DOCUMENTO DE ANÁLISIS

Juan David Salguero, 201923136, J.salguero@uniandes.edu.co
David Molina, 202125176, d.molinad@uniandes.edu.co

En esta observación se realiza un análisis de la complejidad teórica de cada requerimiento para el Reto 3 en notación O, y su justificación.

Análisis de complejidad

Requerimiento 1

Complejidad O(1)

Lógica de búsqueda: ***Mapa>Arbol>Lista***

Como analista FIFA Deseo conocer las cinco últimas adquisiciones de jugadores de un club específico.

Gracias a un sistema de índices amplio, en la carga de datos se guarda un árbol de jugadores por cada club en un mapa, como se muestra a continuación. En este mapa la llave de cada nodo es el nombre del club, y su valor un árbol con todos los jugadores que pertenecen a el club respectivo. De esta manera, dado un club, se puede acceder a un filtro de jugadores por club en O(1).

```
def addPlayer(analyzer, player):
    neoPlayer = newPlayer(player)
    lt.addLast(analyzer['players'], neoPlayer)

    # ____________________________________________
    # Indice por Culbs
    # ____________________________________________
    if not mp.contains(analyzer['clubIndex'], neoPlayer['club_name'].lower()):
        # se crea arbol
        playerTree = om.newMap(omaptype='RBT', comparefunction=compareDates)
        updateDateIndex(playerTree, neoPlayer)
        mp.put(analyzer['clubIndex'], neoPlayer['club_name'].lower(),
               playerTree)
        mp.put(analyzer['clubIndex_num'], neoPlayer['club_name'].lower(), 1)

    elif mp.contains(analyzer['clubIndex'], neoPlayer['club_name'].lower()):
        key = mp.get(analyzer['clubIndex'], neoPlayer['club_name'].lower())
        value = me.getValue(key)
        # Value = Arbol
        key1 = mp.get(analyzer['clubIndex_num'],
                      neoPlayer['club_name'].lower())
        value1 = me.getValue(key1)
        value1 += 1
        mp.put(analyzer['clubIndex_num'], neoPlayer['club_name'].lower(),
               value1)
        updateDateIndex(value, neoPlayer)
    #
```

Por su parte, cada árbol dentro de cada nodo del mapa es de tipo RBT, y organiza a los jugadores según su fecha de unión al club. Un club no debería tener más de 30 jugadores, por lo que en un peor caso debería no haber más de 30 fechas (llaves) en el árbol, en el caso de que no se repita ninguna.

Por ello, encontrar una fecha en específico dentro de un club cuesta O(1) + O(Altura del árbol).

```
def updateDateIndex(map, player):
    """
    CHECKEA que la fecha de cada jugador no se repita por club
    y añade cada fecha/jugador al arbol
    """
    occurreddate = player['club_joined']
    entry = om.get(map, occurreddate)

    if entry is None:
        listPlayers = lt.newList("ARRAY_LIST", cmpfunction=compareIds)
        lt.addLast(listPlayers, player)
        om.put(map, occurreddate, listPlayers)
    else:
        valor = me.getValue(entry)
        lt.addLast(valor, player)

    return map
```

En el caso de que 2 jugadores del mismo club tengan fechas idénticas, se añaden a una lista dentro del árbol. Al iterar por esta lista, usualmente pequeña, se pueden conseguir los jugadores requeridos, como se muestra a continuación:

```
def LastPlayers(club, analyzer):
    """
    Devuelve las ultimas 5 adquisiciones mas recientes del club
    """
    if mp.contains(analyzer['clubIndex'], club.lower()):
        playerList = lt.newList('ARRAY_LIST')
        key = mp.get(analyzer['clubIndex'], club.lower())
        tree = me.getValue(key)
        nodos = lt.size(om.keySet(tree))
        pos = 1

        while pos != nodos + 1:

            fecha = om.select(tree, nodos-pos)
            value = om.get(tree, fecha)
            players = me.getValue(value)

            for player in lt.iterator(players):
                if lt.size(playerList) < 5:
                    lt.addLast(playerList, player)

                else:
                    break
            pos += 1

        return playerList

    else:
        return None
```

Requerimiento 2

Complejidad (NLog(N))

Lógica de búsqueda: ***Mapa>Tupla>Árbol>Lista***

Como analista FIFA Deseo un reporte que me permita identificar los jugadores que juegan en una posición específica con la finalidad de buscar refuerzos para un club. Sin embargo, el club ha definido que los jugadores seleccionados deben estar dentro de un rango de desempeño, potencial y salario

Para este requerimiento se usa un índice que guarda tres árboles de jugadores por cada posición guardada en un mapa, como se muestra a continuación. En este mapa la llave de cada nodo es la posición de juego, y su valor una tupla con tres árboles con todos los jugadores que juegan en esa posición.

Los tres árboles RBT en cada posición se dividen de la siguiente manera:

Árbol organizado por “overall”

Árbol organizado por “potential”

Árbol organizado por “wage_eur”

```
    # ____________________________________________________
    # Indice por Posicion
    # ____________________________________________________
    for pos in neoPlayer['player_positions'].split(","):
        pos = pos.lower().strip()
        if not mp.contains(analyzer['posIndex'], pos):
            # se crean multiples arboles
            # arbol por "overall"
            playerOverall = om.newMap(omaptype='RBT', comparefunction=compareNumbers)
            updateNumericalIndex("overall", playerOverall, neoPlayer)
            # arbol por "potential"
            playerPotetial = om.newMap(omaptype='RBT', comparefunction=compareNumbers)
            updateNumericalIndex("potential", playerPotetial, neoPlayer)
            # arbol por "wage_eur"
            playerWage = om.newMap(omaptype='RBT', comparefunction=compareNumbers)
            updateNumericalIndex("wage_eur", playerWage, neoPlayer)
            # arbol por "Rep Number" BONO:
            playerRep = om.newMap(omaptype='RBT', comparefunction=compareNumbers)
            updateRepIndex(analyzer["min-max"], playerRep, neoPlayer)
            playerTuple = playerOverall, playerPotetial, playerWage, playerRep
            mp.put(analyzer['posIndex'], pos, playerTuple)

        elif mp.contains(analyzer['posIndex'], pos):
            key = mp.get(analyzer['posIndex'], pos)
            value = me.getValue(key)
            # Value -> Tupla
            updateNumericalIndex("overall", value[0], neoPlayer)
            updateNumericalIndex("potential", value[1], neoPlayer)
            updateNumericalIndex("wage_eur", value[2], neoPlayer)
            updateRepIndex(analyzer["min-max"], value[3], neoPlayer)
```

De esta manera, se pueden crear tres listas generadas de cada árbol según los parámetros dados de desempeño, potencial y salario, con una función om.values():

```
def posPlayers(info, analyzer):
    """
    Retorna la cantidad de jugadores en una posicion, filtrados
    por "overall", "potential" y "wage_eur"
    """
    if mp.contains(analyzer['posIndex'], info[0].lower()):
        key = mp.get(analyzer['posIndex'], info[0].lower())
        # Value -> Tupla con tres arboles
        value = me.getValue(key)
        overallTree = value[0]
        potentialTree = value[1]
        wageTree = value[2]
        overallList = om.values(overallTree, info[1], info[2])
        potentiaList = om.values(potentialTree, info[3], info[4])
        wageList = om.values(wageTree, info[5], info[6])
```

Gracias a las estructuras de organización usadas, el buscar la posición en el mapa y seleccionar valores en específico en cada árbol cuesta: O(1) + O(Altura del árbol Overall) + O(Altura del árbol Potential) + O(Altura del árbol Wage). Para optimizar la búsqueda, se compara el tamaño de cada lista y a partir de la menor, independientemente de su propiedad (overall, potential, wage_eur), se itera sobre cada jugador dentro de ella para filtrarlos según las propiedades de las otras dos listas. Esto tras una organización con merge resulta en la lista final.

```
    minPlayers = overallList
    sortList = lt.newList("ARRAY_LIST")
    minN = 0
    if lt.size(potentiaList) <= lt.size(minPlayers):
        minPlayers = potentiaList
        minN = 1
    if lt.size(wageList) <= lt.size(minPlayers):
        minPlayers = wageList
        minN = 2
    for playerList in lt.iterator(minPlayers):
        for player in lt.iterator(playerList):
            if minN == 0:
                if int(player["potential"]) >= info[3] and int(player["potential"]) <= info[4]:
                    if int(player["wage_eur"]) >= info[5] and int(player["wage_eur"]) <= info[6]:
                        lt.addLast(sortList, player)
            elif minN == 1:
                if int(player["overall"]) >= info[1] and int(player["overall"]) <= info[2]:
                    if int(player["wage_eur"]) >= info[5] and int(player["wage_eur"]) <= info[6]:
                        lt.addLast(sortList, player)
            elif minN == 2:
                if int(player["potential"]) >= info[3] and int(player["potential"]) <= info[4]:
                    if int(player["overall"]) >= info[1] and int(player["overall"]) <= info[2]:
                        lt.addLast(sortList, player)
    sortByPos(sortList)
    return sortList

else:
    return None
```

Debido a que, en promedio, la cantidad de jugadores que juegan en las posiciones más populares pueden ser relativamente grandes, en un peor caso en que los rangos dados de las propiedades sean todos los valores en los mapas, organizar e iterar la lista tiene una complejidad menor a N(LogN). En promedio, sin embargo, se espera una complejidad cercana a O(1).

Requerimiento 3 (Hecho por Juan David Salguero)

Como analista FIFA Deseo un reporte de los jugadores cuyo salario este dentro de un rango y tengan una característica de juego específica (player_tags).

Complejidad O(NLog(N))

Lógica de búsqueda: ***Mapa>Árbol>Lista***

De manera similar a los anteriores requerimientos, se usa un índice que guarda un árbol de jugadores organizados por su salario dentro de cada nodo de un mapa. El mapa tiene como llaves Tags, y sus valores corresponden a un árbol con los jugadores marcados con el mismo.

```
    # ___________________________________________________
    # Indice por Tags
    # ___________________________________________________
    for tag in neoPlayer['player_tags'].split(","):
        tag = tag.strip()
        if tag == "":
            tag = "unknown"
        if not mp.contains(analyzer['tagIndex'], tag):
            # se crea arbol
            playerTree = om.newMap(omaptype='RBT', comparefunction=compareIds)
            updateNumericalIndex("wage_eur", playerTree, neoPlayer)
            mp.put(analyzer['tagIndex'], tag, playerTree)

        elif mp.contains(analyzer['tagIndex'], tag):
            key = mp.get(analyzer['tagIndex'], tag)
            value = me.getValue(key)
            # Value -> Arbol
            updateNumericalIndex("wage_eur", value, neoPlayer)
    #
```

Gracias a el índice creado y al TAD, encontrar los jugadores marcados por un tag cuesta O(1). A esto se le suma la complejidad de encontrar los jugadores dentro de un árbol en un rango de salario O(Altura del árbol).

```
def tagsAndWage(inferior, superior, tag, analyzer):
    tagsmp = analyzer['tagIndex']
    if mp.contains(tagsmp, tag.lower()):
        playerlist = lt.newList("ARRAY_LIST", cmpfunction=compareIds)
        key = mp.get(tagsmp, tag.lower())
        tree = me.getValue(key)
        lista = om.values(tree, inferior, superior)
        for nodo in lt.iterator(lista):
            for player in lt.iterator(nodo):
                lt.addLast(playerlist, player)
        sortByWage(playerlist)
        return playerlist
```

En general se espera una complejidad de búsqueda compleja. No obstante, si el rango dado por el usuario es igual a todos los nodos dentro del árbol, la complejidad resultante sería similar al que se tiene iterando una lista. En promedio en un peor caso la complejidad resultaría menor a O(NLog(N)) Aunque en un funcionamiento promedio la complejidad resultante es O(1).

Requerimiento 4 (Hecho por David Molina)

Como analista FIFA Deseo un reporte de los jugadores nacidos en un rango de fechas y que tengan un rasgo característico específico asociado (particularidades del jugador).

Complejidad O(n Log(n))

Lógica de búsqueda: ***Mapa>Árbol>Lista***

Igual que en el anterior requerimiento, se usa un índice que guarda un árbol de jugadores organizados por su fecha de nacimiento dentro de cada nodo de un mapa. El mapa tiene como llaves Traits, y sus valores corresponden a un árbol con los jugadores marcados con el mismo.

```
    # ____________________________________________
    # Indice por traits
    # ____________________________________________
    for trait in neoPlayer['player_traits'].split(","):
        trait = trait.lower().strip()
        if not mp.contains(analyzer['traitIndex'], trait):
            # se crea arbol
            playerTree = om.newMap(omaptype='RBT', comparefunction=compareDobs)
            updateDobIndex(playerTree, neoPlayer)
            mp.put(analyzer['traitIndex'], trait, playerTree)

        elif mp.contains(analyzer['traitIndex'], trait):
            key = mp.get(analyzer['traitIndex'], trait)
            value = me.getValue(key)
            # Value -> Arbol
            updateDobIndex(value, neoPlayer)
    #
```

Gracias a el índice creado y al TAD, encontrar los jugadores marcados por un trait cuesta O(1). A esto se le suma la complejidad de encontrar los jugadores dentro de un árbol en un rango de salario O(Altura del árbol).

```
def playersbyTrait(trait, inferior, superior, analyzer):
    traitsmp = analyzer['traitIndex']
    if mp.contains(traitsmp, trait.lower()):
        playerlist = lt.newList("ARRAY_LIST", cmpfunction=compareIds)
        key = mp.get(traitsmp, trait.lower())
        tree = me.getValue(key)
        lista = om.values(tree, inferior, superior)
        for nodo in lt.iterator(lista):
            sortForDob(nodo)
            for player in lt.iterator(nodo):
                lt.addLast(playerlist, player)
        return playerlist
```

Esto último y el hecho de que la lista filtrada se debe organizar, a este procedimiento se le estima una complejidad menor a O(n Log(n)), considerando un peor caso en el que rango de fechas dadas sea el total de nodos en el árbol. En promedio, sin embargo, la complejidad es cercana a O(1).

Requerimiento 5

Como analista FIFA Deseo un reporte gráfico, específicamente un histograma, de la cantidad de jugadores dada una propiedad o característica de los jugadores. Para este requerimiento no se permite el uso de librerías.

Complejidad O(1)

Este requerimiento hace uso de un índice de propiedades. En este mapa, se guardan 8 llaves con el nombre de las propiedades posibles con las cuales se construye el histograma. Los valores de estas llaves son arboles, en donde se guardan todos los jugadores según su valor en la propiedad correspondiente. Ejm: Llave("overall") Valor: (Árbol con todos los jugadores organizados por su "overall")

```
        updateDobIndex(value, neoPlayer)
# ____________________________________________________
# Indice por properties
# ____________________________________________________
properties = ["overall", "potential", "value_eur", "wage_eur",
              "age", "height_cm", "weight_kg", "release_clause_eur"]
if not mp.contains(analyzer['propertyIndex'], "overall"):
    # se crean arboles
    for property in properties:
        # Overall
        Tree = om.newMap(omaptype='RBT', comparefunction=compareNumbers)
        updateNumericalIndex(property, Tree, neoPlayer)
        mp.put(analyzer['propertyIndex'], property, Tree)
else:
    for property in properties:
        key = mp.get(analyzer['propertyIndex'], property)
        value = me.getValue(key)
        # Value -> Arbol
        updateNumericalIndex(property, value, neoPlayer)
```

Gracias a que no se requiere encontrar ningún valor en específico dentro del árbol, la complejidad se mantiene en O(1)

Las demás operaciones para la implementación de un histograma, se facilitan con la utilización de funciones como lt.size(), para saber la cantidad de jugadores dentro de las listas de cada árbol sin tener que iterarlas.

```
suma = menor
while lt.size(indices) < N-1:
    intervalo = round(suma + amplitud, 3)
    lt.addLast(indices, intervalo)
    suma = intervalo

lt.addLast(indices, mayor)
lt.addFirst(indices, menor)

for i in range(2, lt.size(indices)+1):

    primer = lt.getElement(indices, i-1)
    segundo = lt.getElement(indices, i)
    contador = 0
    if i > 2:
        valores = om.values(tree, (primer+1), segundo)
    else:
        valores = om.values(tree, ceil(primer), segundo)

    for valor in lt.iterator(valores):
        contador += lt.size(valor)

    lt.addLast(indicesbin, f"({primer}-{segundo}]")
    lt.addLast(indicescount, contador)

for valor in lt.iterator(indicescount):
    lvl = valor//X
    mark = "*"*lvl
    lt.addLast(indiceslvl, lvl)
    lt.addLast(indicesmark, mark)

return indicesbin, indicescount, indiceslvl, indicesmark, mayor, menor
```

Requerimiento 6(Bono)

Como analista FIFA requiero un reporte que me permita identificar un listado de posibles jugadores que puedan sustituir un jugador que próximamente abandonará el club. Los jugadores identificados deben ser lo "más similar" posible en cuanto a potencial, edad, altura, posición de juego (player_positions) y costo.

Complejidad O(1)

Lógica de búsqueda: ***Mapa>Tupla>Árbol>Lista***

En primer lugar se usa un índice por nombres para encontrar el jugador requerido:

```
    # ____________________________________________
    # Indice por short_names
    # ____________________________________________
    name = neoPlayer['short_name'].lower().strip()
    if not mp.contains(analyzer['nameIndex'], name):
        # se crea nodo
        mp.put(analyzer['nameIndex'], name, neoPlayer)

    return analyzer
```

A partir de esto se calcula su valor representativo, al normalizar sus propiedades (potencial, edad, altura y costo) en comparación con el valor máximo y mínimo de cada propiedad en la muestra total de jugadores. Este procedimiento gracias a la naturaleza de los mapas, tiene una complejidad de O(1).

```
# Se normaliza cada propiedad
properties = ["potential", "age", "height_cm", "value_eur"]
representative = 0
for prop in properties:
    minV = analyzer["min-max"][prop][0]
    maxV = analyzer["min-max"][prop][1]
    normal = (int(float(player[prop])) - minV) / (maxV - minV)
    representative += normal
```

Los valores mínimos y máximos de la muestra se obtienen al principio de la carga de datos, al realizar una iteración extra que compara los valores de cada jugador.

```
def normalizer(analyzer, player):
    properties = ["potential", "age", "height_cm", "value_eur"]
    for prop in properties:
        if player[prop] != "":
            if int(float(player[prop])) < analyzer["min-max"][prop][0]:
                analyzer["min-max"][prop][0] = int(float(player[prop]))
            if int(float(player[prop])) > analyzer["min-max"][prop][1]:
                analyzer["min-max"][prop][1] = int(float(player[prop]))
```

Este requerimiento resulta similar al requerimiento 2 en ejecución. Se reutiliza el índice de posiciones (mapa), del cual se obtiene un árbol como el cuarto valor de una tupla, valor no utilizado en el

requerimiento 2. Este árbol organiza a los jugadores de una posición en específico, según su valor representativo en la carga de datos:

```
def updateRepIndex(analizer, map, player):
    """
    CALCULA, el numero representativo de cada jugador y lo guadra en el arbol
    de la forma numero/jugador
    """
    properties = ["potential", "age", "height_cm", "value_eur"]
    representative = 0
    for prop in properties:
        normal = 0
        if player[prop] != "unknown":
            normal = (int(float(player[prop])) - analizer[prop][0]) / (analizer[prop][1] - analizer[prop][0])
        representative += normal
    entry = om.get(map, representative)
    if entry is None:
        listPlayers = lt.newList("ARRAY_LIST", cmpfunction=compareIds)
        lt.addLast(listPlayers, player)
        om.put(map, representative, listPlayers)
    else:
        valor = me.getValue(entry)
        lt.addLast(valor, player)

    return map
```

De esta manera se logra encontrar al jugador según su nombre, calcular su valor representativo, y encontrar los 6 jugadores en una posición en específico con un valor representativo cercano, en una complejidad de O(1).

```python
def playerReplace(name, pos, analyzer):
    namemp = analyzer['nameIndex']
    posmp = analyzer['posIndex']
    if not mp.contains(namemp, name.lower()):
        return None
    key = mp.get(namemp, name.lower())
    player = me.getValue(key)
    key2 = mp.get(posmp, pos.lower())
    repTree = me.getValue(key2)[3]
    # Se normaliza cada propiedad
    properties = ["potential", "age", "height_cm", "value_eur"]
    representative = 0
    for prop in properties:
        minV = analyzer["min-max"][prop][0]
        maxV = analyzer["min-max"][prop][1]
        normal = (int(float(player[prop])) - minV) / (maxV - minV)
        representative += normal
    replaceList = lt.newList("ARRAY_LIST", cmpfunction=compareIds)
    take = om.get(repTree, representative)
    playerList = me.getValue(take)
    for player in lt.iterator(playerList):
        player["repDif"] = 0
        player["rep"] = representative
        lt.addLast(replaceList, player)
    minKey = om.floor(repTree, representative)
    maxKey = om.ceiling(repTree, representative)
    getLimits(repTree, minKey, maxKey, replaceList, representative)

    sortByRep(replaceList)

    return replaceList, representative
```

Tabla de complejidad

De esta manera, relacionando lo descrito anteriormente, se creó la siguiente tabla:

Análisis de complejidad	Requerimiento 1 (Notación O)	Requerimiento 2 (Notación O)	Requerimiento 3 (Notación O)	Requerimiento 4 (Notación O)	Requerimiento 5 (Notación O)	BONO6 (Notación O)
Reto 3	O(1)	O(NLogN)	O(NLog)	O(NLog)	O(1)	O(1)

Conclusión y análisis

Hay una gran diferencia de tiempo en lo que respecta a encontrar rangos de valores en un árbol en comparación con un mapa. La implementación apropiada de estos dos TADs permitió construir unos algoritmos que resolvieran los requerimientos de manera más eficaz que en los Retos 1 y 2, sin mencionar que el tiempo que toma cargar los datos no es muy grande.

No obstante, cabe resaltar que en todos los requerimientos donde se espera encontrar un rango de valores dentro de un árbol, considerando un peor aunque improbable caso en donde todos los valores del árbol pertenezcan al rango buscado, la complejidad podría llegar a ser igual a la de una lista, O(N). Empero, se experimentó que en un caso promedio, todos los requerimientos resultan más efectivos al utilizar árboles en comparación con otro tipo de TADs.