Estructuras de datos nodales: Parte 1

Semana 09 - Jueves 10 de octubre de 2019

Anuncios

1. No olviden contestar la encuesta de carga académica: ¡incentivos!

2. T02. ¿Qué tal?
3. Programatón hoy.
4. Actividad de hoy es formativa. Deben subir una retroalimentación personal.

Encuesta Medio Semestre

Principales sugerencias recibidas

1. Largo y claridad/ ambigüedad de enunciados.
2. Densidad de material de estudio.
3. Más ejercicios propuestos.
4. Formato de ayudantías.

Estructuras de datos nodales: Parte 1

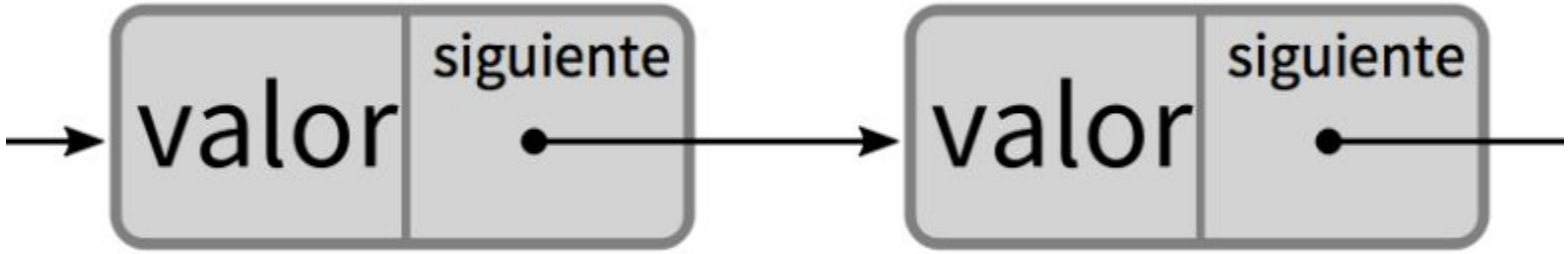
Semana 09 - Jueves 10 de octubre de 2019

Nodo

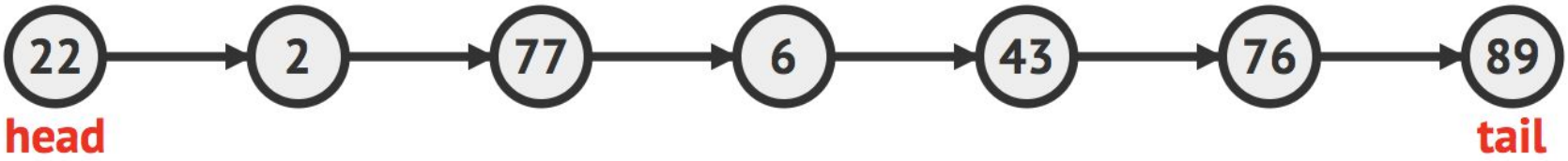
Unidad básica de datos

- Datos
- Vecinos

```python
class Nodo:

    def __init__(self, valor=None):
        """Inicializa la estructura del nodo"""
        self.valor = valor
        self.siguiente = None
```



Listas ligadas



Gracias visualgo.net

Listas ligadas

- Ventajas
  - No necesitamos conocer *a priori* la cantidad de elementos.
  - Inserciones rápidas.
- Desventajas
  - Acceso secuencial.
  - No podemos retroceder al recorrerla.

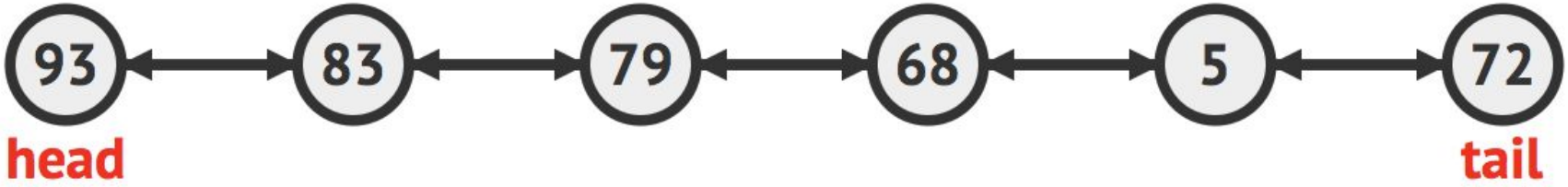
```
class LineaDeMetro:

    def __init__(self):
        self.terminal_inicio = None
        self.terminal_final = None

    def agregar_estacion(self, nombre):
        nueva_estacion = Estacion(nombre)
        if self.terminal_final is not None:
            self.terminal_final.siguiente_estacion = nueva_estacion
            self.terminal_final = self.terminal_final.siguiente_estacion
        else:
            self.terminal_inicio = nueva_estacion
            self.terminal_final = self.terminal_inicio
```

```python
class LineaDeMetro:

    def insertar_estacion(self, nombre, posicion):
        nueva_estacion = Estacion(nombre)
        nodo_actual = self.terminal_inicio

        # Caso particular: insertar en la cabeza
        if posicion == 0:
            # Actualizamos la cabeza
            nueva_estacion.siguiente_estacion = self.terminal_inicio
            self.terminal_inicio = nueva_estacion
            # Caso más particular. Si la lista estaba vacia, actualizamos la cola
            if nueva_estacion.siguiente_estacion is None:
                self.terminal_final = nueva_estacion
            return

        # Buscamos el nodo predecesor
        for i in range(posicion - 1):
            if nodo_actual is not None:
                nodo_actual = nodo_actual.siguiente_estacion

        # Si encontramos el predecesor, actualizamos las referencias
        if nodo_actual is not None:
            # Si no usamos este orden perdemos la referencia al resto de la lista ligada
            nueva_estacion.siguiente_estacion = nodo_actual.siguiente_estacion
            nodo_actual.siguiente_estacion = nueva_estacion
            # Otro caso particular: si es que insertamos en la última posición
            if nueva_estacion.siguiente_estacion is None:
                self.terminal_final = nueva_estacion
```

¿Cómo podemos modelar esto?

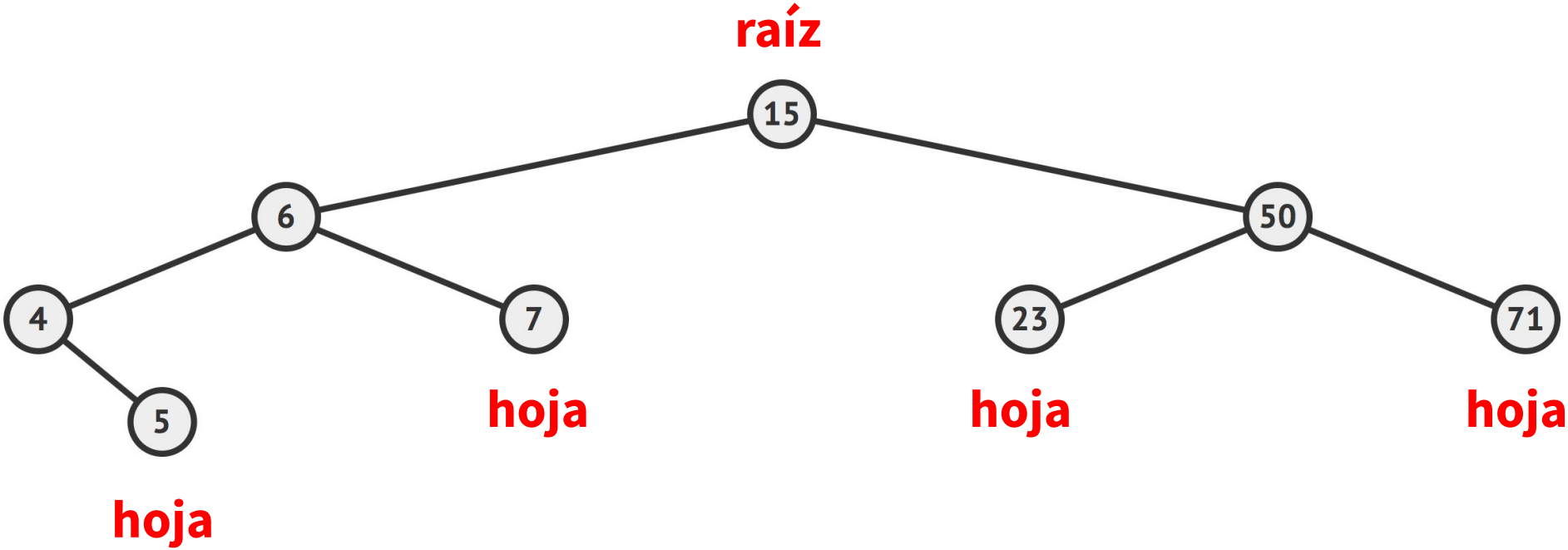
```
L5 = LineaDeMetro()
L5.insertar_estacion(“San Joaquín”, 0)
L5.agregar_estacion(“Carlos Valdovinos”) # Se inserta al final
L5.insertar_estacion(“Rodrigo de Araya”, 2)
L5.insertar_estacion(“Camino Agrícola”, 1)
L5.insertar_estacion(“Ñuble”, 4)
# El orden de las estaciones es: San Joaquín, Camino Agrícola,
# Carlos Valdovinos, Rodrigo de Araya y Ñuble

metro = Metro() # Supongamos que esto representa un metro
L5.iniciar_recorrido(metro)
# Con esto, nuestro metro puede recorrer todas las estaciones,
# pero queremos que en el terminal_final se devuelva

# ¿Cómo podemos referenciar las estaciones en el sentido opuesto?
```

Una variante: lista doblemente ligada



Gracias visualgo.net

Árboles



Gracias visualgo.net

Árboles

Una estructura naturalmente recursiva

- BFS *versus* DFS
- Árbol binario
- ¿Posibles usos?

Recursión *versus* iteración

- Ventajas
- Desventajas

```
class Mafioso:

    jerarquia = {
        “Don”: “Capo”, # Don es el jefe de la mafia
        “Capo”: “Soldado”, # Capo esta en un segundo nivel, bajo el Don
        “Soldado”: None, # Soldado es el nivel más bajo
    }

    def __init__(self, nombre, tipo, seccion):
        self.nombre = nombre
        self.tipo = tipo # Don, Capo o Soldado
        self.seccion = seccion
        self.hijos = [] # Subordinados (“hijos” para entender el código)

    def agregar_miembro(self, miembro):
        if miembro.tipo == Mafioso.jerarquia[self.tipo]:
            self.hijos.append(miembro)
        else:
            # Elegir un hijo
            # Agregar a ese hijo
```

¿Cómo podemos usar este código?

```
# Partimos agregando al Don, el jefe ...
mafia_iic2233 = Mafioso("Cristian Ruz", "Don", seccion=1)
# ... luego a los Capos, ...
capo_seccion_2 = Mafioso("Fernando Florenzano", "Capo", seccion=2)
capo_seccion_3 = Mafioso("Antonio Ossa", "Capo", seccion=3)
capo_seccion_4 = Mafioso("Vicente Domínguez", "Capo", seccion=4)
mafia_iic2233.agregar_miembro(capo_seccion_2)
mafia_iic2233.agregar_miembro(capo_seccion_3)
mafia_iic2233.agregar_miembro(capo_seccion_4)
# ... y a los Soldados
enzo = Mafioso("Enzo Tamburini", "Soldado", seccion=4)
mafia_iic2233.agregar_miembro(enzo)
# Para saber a cuál hijo tenemos que elegir (recuerden la función para
# insertar), tenemos que ocupar la sección
```

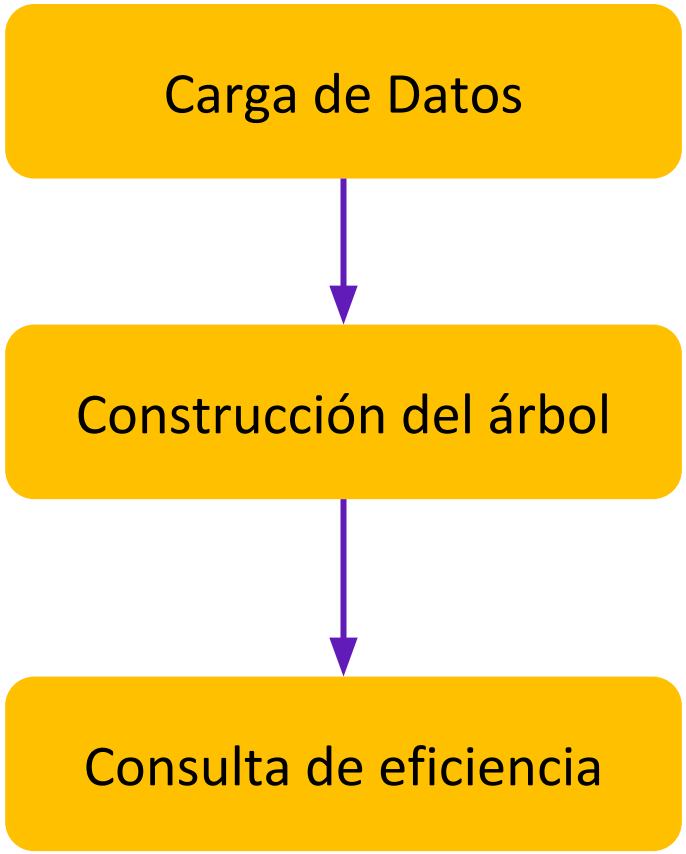
```
Class Mafioso:

    ...

    def agregar_miembro(self, miembro):
        if miembro.tipo == Mafioso.jerarquia[self.tipo]:
            self.hijos.append(miembro)
        else:
            # Elegir un hijo
            hijo = self.elegir_hijo(miembro.seccion)
            # Agregar a ese hijo
            hijo.agregar_miembro(miembro)

    def elegir_hijo(self, seccion):
        for hijo in self.hijos:
            if hijo.seccion == seccion:
                # Retornamos el primer hijo de esa seccion
                return hijo
```

Actividad

1. En el *syllabus*, vayan a la carpeta “Actividades” y descarguen el enunciado de la actividad 7 (AC07) https://github.com/IIC2233/syllabus
2. Trabajen **individualmente** hasta las 16:30.
3. Recuerden hacer *commit* y *push* cada cierto tiempo.

Cierre

Diagrama de flujo de AC




```
ayudantes = read_file()
```

```
cuerpo_ayudantes =
instanciar_cuerpo_ayudantes(ayudantes)
```

```
enzini.agregar_ayudante(ayudante)
```

```
ayudantes, grupo_ganador =
grupo_mayor_eficiencia(cuerpo_ayudantes)
```

```
imprimir_grupo(ayudante)
```

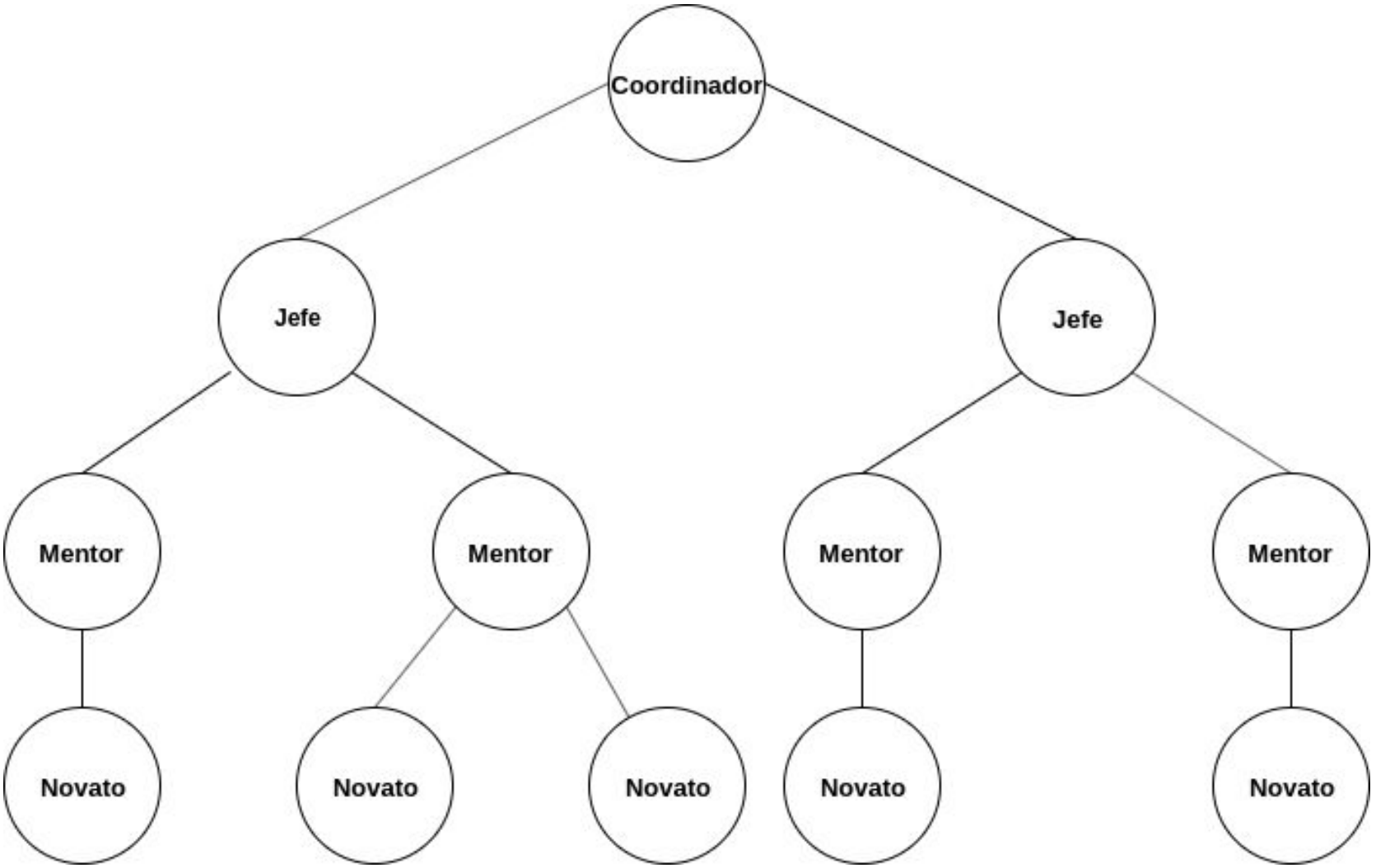
Estructura de Árbol

¿Qué ventajas tiene utilizar una estructura de árbol?

La estructura recursiva del árbol nos permite construir métodos eficientes.

No necesitamos iterar por cada nodo del árbol. Podemos aprovechar su construcción.

Recordemos la composición del árbol



agregar_ayudante(self, ayudante)

```
def agregar_ayudante(self, ayudante):

  if self.rango == “Coordinador”:
    if ayudante.rango == “Jefe”:
      self.subordinados.append(ayudante)    # Base
    else:
      for jefe in self.subordinados:
        if jefe.tipo == ayudante.tipo:
          jefe.agregar_ayudante(ayudante)  # Recursión

  ... …
```

Cada **Ayudante** es la **raíz de un árbol** donde sus hijos también son **Ayudantes**. Al insertar un hijo empezamos desde la raíz y descendemos hasta el nivel apropiado.

agregar_ayudante(self, ayudante)

```
def agregar_ayudante(self, ayudante):
  ... ...
  elif self.rango == “Jefe”:
    if ayudante.rango == “Mentor”:
      self.subordinados.append(ayudante)   # Base
    else:
      mentor_adecuado = None
      for mentor in self.subordinados:
          # Elegir mentor más adecuado
          ... ...
      mentor_adecuado.agregar_ayudante(ayudante)
```

Si soy un Jefe y debo agregar un Mentor, lo agrego a mi lista.
Si debo agregar un Novato, entonces se lo paso al Mentor más adecuado.

agregar_ayudante(self, ayudante)

```
def agregar_ayudante(self, ayudante):
  ... ...
  elif self.rango == “Mentor”:
      self.subordinados.append(ayudante)
```

Caso fácil. Agregar una hoja (Novato) al último nivel intermedio (Mentor).

`grupo_mayor_eficiencia(coordinador)`

Recorrer todos los nodos y sumar eficiencias

¿Qué tipo de recorrido debemos hacer?

¿BFS ó DFS?

grupo_mayor_eficiencia(coordinador)

```
# Recorrido DFS (también podría haber sido BFS)
stack = [coordinador]
while len(stack) > 0:
  nodo = stack.pop()


    ... ... # aquí hacemos algo con el nodo

  # agregamos los próximos a visitar al stack
  stack.extend(nodo.subordinados)
```

Recorremos el árbol usando su estructura. Podemos transformar el recorrido a BFS usando una cola en lugar de un stack.

grupo_mayor_eficiencia(coordinador)

```
p = {“Coordinador”:4, “Jefe”:3, “Mentor”:2, “Novato”:1 }
e_tareos = 0
e_docencios = 0
stack = [coordinador]
while len(stack) > 0:
  nodo = stack.pop()
  eficiencia = p[nodo.rango] * nodo.eficiencia
  if nodo.tipo == “Tareo”: e_tareos += eficiencia
  elif nodo.tipo == “Docencio”: e_docencios += eficiencia
  else:
    e_tareos += eficiencia
    e_docencios += eficiencia
  stack.extend(nodo.subordinados)
```

Recorremos el árbol usando su estructura. Podemos transformar el recorrido a BFS usando una cola en lugar de un stack.

grupo_mayor_eficiencia(coordinador)

```
... ... ...
if e_tareos > e_docencios:
  imprimir_grupo(coordinador.subordinados[0])
else:
  imprimir_grupo(coordinador.subordinados[1])
```

Imprimir el grupo que corresponda.

imprimir_grupo

```
# Recorrido BFS
cola = [principal]
while len(cola) > 0:
  nodo = cola.pop(0)
  ... ... # aquí hacemos algo con el nodo
  # agregamos los próximos a visitar al stack
  cola.extend(nodo.subordinados)
```

Recorremos el árbol por niveles usando BFS.

imprimir_grupo

```
a_imprimir = defaultdict(list)

# Recorrido BFS
cola = [principal]
while len(cola) > 0:
  nodo = cola.pop(0)
  a_imprimir[nodo.rango].append(nodo.nombre)
  # agregamos los próximos a visitar al stack
  cola.extend(nodo.subordinados)

# imprimir cada lista del diccionario a_imprimir
```

Recorremos el árbol por niveles usando BFS.

Próxima semana

1. Pasen a la Programatón desde las 18:00 en el Hall de Estudiantes.
2. Hay AC formativa, sobre la parte 2 sobre estructuras de datos nodales.

¡Recuerden responder la auto-evaluación!