i in range (0,len(L)):
72         if L[i].key==key:
73             return i
74     return None
75
76 def search(T,element):
77     if T.root==None: return False
78     lista=list(element)
79     return search_recursivo(T.root,lista)
--
```

Ln 221, Col 30 History

```
main.py x trie.py x +
trie.py
81
82
83 ###search recursivo
84 def search_recursivo(L,element):
85     indice=buscar_node(L,element[0])
86     #en caso de que no se encuentre la letra con la que sigue la palabra en
la lista, es False
87     if indice==None:
88         return False
89     else:
90         del element[0]
91         if element:
92             if L[indice].children==None:
93                 #en caso de que la palabra tenga más letras pero el node no
tenga más hijos, es False
94                 return False
95             else:
96                 #en caso de que a la palabra le queden letras y el node tenga
hijos, se sigue buscando
97                 return search_recursivo(L[indice].children,element)
98         else:
99             #si la palabra se termino, se evalua si el nodo marca un EndOfWord
100             if L[indice].isEndOfWord: return True
Ln 221, Col 30 History
```

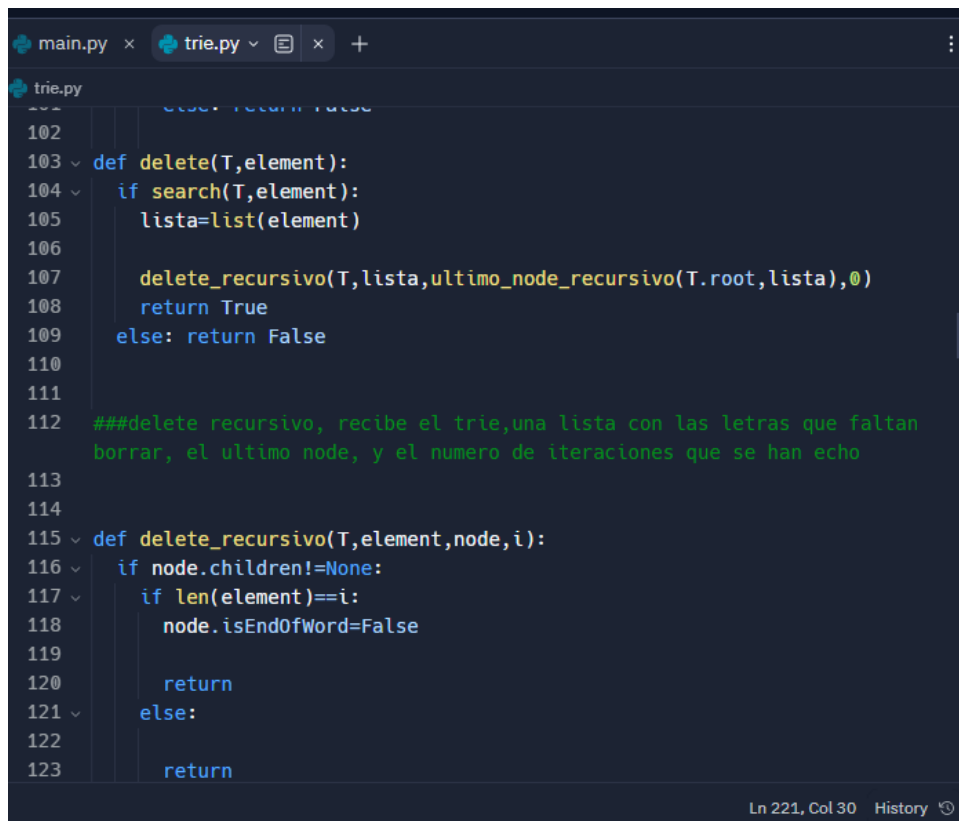
```
main.py x trie.py x +
trie.py
85     indice=buscar_node(L,element[0])
86     #en caso de que no se encuentre la letra con la que sigue la palabra en
la lista, es False
87     if indice==None:
88         return False
89     else:
90         del element[0]
91         if element:
92             if L[indice].children==None:
93                 #en caso de que la palabra tenga más letras pero el node no
tenga más hijos, es False
94                 return False
95             else:
96                 #en caso de que a la palabra le queden letras y el node tenga
hijos, se sigue buscando
97                 return search_recursivo(L[indice].children,element)
98         else:
99             #si la palabra se termino, se evalua si el nodo marca un EndOfWord
100             if L[indice].isEndOfWord: return True
101             else: return False
102
103 def delete(T,element):
104     if search(T,element):
105         lista=list(element)
Ln 221, Col 30 History
```

Ejercicio 2

Sabiendo que el orden de complejidad para el peor caso de la operación `search()` es de $O(m |\Sigma|)$. Proponga una versión de la operación `search()` cuya complejidad sea $O(m)$.

Una forma de implementar un `search` para tener complejidad $O(m)$ sería ingresando a los nodos del Trie en arrays de longitud k (k siendo la cantidad de caracteres del código ASCII), donde cada key tendría un lugar específico en el array, entonces a la hora usar `search` no tendríamos que recorrer las listas, si no buscar en el lugar fijado de cada key, y el orden de complejidad solo estaría determinado por la longitud de la palabra a buscar

Ejercicio 3



```
main.py x trie.py x +
trie.py
102
103 def delete(T,element):
104     if search(T,element):
105         lista=list(element)
106
107         delete_recursivo(T,lista,ultimo_node_recursivo(T.root,lista),0)
108         return True
109     else: return False
110
111
112 ###delete recursivo, recibe el trie,una lista con las letras que faltan
113 borrar, el ultimo node, y el numero de iteraciones que se han echo
114
115 def delete_recursivo(T,element,node,i):
116     if node.children!=None:
117         if len(element)==i:
118             node.isEndOfWord=False
119
120             return
121         else:
122
123             return
```

Ln 221, Col 30 History

```
main.py x trie.py x +
trie.py
115 def delete_recursivo(T,element,node,i):
116     if node.children!=None:
117         if len(element)==i:
118             node.isEndOfWord=False
119
120             return
121         else:
122
123             return
124     else:
125         if node.isEndOfWord==True and i!=0:
126
127             return
128         else:
129             element.pop()
130             newnode=node.parent
131             if newnode==T:
132                 if len(T.root)==1:
133                     T.root=None
134                     return
135                 else:
136                     T.root.remove(node)
137
```

```
main.py x trie.py x +
trie.py
134         return
135     else:
136         T.root.remove(node)
137
138         return
139     else:
140         if len(newnode.children)==1:
141             newnode.children=None
142
143         else:
144             newnode.children.remove(node)
145
146         delete_recursivo(T,element,newnode,i+1)
147
148
149
150
151 ##devuelve el ultimo nodo de una palabra, en un trie
152 def ultimo_node_recursivo(L,element):
153     caracter=element.pop(0)
154     indice=buscar_node(L,caracter)
155     if element:
156         return ultimo_node_recursivo(L[indice].children,element)
Ln 221, Col 30 History
```

```
main.py x trie.py x +
trie.py
143 else:
144     newnode.children.remove(node)
145
146     delete_recursivo(T,element,newnode,i+1)
147
148
149
150
151 ##devuelve el ultimo nodo de una palabra, en un trie
152 def ultimo_node_recursivo(L,element):
153     caracter=element.pop(0)
154     indice=buscar_node(L,caracter)
155     if element:
156         return ultimo_node_recursivo(L[indice].children,element)
157     else:
158         return L[indice]
159
160
161 ##ejercicio 4:
162 def palabra_p_n(T,p,n):
163     pre=list(p)
164     m=n-len(pre)
165     current_lista=T.root
166     if len(pre)==n:
```

Ejercicio 4

```
main.py x trie.py x +
trie.py
158     return L[indice]
159
160
161  ##ejercicio 4:
162  def palabra_p_n(T,p,n):
163      pre=list(p)
164      m=n-len(pre)
165      current_lista=T.root
166      if len(pre)==n:
167          if search(T,p):
168              print(pre)
169              return
170      while pre:
171          current_letra=pre.pop(0)
172          indice=buscar_node(current_lista,current_letra)
173          current_lista=current_lista[indice].children
174          if indice==None or current_lista==None:
175              return
176
177      for i in range(0,len(current_lista)):
178
179          palabra_p_n_recursivo(T,current_lista[i],m,0)
180
```

```
main.py x trie.py x +
trie.py
177  for i in range(0,len(current_lista)):
178
179      palabra_p_n_recursivo(T,current_lista[i],m,0)
180
181
182
183
184  def palabra_p_n_recursivo (T,node,n,i):
185      if n==1:
186          if node.isEndOfWord==True:
187              L=[]
188              while node!=T:
189                  L.insert(0,node.key)
190                  node=node.parent
191              print(L)
192              return
193
194      if i==n-2:
195          if node.children!=None:
196              for j in range(0,len(node.children)):
197
198                  if node.children[j].isEndOfWord:
199                      current=node.children[j]
```

```
main.py x trie.py x +
trie.py
196     for j in range(0, len(node.children)):
197
198         if node.children[j].isEndOfWord:
199             current = node.children[j]
200             L = []
201             while current != T:
202                 L.insert(0, current.key)
203                 current = current.parent
204             print(L)
205
206         return
207
208     else:
209         return
210
211 else:
212     if node.children == None:
213         return
214     else:
215         for k in range(0, len(node.children)):
216             palabra_p_n_recursivo(T, node.children[k], n, i+1)
217
218
Ln 221, Col 30 History
```

Ejercicio 5

```
main.py x trie.py x +
trie.py
---
220 ##ejercicio 5:
221 ##orden de complejidad  $O(n^2)$ 
222
223 def sub_arbol(T1, T2):
224     if T1.root == None and T2.root == None:
225         return True
226     if T1.root == None or T2.root == None:
227         return False
228     return sub_arbol_recursivo(T1.root, T2.root)
229
230
231
232 def sub_arbol_recursivo(La, Lb):
233     if La and not Lb: return False
234     if not La and not Lb: return True
235
236     for i in range(0, len(La)):
237         indice = buscar_node(Lb, La[i].key)
238         if indice == None:
239
240             return False
241         else:
242             if La[i].isEndOfWord == True and Lb[indice] == False:
```



```
main.py x trie.py x +
trie.py
239
240     return False
241 else:
242     if La[i].isEndOfWord==True and Lb[indice]==False:
243         return False
244     elif La[i].children!=None:
245         if Lb[indice].children==None:
246             return False
247         else:
248             if
249 sub_arbol_recursivo(La[i].children,Lb[indice].children)==False: return
False
249     return True
250
251
252
253
254 ###ejercicio7:
255
256
257 def cadena_invertida(T):
258     if not T.root: return False
259     return cadena_invertida_rec(T,T.root)
Ln 221, Col 30 History
```

Ejercicio 6

```
main.py x trie.py x +
trie.py
251
252
253
254 ###ejercicio 6:
255
256
257 def cadena_invertida(T):
258     if not T.root: return False
259     return cadena_invertida_rec(T,T.root)
260
261
262
263
264 def cadena_invertida_rec(T,L):
265     for i in range (0,len(L)):
266         if L[i].isEndOfWord:
267             current=L[i]
268             palabra=[]
269             while current!=T:
270                 palabra.append(current.key)
271                 current=current.parent
272             stri="".join(palabra)
273             #print(palabra)
274
Ln 263, Col 1 History
```

```
trie.py
261
262
263
264 def cadena_invertida_rec(T,L):
265     for i in range (0,len(L)):
266         if L[i].isEndOfWord:
267             current=L[i]
268             palabra=[]
269             while current!=T:
270                 palabra.append(current.key)
271                 current=current.parent
272             stri=".".join(palabra)
273             #print(palabra)
274             if search(T,stri):
275                 return True
276         else:
277             if cadena_invertida_rec(T,L[i].children):
278                 return True
279     return False
280
281
282
283
Ln 263, Col 1 History
```

Ejercicio 7

```
main.py x trie.py x +
trie.py
284 ###ejercicio 7:
285
286
287 def autocompletado(T,p):
288     pre=list(p)
289
290     current_lista=T.root
291
292     while pre:
293         current_letra=pre.pop(0)
294         indice=buscar_node(current_lista,current_letra)
295         current_lista=current_lista[indice].children
296         if indice==None or current_lista==None:
297             return
298     node=current_lista[indice].parent
299
300     autocompletado_rec(T,current_lista,node)
301
302
303
304 def autocompletado_rec (T,L,node):
305     for i in range (0,len(L)):
306         if L[i].isEndOfWord:
307             current=L[i]
```

```
main.py x trie.py x +
trie.py
298 node=current_list[current].parent
299
300 autocompletado_rec(T,current_lista,node)
301
302
303
304 def autocompletado_rec (T,L,node):
305     for i in range (0,len(L)):
306         if L[i].isEndOfWord:
307             current=L[i]
308             lista=[]
309             while current!=node:
310                 lista.insert(0,current.key)
311                 current=current.parent
312             print(lista)
313         if L[i].children!=None:
314             autocompletado_rec(T,L[i].children,node)
315
316
```