Algoritmos y Estructura de Datos I

Ejercitación: Árboles Binarios de Búsqueda

Link Repl.it: https://replit.com/@Paulonia/Martinez13866TADArbol#binarytree.py

Ejercicio 1:

binarytree.py

```python
import mylinkedlist
from myqueue import *

class BinaryTree:
    root=None

class BinaryTreeNode:
    key=None
    value=None
    leftnode=None
    rightnode=None
    parent=None
```

search(B,element):

Descripción: Busca un elemento en el TAD árbol binario.

Entrada: el árbol binario B en el cual se quiere realizar la búsqueda (BinaryTree) y el valor del elemento (element) a buscar.

Salida: Devuelve la key asociada a la primera instancia del elemento. Devuelve None si el elemento no se encuentra.

```python
def search(B,element):
    if B.root==None:
        return None
    else:
        key=None
        key=searchValue(B.root,element,key)
        return key
def searchValue(current,element,key):
    #Busca un elemento en el TAD árbol binario
    if element==current.value:
        key=current.key
        return key
    else:
        if current.leftnode!=None:
            key=searchValue(current.leftnode,element,key)
        if current.rightnode!=None:
            key=searchValue(current.rightnode,element,key)
    return key
```

insert(B,element,key)

Descripción: Inserta un elemento con una clave determinada del TAD árbol binario.

Entrada: el árbol B sobre el cual se quiere realizar la inserción (BinaryTree), el valor del elemento (element) a insertar y la clave (key) con la que se lo quiere insertar.

Salida: Si pudo insertar con éxito devuelve la key donde se inserta el elemento. En caso contrario devuelve None.

```
def insert(B,element,key):
  #Inserta un elemento con una clave determinada
  newNode=BinaryTreeNode()
  newNode.value=element
  newNode.key=key
  if B.root==None:
    B.root=newNode
  else:
    keyInsert=None
    keyInsert=insertNode(newNode,B.root,keyInsert)
    return keyInsert
def insertNode(newNode,current,keyInsert):
  if newNode.key>current.key:
    if current.rightnode!=None:
      insertNode(newNode,current.rightnode,keyInsert)
    else:
      current.rightnode=newNode
      newNode.parent=current
  elif newNode.key<current.key:
    if current.leftnode!=None:
      insertNode(newNode,current.leftnode,keyInsert)
    else:
      current.leftnode=newNode
      newNode.parent=current
  else:
    return None
  return newNode.key
```

delete(B,element)

Descripción: Elimina un elemento del TAD árbol binario.
Poscondición: Se debe desvincular el Node a eliminar.
Entrada: el árbol binario B sobre el cual se quiere realizar la eliminación (BinaryTree) y el valor del elemento (element) a eliminar.
Salida: Devuelve clave (key) del elemento a eliminar. Devuelve None si el elemento a eliminar no se encuentra.

```
def delete(B,element):
  #Elimina un elemento del TAD árbol binario
  if B.root==None:
    return None
  elif B.root.rightnode==None and B.root.leftnode==None:
    if B.root.value==element:
      key=B.root.key
      B.root=None
      return key
    else:
      return None
  else:
    key=search(B,element)
    if key!=None:
      deletedKey=deleteKey(B,key)
      return deletedKey
    else:
      return None
```

deleteKey(B,key)

Descripción: Elimina una clave del TAD árbol binario.
Poscondición: Se debe desvincular el Node a eliminar.
Entrada: el árbol binario B sobre el cual se quiere realizar la eliminación (BinaryTree) y el valor de la clave (key) a eliminar. Salida: Devuelve clave (key) a eliminar. Devuelve None si el elemento a eliminar no se encuentra.

```
81 def deleteKey(B,key):
82     #Elimina una clave del TAD árbol binario
83     if B.root==None:
84         return None
85     elif B.root.rightnode==None and B.root.leftnode==None:
86         if B.root.key==key:
87             B.root=None
88             return key
89         else:
90             return None
91     else:
92         current=B.root
93         deletedKey=None
94         deletedKey=deleteNodebyKey(B,current,key,deletedKey)
95         return deletedKey
96 def deleteNodebyKey(B,current,key,deletedKey):
97     #Busco el nodo a borrar
98     if key>current.key:
99         if current.rightnode!=None:
100            deletedKey=deleteNodebyKey(B,current.rightnode,key,deletedKey)
101    elif key<current.key:
102        if current.leftnode!=None:
103            deletedKey=deleteNodebyKey(B,current.leftnode,key,deletedKey)
104    elif key==current.key:
105        deletedKey=current.key
106        #Caso 1: El nodo a borrar es la raiz
107        newNode=BinaryTreeNode()
108        newNode=current
109        if key==B.root.key:
110            newNode=buscarMenordeMayores(current.rightnode)
111            newNode.leftnode=current.leftnode
112            newNode.rightnode=current.rightnode
113            B.root=newNode
114        #Caso 2: El nodo a borrar tiene un hijo a la izquierda
115        elif current.rightnode==None and current.leftnode!=None:
116            if current.parent.rightnode==current:
117                current.parent.rightnode=current.leftnode
118            else:
119                current.parent.leftnode=current.leftnode
```

```
    #Caso 3: El nodo a borrar tiene un hijo a la derecha
    elif current.rightnode!=None and current.leftnode==None:
      if current.parent.rightnode==current:
        current.parent.rightnode=current.rightnode
      else:
        current.parent.leftnode=current.rightnode
    #Caso 4: El nodo a borrar tiene dos hijos
    elif current.rightnode!=None and current.leftnode!=None:
      newNode=buscarMenordeMayores(current.rightnode)
      #Borrar nodo
      newNode.leftnode=current.leftnode
      newNode.rightnode=current.rightnode
      if current.parent.rightnode==current:
        current.parent.rightnode=newNode
      else:
        current.parent.leftnode=newNode
    #Caso 5: El nodo a borrar no tiene hijos (es una hoja)
    else:
      if current.parent.rightnode==current:
        current.parent.rightnode=None
      else:
        current.parent.leftnode=None
  return deletedKey

def buscarMenordeMayores(newNode):
  #Busca el Menor de los Mayores para reemplazar en el nodo a borrar
  if newNode.leftnode!=None:
    newNode=buscarMenordeMayores(newNode.leftnode)
  #Llega al menor de los mayores
  if newNode.leftnode==None:
    newNode.key=newNode.key
    newNode.value=newNode.value
    #Borrar nodo de abajo
    if newNode.parent.rightnode==newNode:
      newNode.parent.rightnode=None
    else:
      newNode.parent.leftnode=None
  return newNode

```

access(B,key)

Descripción: Permite acceder a un elemento del árbol binario con una clave determinada.
Entrada: El árbol binario (BinaryTree) y la key del elemento al cual se quiere acceder.
Salida: Devuelve el valor de un elemento con una key del árbol binario, devuelve None si no existe elemento con dicha clave.

```
159 def access(B,key):
160     #Permite acceder a un elemento del árbol binario con una clave determinada
161     current=B.root
162     element=None
163     element= accessValue(current,key,element)
164     return element
165 def accessValue(current,key,element):
166     if current.key==key:
167         element=current.value
168         return element
169     if key>current.key:
170         if current.rightnode!=None:
171             element=accessValue(current.rightnode,key,element)
172     elif key<current.key:
173         if current.leftnode!=None:
174             element=accessValue(current.leftnode,key,element)
175     return element
```

update(L,element,key)

Descripción: Permite cambiar el valor de un elemento del árbol binario con una clave determinada. Entrada: El árbol binario (BinaryTree) y la clave (key) sobre la cual se quiere asignar el valor de element.
Salida: Devuelve None si no existe elemento para dicha clave. Caso contrario devuelve la clave del nodo donde se hizo el update.

```
177 def update(B,element,key):
178     #Permite cambiar el valor de un elemento del árbol binario con una clave determinada
179     newNode=BinaryTreeNode()
180     newNode.value=element
181     newNode.key=key
182     if B.root==None:
183         return None
184     else:
185         keyUp=None
186         keyUp=updateNode(newNode,B.root,keyUp)
187         return keyUp
188 def updateNode(newNode,current,keyUp):
189     if current.key==newNode.key:
190         current.value=newNode.value
191         keyUp=newNode.key
192         return keyUp
193     elif newNode.key>current.key:
194         if current.rightnode!=None:
195             keyUp=updateNode(newNode,current.rightnode,keyUp)
196     elif newNode.key<current.key:
197         if current.leftnode!=None:
198             keyUp=updateNode(newNode,current.leftnode,keyUp)
199     return keyUp
```

Ejercicio 2:

traverseInOrder(B)

Descripción: Recorre un árbol binario en orden
Entrada: El árbol binario (BinaryTree)
Salida: Devuelve una lista (LinkedList) con los elementos del árbol en orden. Devuelve None si el árbol está vacío.

```
203 def traverseInOrder(B):
204     #Recorre un árbol binario en orden
205     L=mylinkedlist.LinkedList()
206     recorridoInOrder(L,B.root)
207     return L
208 def recorridoInOrder(L,current):
209     if current!=None:
210         if current.leftnode!=None:
211             recorridoInOrder(L,current.leftnode)
212         mylinkedlist.insert(L,current.value,mylinkedlist.length(L))
213         if current.rightnode!=None:
214             recorridoInOrder(L,current.rightnode)
215
```

traverseInPostOrder(B)

Descripción: Recorre un árbol binario en post-orden

Entrada: El árbol binario (BinaryTree)

Salida: Devuelve una lista (LinkedList) con los elementos del árbol en post-orden. Devuelve None si el árbol está vacío.

```
216 def traverseInPostOrder(B):
217     #Recorre un árbol binario en post-orden
218     L=mylinkedlist.LinkedList()
219     recorridoPostOrder(L,B.root)
220     return L
221 def recorridoPostOrder(L,current):
222     if current!=None:
223         if current.leftnode!=None:
224             recorridoPostOrder(L,current.leftnode)
225         if current.rightnode!=None:
226             recorridoPostOrder(L,current.rightnode)
227         mylinkedlist.insert(L,current.value,mylinkedlist.length(L))
228
```

traverseInPreOrder(B)

Descripción: Recorre un árbol binario en pre-orden

Entrada: El árbol binario (BinaryTree)

Salida: Devuelve una lista (LinkedList) con los elementos del árbol en pre-orden. Devuelve None si el árbol está vacío.

```
229 def traverseInPreOrder(B):
230     #Recorre un árbol binario en pre-orden
231     L=mylinkedlist.LinkedList()
232     recorridoPreOrder(L,B.root)
233     return L
234 def recorridoPreOrder(L,current):
235     if current!=None:
236         mylinkedlist.insert(L,current.value,mylinkedlist.length(L))
237         if current.leftnode!=None:
238             recorridoPreOrder(L,current.leftnode)
239         if current.rightnode!=None:
240             recorridoPreOrder(L,current.rightnode)
241
```

traverseBreadFirst(B)

Descripción: Recorre un árbol binario en modo primero anchura/amplitud

Entrada: El árbol binario (BinaryTree)

Salida: Devuelve una lista (LinkedList) con los elementos del árbol ordenados de acuerdo al modo primero en amplitud. Devuelve None si el árbol está vacío.

```
242 def traverseBreadFirst(B):
243     #Recorre un árbol binario en modo primero anchura/amplitud
244     L=mylinkedlist.LinkedList()
245     if B.root!=None:
246         LR=mylinkedlist.LinkedList()
247         enqueue(L,B.root)
248         while L.head!=None:
249             current=dequeue(L)
250             enqueue(LR,current.value)
251             if current.leftnode!= None:
252                 enqueue(L,current.leftnode)
253             if current.rightnode != None:
254                 enqueue(L,current.rightnode)
255         while LR.head!=None:
256             aux=dequeue(LR)
257             mylinkedlist.insert(L,aux,mylinkedlist.length(L))
258     return L
```

Resultado Unittest:

```
----------------
Ran 10 tests in 0.005s

OK
```