Trabajo Práctico: Algoritmos de búsqueda y ordenamiento

Estructuras de datos
Comisión A (Turno Mañana)
Segundo cuatrimestre de 2020

Docentes:	Ariel Clocchiatti Sergio González Román García
Alumnos:	Kener, Sebastián

Introducción

En este trabajo nos basaremos en la utilización de los TDA pilas y colas para la resolución del problema planteado. Se pidió modelar el funcionamiento del manejo de los auxilios pedidos por los clientes, en una empresa de remolque y reparación de vehículos.

Primera Parte

Aquí se presentaran diversos elementos necesarios, previos al modelado del algoritmo. Para empezar se modelaron dos TDA, uno de pila y uno de cola.

```
#COLAS
class Queue:
  def __init__(self, startQueue = None):

  def vaciar(self):

  def enqueue(self, elemento):

  def estaVacia(self):

  def dequeue(self):

  def top(self):

  def clone(self):

  def tamanioCola(self):

  def __repr__(self):
```

```
#PILAS
class Stack:
  def __init__(self, startStack = None):

  def vaciar(self):

  def push(self, elemento):

  def estaVacia(self):

  def pop(self):

  def top(self):

  def clone(self):

  def tamanioPila(self):

  def __repr__(self):
```

Tanto las pilas como las colas son tipos de datos diseñados para manejar las listas de una manera particular, si bien tienen métodos idénticos, la diferencia radica en el ingreso y egreso de datos.

- `def __init__(self, startStack = None)` -> crea una lista y la inicializa.
- `def vaciar(self)` -> Vacia la lista del TDA.
- `def enqueue(self, elemento)` -> Es propio de la cola (su traducción es poner en la cola), agrega un elemento en la posición cero de la lista, usando la función insert corre todos los elementos en una posición de manera ascendente.
- `def push(self, elemento)` -> Esta función pertenece a la pila, y agrega un elemento a la lista, pero a diferencia del insert, lo agrega uno tras otro.
- `def estaVacia(self)` -> Revisa si la lista del TDA esta vacía o no, devuelve True o False.
- `def dequeue(self)` -> Devuelve el último elemento de la lista, el primero que ingreso.
- `def pop(self)` -> Devuelve el ultimo de la lista, el ultimo que ingreso.

- `def top(self)->` Devuelve un elemento, copia del último elemento de la lista, el primero que va a salir.
- `def clone(self)->` Devuelve una lista copia de la lista actual.
- `def tamanioPila/Cola(self)->` Nos devuelve el tamaño de la lista, ya sea cola o pila dependiendo el nombre.
- `def __repr__(self)->` Devuelve una cadena de caracteres de la lista.

```
from enum import Enum

class ZonaAuxilio(int, Enum):
  Sur = 0
  Norte = 1
  Este = 2
  Oeste = 3
  CABA = 4

class TipoAuxilio(int, Enum):
  Remolque = 0
  Reparacion = 1

class EstadoAuxilio(int, Enum):
  Espera = 0
  Aprobado = 1
```

Utilizando la clase **Enum** se crearon tres subclases para poner asociar y manejar mejor ciertas variables.

`ZonaAuxilio->` Indica los valores que se le da a cada zona.

`TipoAuxilio->` Indica los valores que se le da a los auxilios.

`EstadoAuxilio->` Indica los valores que se le al estado que puede tener un auxilio.

Funciones Auxiliares

Estas son funciones que no van definidas dentro de ningún TDA, pero que son necesarias para cumplir funciones dentro de estos.

```
#VALIDACION DE PATENTE

def validarPatente(patente):
  if len(patente) != 6:
    raise Exception("Ingrese las 3 letras y 3 numeros de la Patente sin espacios")
  if patente[0:3].isalpha() and patente[3:6].isdigit():
    return patente.upper()
  else:
    raise Exception("Patente mal ingresada")
```

Utilizada en el TDA Auxilio, verifica que la patente ingresada este dentro de los parámetros, que contenga más de seis caracteres, que los primeros tres sean letras y que los otros tres sean números. Si es correcto devuelve la patente en mayúscula, sino manda un error.

```
def ordenamientoRapido(unaLista):

def ordenamientoRapidoAuxiliar(unaLista,primero,ultimo):

def particion(unaLista,primero,ultimo):
```

Estas tres son las funciones necesarias para un ordenamiento rápido recursivo. Fue puesto solo el nombre de cada una ya que su funcionamiento y explicación no son más que las de un ordenamiento.

Principales TDA

Auxilio

```
class Auxilio:
  def __init__(self, patente, zonaPartida, zonaDestino, tipo, estado):
    self.patente = validarPatente(patente)
    self.zonaPartida = ZonaAuxilio(zonaPartida)
    self.zonaDestino = ZonaAuxilio(zonaDestino)
    self.tipo = TipoAuxilio(tipo)
    self.estado = EstadoAuxilio(estado)

  def __repr__(self):
    cadenaPrint = "Patente:" + str(self.patente) + " Partida:" + self.zonaPartida.name +
    return cadenaPrint
```

El TDA Auxilio tiene dos funciones. Por un lado __init__, que recibe los datos de patente, zonaPartida, zonaDestino, tipo y estado. La patente es validada y guardada en una variable, y los demás datos transformados en enumeraciones y guardadas en variables.
Por otro lado la función __repr__, que devuelve un string con los datos de inicialización.

Oficinas

La función inicializadora, `__init__`, recibe como parámetro el número de interno de la oficina y la cantidad critica. Dentro de esta se le asigna una variable al número de interno, a la cantidad crítica, y se crean dos colas que cada oficina manejara: la cola de remolques y la de reparación.

```
class OficinaAtencion:
  def __init__(self, nroInterno, cantCritica):
    if nroInterno in range(1, 999):
      self.nroInterno = nroInterno
    self.cantCritica = cantCritica
    self.colaRemolques = Queue()
    self.colaReparacion = Queue()
```

```
def recibirAuxilio(self, auxilio):
    if auxilio.tipo == TipoAuxilio(TipoAuxilio.Remolque).value:
      self.colaRemolques.enqueue(auxilio)
      if self.colaRemolques.tamanioCola() > self.cantCritica:
        print("Cantidad de auxilios critica en Remolques")
    elif auxilio.tipo == TipoAuxilio(TipoAuxilio.Reparacion).value:
      self.colaReparacion.enqueue(auxilio)
      if self.colaReparacion.tamanioCola() > self.cantCritica:
        print("Cantidad de auxilios critica en Reparaciones")
```

Recibe como parámetro un auxilio y lo deriva a la cola correspondiente. A través de un “if” se evalúa el tipo de auxilio, comparándolo con el Enum y lo agrega a la cola correspondiente. También evalúa la cantidad de

auxilios en la cola donde se agrega el dato ingresado, y manda un mensaje si se supero la cantidad crítica planteada.

Esta función revisa si la cola remolques esta vacía, si es así te muestra el próximo auxilio en cola de reparación a salir. Sin embargo si hay auxilios en la cola remolques, mostrara el que saldrá de esta.

```
def primerAuxilioAEnviar(self):
  if not self.colaRemolques.estaVacia():
    auxilioAEnviar = self.colaRemolques.top()
  else:
    auxilioAEnviar = self.colaReparacion.top()
  return auxilioAEnviar
```

```
def enviarAuxilio(self, zonaDeGrua):
    auxilio = 0
    for aux in range(len(self.colaRemolques.cola)):
      if ZonaAuxilio(zonaDeGrua).name == self.colaRemolques.cola[aux].zonaPartida.name:
        print(self.colaRemolques.cola[aux].zonaPartida.name)
        auxilio = self.colaRemolques.cola.pop(aux)
        break
      else:
        for aux in range(len(self.colaReparacion.cola)):
          if ZonaAuxilio(zonaDeGrua).name == self.colaReparacion.cola[aux].zonaPartida.name:
            auxilio = self.colaReparacion.cola.pop(aux)
            break
    return auxilio
```

Recibe por parámetro la zona de la grúa y a través de un for revisa la cola de remolques en busca de un auxilio para esa zona. Si no encuentra auxilios en la cola de remolques revisa la cola de reparación. Una vez encontrado el auxilio que coincide con la zona, lo saca de la cola y guarda en una variable que retorna al finalizar la búsqueda.

```
def esCritica(self):
  return self.colaRemolques.tamanioCola() > self.cantCritica or self.colaReparacion.tamanioCola() > self.cantCritica
```

Retorna un booleano. Revisa el tamaño de las colas remolque y reparación y los compara con la cantidad crítica.

```
def buscarAuxilio(self, nroPatente):
    auxilio = 0
    for aux in range(self.colaRemolques.tamanioCola()):
      if self.colaRemolques.cola[aux].patente == nroPatente:
        auxilio = self.colaRemolques.cola[aux]
    for aux in range(self.colaReparacion.tamanioCola()):
      if self.colaReparacion.cola[aux].patente == nroPatente:
        auxilio = self.colaReparacion.cola[aux]
    return auxilio
```

Recibe el número de patente y devuelve el auxilio con ese número. Recorre las colas con un for y si encuentra coincidencia, a través de un if, guarda ese auxilio en una variable que luego es retornada.

Similar a la función anterior, recorre con for las colas en busca de una coincidencia a partir de una patente ingresada. Al encontrarla es sacada de la cola.

```
def eliminarAuxilio(self, nroPatente):
    for aux in range(self.colaRemolques.tamanioCola()):
      if self.colaRemolques.cola[aux].patente == nroPatente:
        auxilio = self.colaRemolques.cola.pop(aux)
    for aux in range(self.colaReparacion.tamanioCola()):
      if self.colaReparacion.cola[aux].patente == nroPatente:
       auxilio = self.colaRemolques.cola.pop(aux)
```

```
def auxiliosPorTipo(self):
    remolques = self.colaRemolques.tamanioCola()
    reparacion = self.colaReparacion.tamanioCola()
    print("Remolques: "+ str(remolques) +"\n"+ "Reparacion: "+ str(reparacion))
```

Guarda el tamaño de las colas remolque y reparación en variables que son devueltas en una cadena de texto.

```
def cantidadTotalDeAuxilios(self):
   cantidad = self.colaRemolques.tamanioCola() + self.colaReparacion.tamanioCola()
   return cantidad
```

Suma el tamaño de las colas (remolque y reparación), lo guarda en una variable y la retorna.

```
def auxiliosEnEspera(self):
    colaAux = []
    for aux in range(self.colaRemolques.tamanioCola()):
      if self.colaRemolques.cola[aux].estado == "Espera":
        cosa = self.colaRemolques.cola[aux]
        colaAux.append(cosa)
    for aux in range(self.colaReparacion.tamanioCola()):
      if self.colaRemolques.cola[aux].estado == "Espera":
        cosa = self.colaReparacion.cola[aux]
        colaAux.append(cosa)
    return len(colaAux)
```

Recorre las colas, con un for, en busca de auxilios en espera, los copia en una variable, y los agrega a una lista interna. Al finalizar devuelve el tamaño de la lista, o sea la cantidad de auxilios en espera.

Recibe la patente, y busca con un for en ambas colas, si encuentra un auxilio que coincida y lo agrega a una lista interna. Finalmente devuelve un booleano que sale de comparar el tamaño de esa lista contra cero.

```
def hayPedidoPara(self, nroPatente):
    lista = []
    for aux in range(self.colaRemolques.tamanioCola()):
      if self.colaRemolques.cola[aux].patente == nroPatente:
        dato = self.colaRemolques.cola[aux]
        lista.append(dato)
    for aux in range(self.colaReparacion.tamanioCola()):
      if self.colaReparacion.cola[aux].patente == nroPatente:
       dato = self.colaReparacion.cola[aux]
       lista.append(dato)
    return len(lista) > 0
```

```
def cambiaDeTipo(self, nroPatente):
    if self.hayPedidoPara(nroPatente):
     for aux in range(self.colaRemolques.tamanioCola()):
      if self.colaRemolques.cola[aux].patente == nroPatente:
        dato = self.colaRemolques.cola.pop(aux)
        self.colaReparacion.enqueue(dato)
        print(dato)
        break
      else:
        for aux in range(self.colaReparacion.tamanioCola()):
          if self.colaReparacion.cola[aux].patente == nroPatente:
            dato = self.colaReparacion.cola.pop(aux)
            self.colaRemolques.enqueue(dato)
            print(dato)
            break
```

A través de un if se fija si hay pedido para la patente ingresada. Si es así recorre una cola remolques en búsqueda del auxilio, si lo encuentra lo saca de esa cola y lo introduce en la otra. Sino recorre la cola reparación y hace el mismo procedimiento.

Edificio

```
class EdificioEmpresa:
  def __init__(self, cantPisos, nroHabitaculos):
    self.cantPisos = cantPisos
    self.nroHabitaculos = nroHabitaculos
    self.Edificio = self.creacion()
```

La función de inicialización recibe la cantidad de pisos y de habitáculos, que irán en cada piso. Estos se ubicaran en variables y se creara una variable Edificio que será una matriz.

Esta función a través de dos recorridos for crea una matriz que representaría al edificio. Con pisos y habitáculos por piso, cada habitáculo estará vacio, representado con None. Finalmente se retorna la matriz.

```
def creacion(self):
    pisos = []
    habitaculos = []
    for i in range(self.cantPisos):
      pisos.append(habitaculos)
      for j in range(self.nroHabitaculos):
        habitaculos.append(None)
    return pisos
```

```
def establecerOficina(self, piso, habitaculo, oficina):
    if self.Edificio[piso][habitaculo] == None:
      self.Edificio[piso][habitaculo] = oficina
    else:
      print("El habitaculo esta ocupado")
```

Recibe los parámetros piso, habitáculo y oficina. Con un If se fija si ese espacio este vacio, si es así instala la oficina, sino da un mensaje de habitáculo ocupado.

Recibe el piso y a través de un for y un if revisa los habitáculos del piso, si hay una oficina se fija si es crítica y aumenta el contador. Finalmente devuelve un entero, la cantidad de oficinas criticas en ese piso.

```
def cantidadDeOficinasCriticas(self, piso):
    cantidad = 0
    for i in range(self.nroHabitaculos):
     if self.Edificio[piso][i] != None:
       if self.Edificio[piso][i].esCritica():
         cantidad = cantidad + 1
    return cantidad
```

```
def oficinaMenosRecargada(self):
   losMenosCriticosXPiso = []
   for i in range(self.cantPisos):
     for j in range(self.nroHabitaculos):
      dato = self.Edificio[i][j].colaRemolques.tamanioCola()
      losMenosCriticosXPiso.append(dato)

   ordenamientoRapido(losMenosCriticosXPiso)
   for x in range(self.cantPisos):
     for y in range(self.nroHabitaculos):
       if self.Edificio[x][y].colaRemolques.tamanioCola() == losMenosCriticosXPiso[0]:
         return x, y
         break
```

Recorre el edificio con dos For, y pone todos los tamaños de las oficinas en una lista. Luego se usa una función de ordenamiento, y se recorre nuevamente el edificio en busca de la oficina que coincida con el tamaño guardado en la lista. Finalmente se retorna el piso y el habitáculo.

```
def buscaOficina(self, nroInterno):
  pisoYOf = []
  for i in range(self.cantPisos):
    for j in range(self.nroHabitaculos):
      if self.Edificio[i][j].nroInterno == nroInterno:
        pisoYOf.append(i)
        pisoYOf.append(j)
        return pisoYOf
```

Se recorre el edificio con dos lazos for y se compra con un if el número de interno de cada oficina con el ingresado. Si coincide se guarda el piso y el habitáculo en una matriz que es retornada.

```
def moverAuxilio(self, nroPatente, internoOficinaOrigen, internoOficinaDestino):
  auxilio = 0
  for i in range(self.cantPisos):
    for j in range(self.nroHabitaculos):
      auxilio = self.Edificio[i][j].buscarAuxilio(nroPatente)
      if auxilio != 0:
        self.Edificio[i][j].eliminarAuxilio(nroPatente)
        coordenadas = self.buscaOficina(internoOficinaDestino)
        self.Edificio[coordenadas[0]][coordenadas[1]].recibeAuxilio(auxilio)
```

Recibe una patente y dos internos de oficinas. Recorre el edificio buscando el auxilio con la patente indicada. Luego lo elimina de la oficina en la que esta, busca la dirección de la oficina a la que se debe agregar, y se agrega a la nueva oficina.