ANÁLISIS DEL RETO

Sara García Agudelo, 202320378, s.garcia112@uniandes.edu.co

Daniela González Ovalle, 202320856, d.gonzalezo2@uniandes.edu.co

Sofía Arias Zuluaga, 202310260, s.ariasz2@uniandes.edu.co

Requerimiento 1

Este requerimiento busca obtener las ofertas entre un rango de fechas.

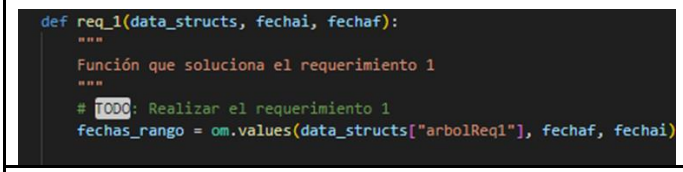
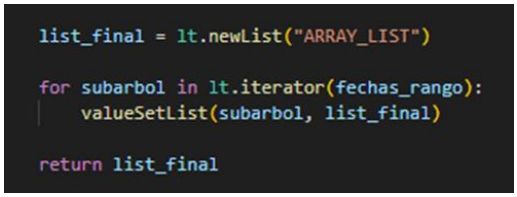
Descripción

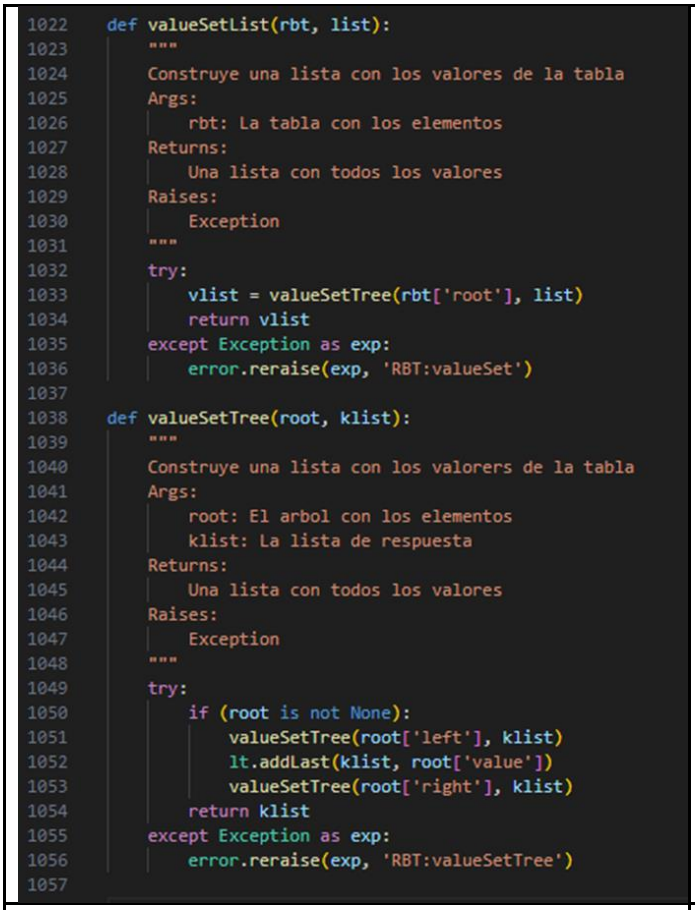
Breve descripción de como abordaron la implementación del requerimiento

Entrada	Fecha inicial y fecha final del rango.
Salidas	Tabla con las ofertas dentro de fechas ordenadas de más reciente a más antigua, si tiene misma fecha están ordenadas por salario.
Implementado (Sí/No)	Si, grupal

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	Complejidad
	O(N) En el peor caso, la función values() tendrá que recorrer todas las fechas si el usuario pide un rango desde la oferta más reciente a la más antigua.
	O(R) Primero, se crea una nueva lista que es O(C), luego se hace un recorrido con un for que debe recorrer todos los subárboles (divididos por fechas) dentro de las fechas del rango. Esto tiene complejidad O(R). Donde R es el número de fechas dentro del rango. A continuación, se explica la función valueSetList.

	O(M) La función valueSetList creada se basa en la función valueSet. ValueSet crea una nueva lista por cada árbol para añadir las ofertas dentro de ese árbol. No obstante, cuando esto se usa dentro del iterador for en la instancia anterior, la lista final quedaría como una lista de listas (una lista donde cada elemento es otra lista). Para solucionar esto, se creó una función donde ValueSet no cree una lista nueva por cada árbol, sino que añada las ofertas a una lista dada. Así, todas las ofertas quedarán dentro de una sola lista. La complejidad sería O(M), donde M es el número de ofertas que están dentro del árbol de una misma fecha. Dado que pocas (en comparación al número de ofertas totales) comparten la misma fecha de publicación ese valor será pequeño.
TOTAL	**O(N)** La complejidad total sería O(R x M), sin embargo, estos números serán ambos más pequeños que el número total de elementos. Por ende, juntos equivalen a O(N).

Pruebas Realizadas y Tablas

Descripción de las pruebas de tiempos de ejecución y memoria utilizada. Incluir descripción del procedimiento, las condiciones, las herramientas y recursos utilizados (librerías, computadores donde se ejecutan las pruebas, entre otros).

2022-10-20 a 2023-03-10

Procesadores	Intel(R) Core(TM) i7-5600U CPU @ 2.60GHz
Memoria RAM	8 GB
Sistema Operativo	Windows 10 Pro – 64 bits

Entrada	Tiempo (ms)	Memoria (kB)
10 pct	9.782	74.688
20 pct	13.108	150.938
30 pct	29.983	241.156
40 pct	39.024	343.484

50 pct	45.347	386.039
60 pct	56.493	434.594
70 pct	65.390	438.109
80 pct	81.326	492.359
large	99.839	492.391

Graficas

Las gráficas con la representación de las pruebas realizadas.

Análisis

Por un lado, la gráfica de tiempo tiene un comportamiento que se acerca mucho al linear; por ende, se puede verificar que la complejidad temporal experimental es O(N), lo cual es igual a la teórica. Similarmente, la memoria se comporta de forma linear. Aun así, al final se estabiliza por lo que podría ser logarítmica.

Requerimiento 2

El requerimiento 2 busca obtener todas las ofertas dentro de un rango de salarios mínimos.

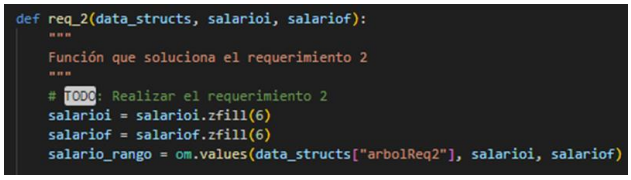
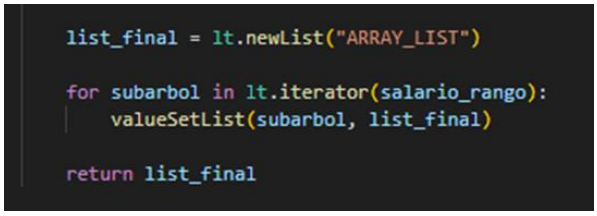
Descripción

Breve descripción de como abordaron la implementación del requerimiento

Entrada	Límite superior del salario mínimo y límite inferior del salario mínimo,
Salidas	Tabla con las ofertas dentro del rango de salario mínimo ordenadas de forma ascendente por el salario mínimo, si tiene mismo salario mínimo están ordenadas por fecha.
Implementado (Sí/No)	Si, grupal

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

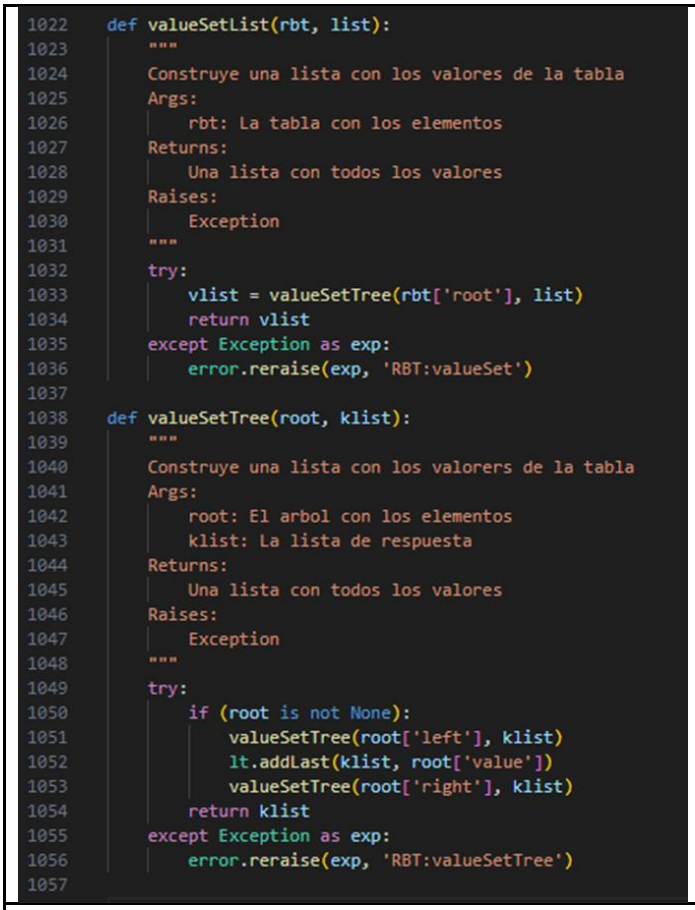
Pasos	**Complejidad**
	O(N) Las funciones zfill tienen complejidad constante. En el peor caso, la función values() tendrá que recorrer todos los salarios si el usuario pide un rango desde el menor salario mínimo hasta el mayor.
	O(R) Primero, se crea una nueva lista que es O(C), luego se hace un recorrido con un for que debe recorrer todos los subárboles (divididos por salarios mínimos) dentro de los salarios del rango. Esto tiene complejidad O(R). Donde R es el número de salarios mínimos dentro del rango. A continuación, se explica la función valueSetList.

```
def req_2(data_structs, salarioi, salariof):
    """
    Función que soluciona el requerimiento 2
    """
    # TODO: Realizar el requerimiento 2
    salarioi = salarioi.zfill(6)
    salariof = salariof.zfill(6)
    salario_rango = om.values(data_structs["arbolReq2"], salarioi, salariof)
```

```
    list_final = lt.newList("ARRAY_LIST")

    for subarbol in lt.iterator(salario_rango):
        valueSetList(subarbol, list_final)

    return list_final
```

	O(M) La función valueSetList creada se basa en la función valueSet. ValueSet crea una nueva lista por cada árbol para añadir las ofertas dentro de ese árbol. No obstante, cuando esto se usa dentro del iterador for en la instancia anterior, la lista final quedaría como una lista de listas (una lista donde cada elemento es otra lista). Para solucionar esto, se creó una función donde ValueSet no cree una lista nueva por cada árbol, sino que añada las ofertas a una lista dada. Así, todas las ofertas quedarán dentro de una sola lista. La complejidad sería O(M), donde M es el número de ofertas que están dentro del árbol de un mismo salario mínimo. Dado que pocas (en comparación al número de ofertas totales) comparten el mismo salario mínimo, este valor será pequeño.
TOTAL	**O(N)** La complejidad total sería O(R x M), sin embargo, estos números serán ambos más pequeños que el número total de elementos. Por ende, juntos equivalen a O(N).

Pruebas Realizadas y Tablas

Descripción de las pruebas de tiempos de ejecución y memoria utilizada. Incluir descripción del procedimiento, las condiciones, las herramientas y recursos utilizados (librerías, computadores donde se ejecutan las pruebas, entre otros).

Límite inferior salario mínimo: 2000

Límite superior salario mínimo: 6000

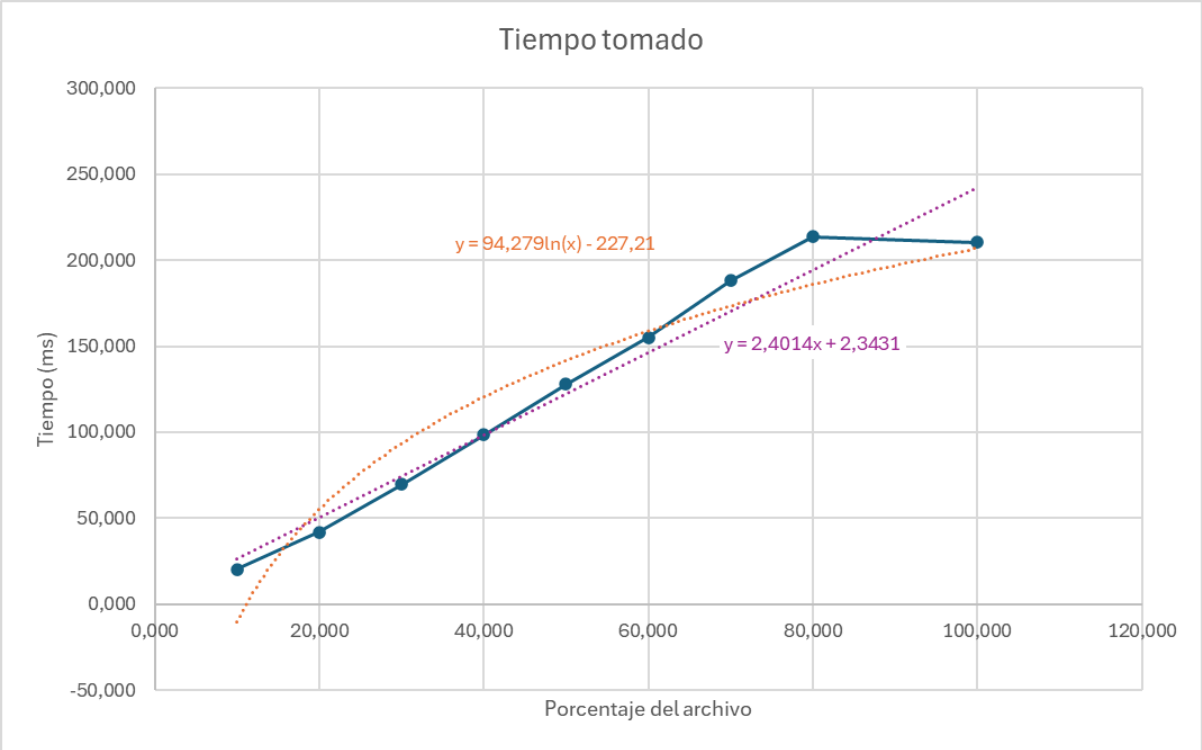
Procesadores	Intel(R) Core(TM) i7-5600U CPU @ 2.60GHz
Memoria RAM	8 GB
Sistema Operativo	Windows 10 Pro – 64 bits

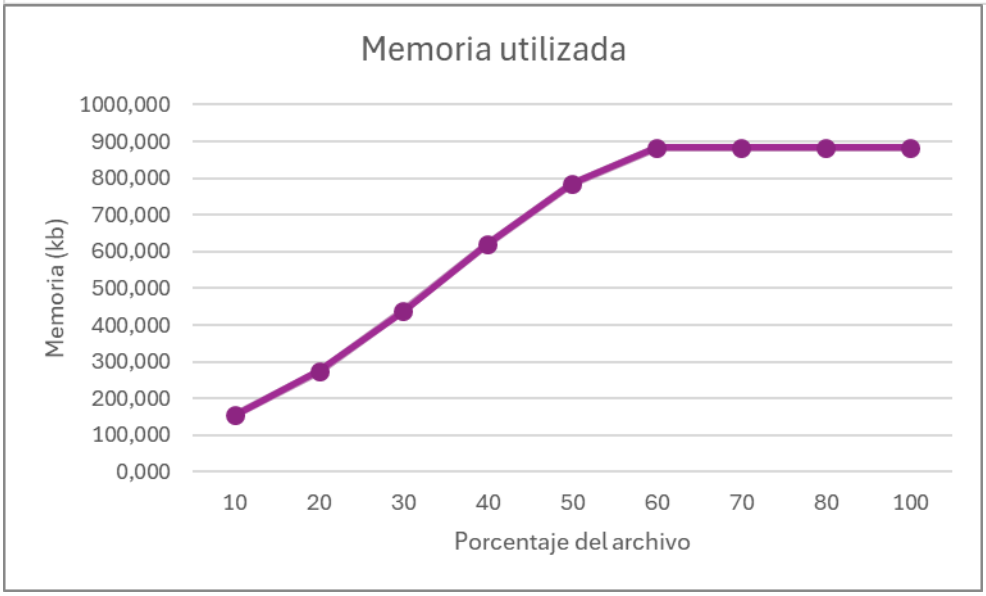
Entrada	**Tiempo (ms)**	**Memoria (kB)**
10 pct	20.285	154.297
20 pct	41.944	274.953
30 pct	69.706	437.688

40 pct	98.569	621.680
50 pct	127.858	785.898
60 pct	155.150	883.711
70 pct	188.111	884.141
80 pct	213.816	884.172
large	210.307	884.141

Graficas

Las gráficas con la representación de las pruebas realizadas.





Análisis

En la gráfica de tiempo se puede evidenciar que el comportamiento es mayormente lineal. El penúltimo dato es mayor al esperado en la gráfica lineal. Sin embargo, el último dato se comporta de forma coherente respecto a la gráfica logarítmica. En suma, el comportamiento de todos los datos se asemeja más al lineal, por lo cual es una complejidad O(N). La gráfica de memoria al inicio tiene un comportamiento lineal dado que los datos aumentan, pero luego del 60% estos se estabilizan y se vuelve constante. Por lo cual el comportamiento de la memoria podría ser logarítmico.

Requerimiento 3

Este requerimiento busca cumplir la función de recopilar un número específico N de ofertas de trabajo recientes que coincidan con un determinado código de país y nivel de experiencia.

Descripción

Breve descripción de como abordaron la implementación del requerimiento

Entrada	La estructura de datos, las N ofertas recientes que se desean recolectar, el código del país, el nivel de experiencia deseado.
Salidas	Lista de las ofertas de trabajo recientes recolectadas según los criterios especificados.
Implementado (Sí/No)	Sí, Implementado por Sofía Arias

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Paso 1: `def req_3(data_struct, N, country_code, experience_level):` `  year_keys = om.keySet(data_struct['tablaReq37'])  # Obtengo los años disponibles` `  recent_jobs= lt.newList('ARRAY_LIST')  # creo una lista para trabajos recientes` Extrae los años para para recorrerlos, pues la información viene de la tablareq37, para luego crear la lista 'recent_jobs' para guardar los trabajos recientes.	O(C)
Paso 2: `years_list = lt.newList()` `for year in lt.iterator(year_keys):` `  lt.addLast(years_list, year)` `# Ordenamos la lista de años en orden descendente` `sa.sort(years_list, sort_crit_years)` La lista de años se ordena en orden descendente utilizando una función de ordenamiento personalizada **sort_crit_years**.	**O(nlogn)**

Paso 3: Para cada año, se accede secuencialmente a mapas y submapas para obtener la información específica del país y del nivel de experiencia deseado.	O(1)
Paso 4: Después de iterar sobre los años, verifica si el número de ofertas en **recent_jobs** ha alcanzado *N*, en cuyo caso detiene la búsqueda. Finalmente, retorna la lista **recent_jobs** que contiene las ofertas recientes recopiladas según los criterios especificados.	O(C)
TOTAL	**Sin embargo, gracias a que la carga de datos ya ordena los datos necesarios para el requerimiento, la complejidad es mucho menor, O(NLOGN), es en el peor caso.**

```
for year in lt.iterator(years_list):
    year_value = me.getValue(om.get(data_struct['tablaReq37'], year))
    country_map = year_value['paises']
    country_value = me.getValue(mp.get(country_map, country_code))

    if country_value:
        exp_tree = country_value['tabla_experience_levels']
        exp_value = me.getValue(mp.get(exp_tree, experience_level))

        if exp_value:
            jobs_tree = exp_value['jobs']
            jobs_keys = om.keySet(jobs_tree)

            jobs_list = lt.newList()
            for job_key in lt.iterator(jobs_keys):
                lt.addLast(jobs_list, job_key)
```

```
        if lt.size(recent_jobs) >= N:  # Verifico si ya se tienen N ofertas
            break

return recent_jobs  # Retorno las ofertas recientes
```

Pruebas Realizadas

Descripción de las pruebas de tiempos de ejecución y memoria utilizada. Incluir descripción del procedimiento, las condiciones, las herramientas y recursos utilizados (librerías, computadores donde se ejecutan las pruebas, entre otros).

Los datos de entrada fueron:

N ofertas recientes: 9

País: PL

Nivel de experiencia: mid

Procesadores	**Apple Silicon Chip M2**
Memoria RAM	8 GB
Sistema Operativo	MacOS Sonoma 14.4.1

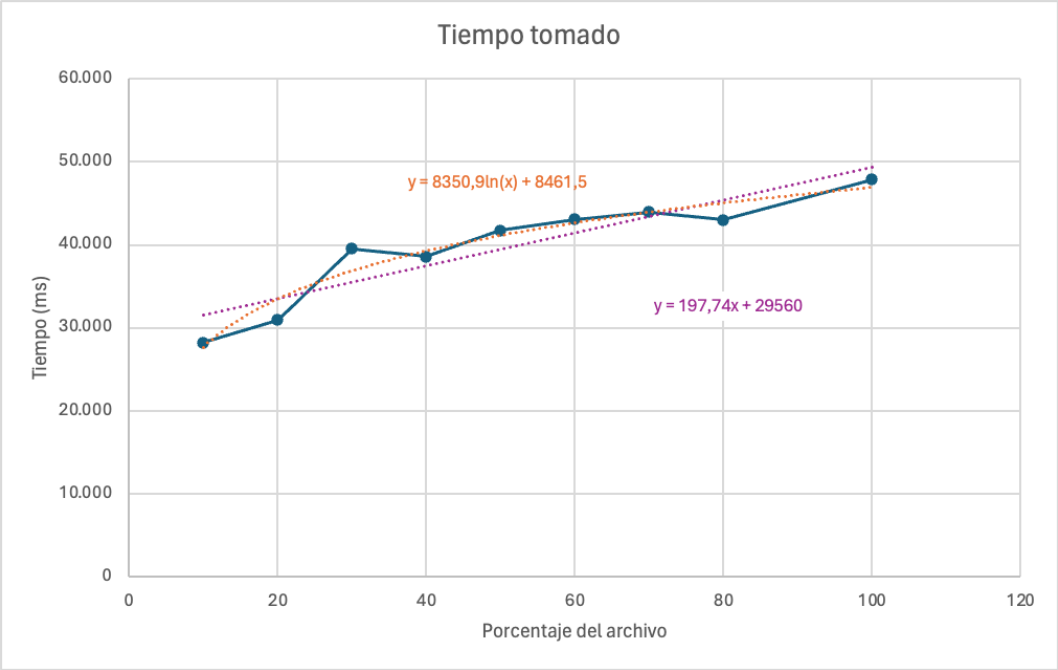
Tablas de datos

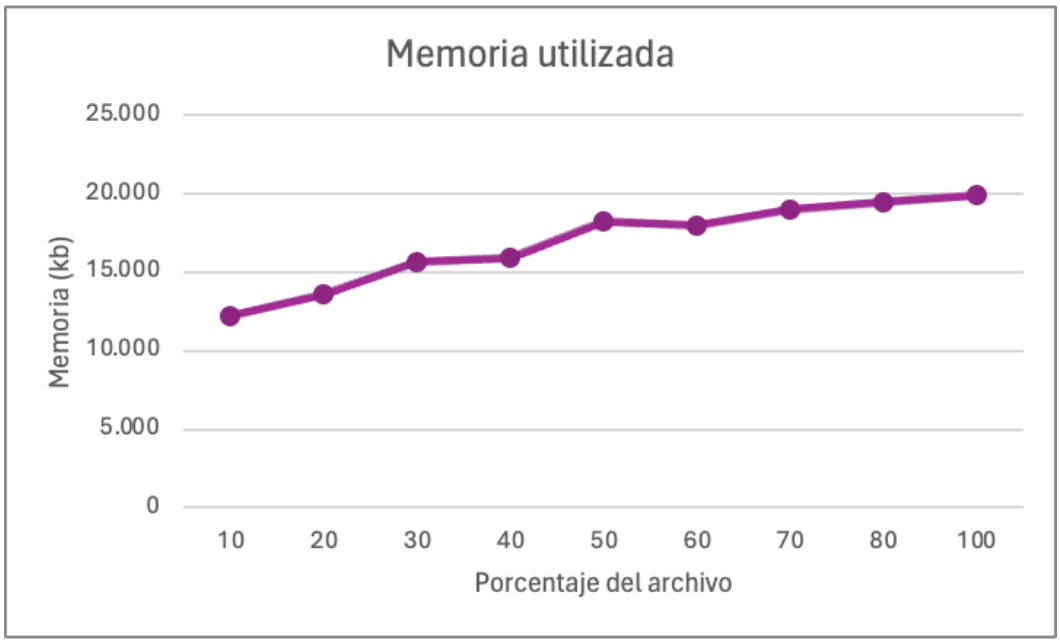
Las tablas con la recopilación de datos de las pruebas.

Entrada	Tiempo (ms)	Memoria (kB)
10 pct	28.224	12.237
20 pct	30.970	13.562
30 pct	39.543	15.678
40 pct	38.605	15.901
50 pct	41.748	18.232
60 pct	43.091	17.974
70 pct	43.945	19.005
80 pct	42.981	19.459
large	47.892	19.901

Graficas

Las gráficas con la representación de las pruebas realizadas.





Análisis

La relación entre la complejidad teórica del requerimiento y el comportamiento observado en las gráficas de memoria y tiempo refleja una implementación efectiva de estructuras de datos complejas y anidadas que están optimizadas para manejar grandes volúmenes de datos con eficiencia. El rendimiento observado en estas gráficas indica que, pese al aumento en los datos procesados, la memoria y el tiempo se gestionan para reflejar las características teóricas esperadas de las operaciones de complejidad O(N) y lineales combinadas.

Requerimiento 4

Este requisito consulta las N ofertas más recientes con un tipo de ubicación de trabajo y ciudad específicos.

Descripción

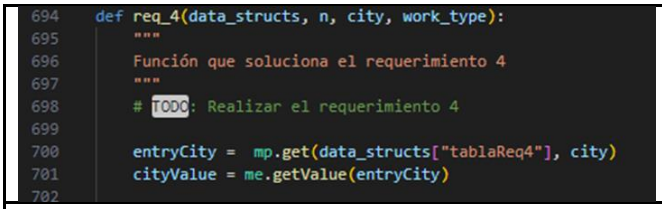
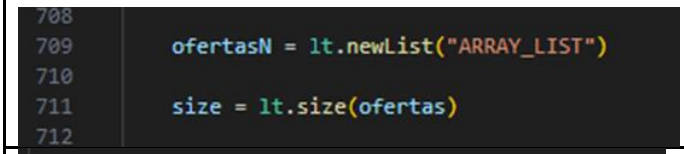
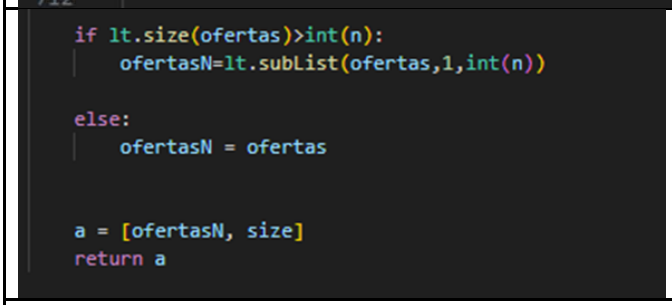
Breve descripción de como abordaron la implementación del requerimiento

Entrada	Ciudad y tipo de ubicación.
Salidas	Las N ofertas más recientes de esa ciudad y ubicación
Implementado (Sí/No)	Sí, por Sara García

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
	O(C) Se obtiene dentro de una tabla de hash la ciudad específica. El valor es otra tabla

	dividida entre los tipos de ubicación de trabajo.
	O(C) Dentro de una tabla hash, se obtienen las ofertas del tipo de ubicación de trabajo específico. El valor es un árbol de las ofertas.
	O(N) La función valueSet obtiene el árbol que contiene las ofertas de esa ciudad y tipo de ubicación especifico y recorre todas las ofertas.
	O(C) Se crea una nueva lista que contendrá las N ofertas más recientes y el tamaño de todas las ofertas.
	O(Número N de ofertas) Depende del número que pida el usuario. En el peor caso, sacar la sublista sería O(N).
TOTAL	**O(N)** Sin embargo, dado que ya se filtró por la ciudad y tipo de ubicación el número de elementos es mucho más pequeño.

Pruebas Realizadas y Tablas

Descripción de las pruebas de tiempos de ejecución y memoria utilizada. Incluir descripción del procedimiento, las condiciones, las herramientas y recursos utilizados (librerías, computadores donde se ejecutan las pruebas, entre otros).

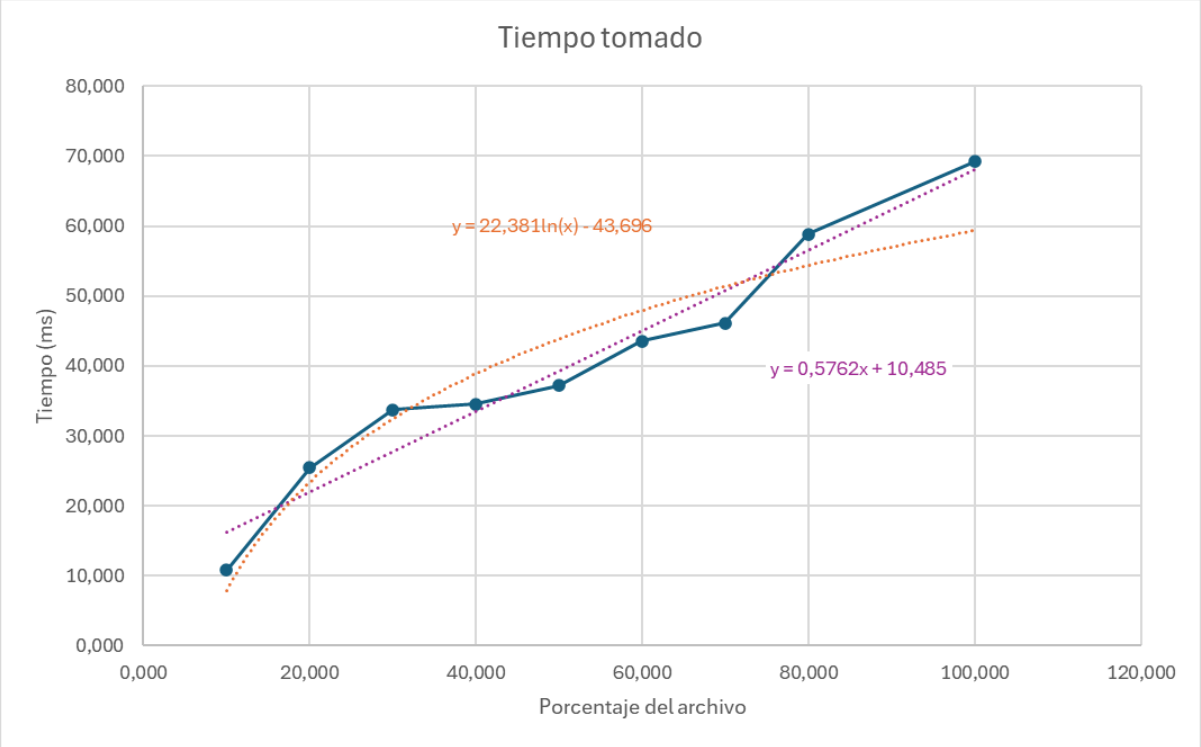
Los datos de entrada fueron 10 más recientes, Warszawa, remote

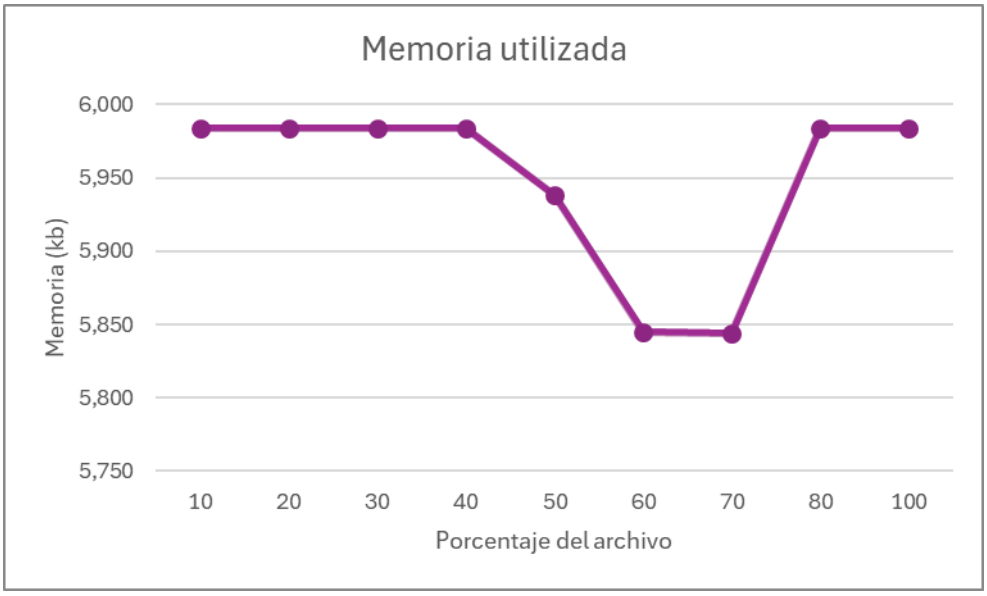
Procesadores	Intel(R) Core(TM) i7-5600U CPU @ 2.60GHz
Memoria RAM	8 GB

Sistema Operativo	Windows 10 Pro – 64 bits

Entrada	Tiempo (ms)	Memoria (kB)
10 pct	10.776	5.984
20 pct	25.422	5.984
30 pct	33.729	5.984
40 pct	34.566	5.984
50 pct	37.223	5.938
60 pct	43.570	5.845
70 pct	46.111	5.844
80 pct	58.831	5.984
large	69.198	5.984

Graficas





Análisis

Las gráficas muestran que los datos de tiempo se acercan mucho al comportamiento linear, lo cual representaría una complejidad temporal de O(N). Así mismo, se puede ver que los datos de memoria tienen muy poca variación. Pues, en 60 y 70 porciento disminuye, pero esto es menor de 1kB.

Requerimiento 5

Descripción

Este requisito consulta las N ofertas más antiguas con un nivel mínimo y máximo para una habilidad solicitada para empresas en un rango de tamaño.

Entrada	El número (N) de ofertas laborales para consulta. El límite inferior del tamaño de la compañía. El límite superior del tamaño de la compañía. Nombre de la habilidad solicitada. El límite inferior del nivel de la habilidad. El límite superior del nivel de la habilidad.
Salidas	Lista_final= Las N ofertas laborales publicadas más antiguas que cumplan con las condiciones especificadas. Ofertas_totales= El número total de ofertas laborales publicadas para las compañías que tengan un tamaño en un rango y que requieran una habilidad específica.
Implementado (Sí/No)	Sí, por Daniela González.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	Complejidad
580 arbolFinalJobs=om.newMap(omaptype="RBT") 581 ofertas_totales=0 582 583 #Saco los arboles de la habilidad requerida 584 skillPareja=mp.get(data_structs["tablaReq5"], skill) 585 arbolNiveles=me.getValue(skillPareja) Primero se crea un nuevo mapa donde se irán guardando los trabajos que cumplen con los rangos puestos por el usuario. Después se saca la habilidad pedida por el usuario de la tabla de hash y de esta el árbol de niveles de esa habilidad.	O(C) Conseguir la habilidad tiene complejidad constante porque es sacada de una tabla de hash. Si los datos están distribuidos equitativamente, esto es O(1). Si hay clusters, la complejidad aumenta un poco más que O(1). La creación de un arbol es O(C) también.
587 #Saco los valores del arbol que están en el rango de nivel de habilidades pedido 588 niveles=om.values(arbolNiveles, minSkillLev, maxSkillLev) Se saca los niveles dentro del rango de nivel de habilidad dada por el usuario.	En el peor caso que el usuario pide un rango que incluye todos los niveles de habilidad sería O(5) porque solo existen 5 niveles de habilidad.
590 #Itero por los arboles de las empresas en cada nivel para buscar las ofertas con los 591 # tamaños de empresa correctos: 592 for arbolEmpresa in lt.iterator(niveles): 593 minCompKey=minCompSize.zfill(7) 594 maxCompKey=maxCompSize.zfill(7) 595 lista_listasJobs=om.values(arbolEmpresa, minCompKey, maxCompKey) 596 597 #Recorro cada lista de trabajos 598 for listaJobs in lt.iterator(lista_listasJobs): 599 600 #Recorro cada trabajo 601 for jobTuple in lt.iterator(listaJobs): 602 ofertas_totales+=1 603 om.put(arbolFinalJobs, jobTuple[0], jobTuple[1]) Dentro del valor de cada nivel hay un árbol que contiene como llave el tamaño de las empresas. Estas tienen como valor una lista que contiene todos los	El for arbolEmpresas in lt.iterator(niveles) recorre máximo 5 veces si el usuario pidió el rango de 1-5 de nivel de habilidad lo cual sería O(5). Sacar los tamaños de empresas con la función om.values() y recorrer cada uno de estos en el peor caso sería O(E), siendo E el número de distintos tamaños de empresas, si el usuario pide

trabajos de ese nivel de habilidad con una empresa de ese tamaño. En este paso, primero se recorre cada nivel de habilidad, de cada uno se sacan los tamaños de empresas que están en el rango pedido por el usuario (para eso hay que cambiar el máximo y mínimo tamaño de compañía al formato de llave), y de cada uno se sacan los trabajos que tenían esa habilidad y ese tamaño de empresa. Se guardan en un arbol con una llave compuesta de su fecha, salario e id para que queden ordenados.	un rango que abarca todos los tamaños de empresa. Recorrer cada tupla de trabajo es O(N) e insertarla en el arbol es O(LogN). En total este paso quedaría O(5*E*N) en el peor de los casos, pero como cada trabajo tiene apenas un solo tamaño de empresa y un solo nivel para la habilidad pedida, no se pasa más de una vez por ningún trabajo. Por ende, la complejidad termina siendo más cercana a O(N).
605 listaFinal=om.valueSet(arbolFinalJobs) 606 Se guarda en una lista de manera ordenada todos los trabajos guardados en el arbol.	O(N) porque se pasa por cada uno de los trabajos guardados en el arbol.
607 if lt.size(listaFinal)>int(numOfertas): 608 listaFinal=lt.subList(listaFinal,1,int(numOfertas)) 609 610 return listaFinal, ofertas_totales Si el usuario pidió un número menor de ofertas que la cantidad total de ofertas que estaban dentro de los rangos pedidos, se acorta la lista que será impresa. Como fueron guardados de manera ordenada, son los N trabajos más recientes y ordenados por salarios si tienen la misma fecha.	O(numero de ofertas pedidas).
Total: O(N)	

Pruebas realizadas y sus tablas

La maquina usada para las pruebas fue la siguiente. Los datos de entrada fueron 25 ofertas, tamaño mínimo de empresa 1000, tamaño máximo de empresa 10000, habilidad PYTHON, nivel de habilidad mínimo 2 y máximo 4.

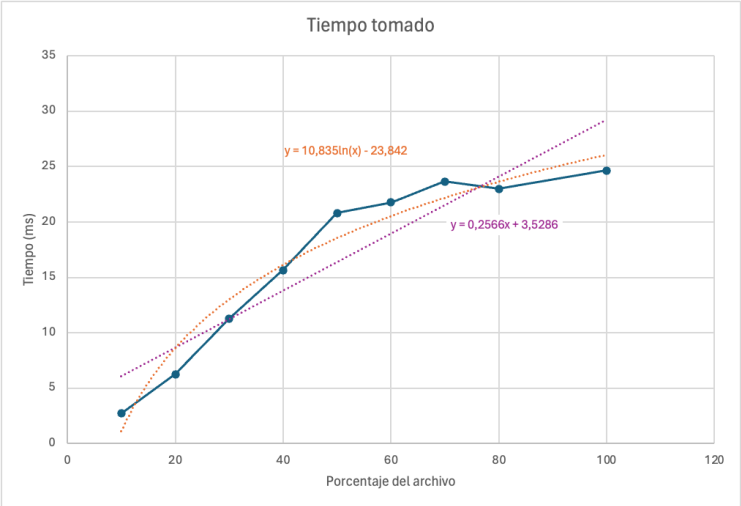
Procesadores	**Apple M2**
Memoria RAM	8 GB
Sistema Operativo	Sonoma 14.4.1

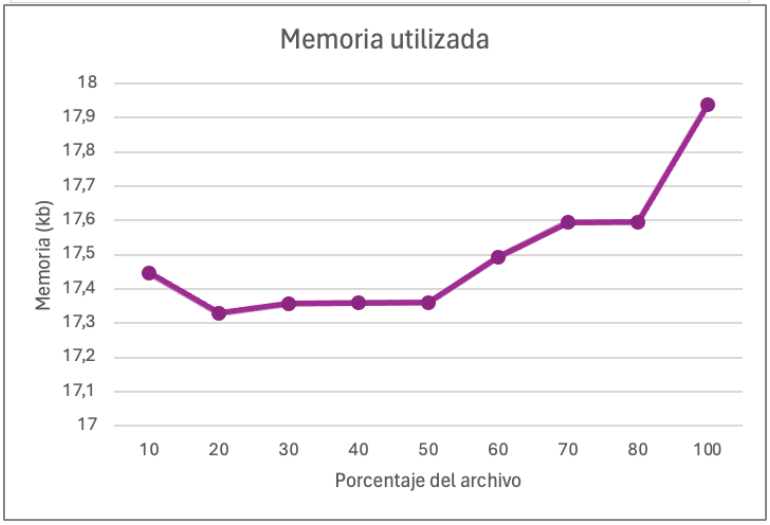
Entrada	**Tiempo (ms)**	**Memoria (kB)**
10 pct	2.717	17.447
20 pct	6.259	17.330
30 pct	11.286	17.358
40 pct	15.662	17.360

50 pct	20.813	17.361
60 pct	21.769	17.494
70 pct	23.667	17.594
80 pct	22.993	17.596
large	24.649	17.939

Graficas

Las gráficas con la representación de las pruebas realizadas.





Análisis

En la gráfica de tiempo, la complejidad del algoritmo se ve que es más logarítmica que lineal. La diferencia con respecto a la complejidad teórica que era lineal puede ser dada a que om.values() tiene una complejidad más similar a una logarítmica si el rango de valores no abarca a todos los valores en el árbol.

Requerimiento 6

El requerimiento 6 pide extraer información detallada sobre ofertas de empleo basadas en criterios específicos de salario y fecha, ofreciendo un resumen sobre cuáles ciudades tienen el mayor número de ofertas que cumplen con estos criterios, y proporcionando detalles adicionales de las ofertas en la ciudad con más oportunidades.

Descripción

Breve descripción de como abordaron la implementación del requerimiento

Entrada	La estructura de datos, las N ciudades de las que se quieran ver ofertas, la fecha inicial, la fecha final, el salario mínimo ofertado, el salario máximo ofertado.
Salidas	El número total de ofertas, las ciudades con ofertas que cumplen las especificaciones ordenadas por número de ofertas, la ciudad con más ofertas y sus detalles, numero de ciudades que cumplen con las especificaciones.
Implementado (Sí/No)	Sí, Implementado (grupal)

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Paso1: `city_offers_count = mp.newMap(numelements=N, maptype='PROBING', loadfactor=0.5, cmpfunction=compare_mapId)` `total_offers = 0` Se inicializa city_offers_count como un mapa de tipo 'PROBING' para almacenar el conteo de ofertas que cumplen con los criterios en cada ciudad. total_offers se inicializa en 0 para llevar el conteo total de ofertas que cumplen con los criterios especificados.	O(1)
Paso 2: `# Recorro las ciudades y cuento las ofertas que cumplen con las especificaciones` `for city in lt.iterator(mp.keySet(data_struct['tablaReq6'])):` `  city_entry = mp.get(data_struct['tablaReq6'], city)` Este loop recorre cada ciudad en la estructura de datos data_struct['tablaReg']. Para cada ciudad, verifica si existen trabajos y cuenta aquellos que cumplen con los rangos de salario y fecha utilizando la función count_city_offers. Si hay trabajos que cumplen los criterios en una ciudad, estas se añaden al mapa city_offers_count y se actualiza el contador total_offers.	O(N)
Paso 3:	

`` # Creo una lista de tuplas (ciudad, conteo) city_count_list = lt.newList('ARRAY_LIST') keySet=mp.keySet(city_offers_count) for cityKey in lt.iterator(keySet): pareja=mp.get(city_offers_count, cityKey) count = me.getValue(pareja) if count > 0: lt.addLast(city_count_list, (cityKey, count)) ``` Se crea una lista city_count_list para almacenar tuplas de cada ciudad junto con su conteo de ofertas, para luego recorrer las ciudades en city_offers_count y agrega cada par (ciudad, conteo) a la lista city_count_list	O(N)
Paso 4: ``` city_count_list = sa.sort(city_count_list, sort_crit_cityJobSize) num_ciudades=lt.size(city_count_list) # Selecciono las N primeras ciudades de la lista ordenada top_cities = lt.subList(city_count_list , 1, N) if lt.size(city_count_list) >= N else city_count_list top_city_details = None city_with_most_offers = lt.firstElement(top_cities) top_city_details = get_city_offer_details(data_struct, city_with_most_offers[0],start_date, end_date,min_salaryKey, max_salaryKey) #Ordeno alfabeticamente sortedalf_city_count_list = sa.sort(top_cities, sort_critJobs_ciudad_MenorMayor) answer=[total_offers, sortedalf_city_count_list, top_city_details, num_ciudades] return answer ``` Ordena la lista city_count_list según el criterio especificado en sort_criterion, que probablemente ordene de mayor a menor basado en el número de ofertas. Luego se extraen las primeras N ciudades de la lista ordenada city_count_list, basado en el parámetro N de la función. Finalmente, se obtiene la ciudad con más ofertas de la lista top_cities.Se llama a get_city_offer_details para obtener detalles más específicos de la ciudad con más ofertas, para luego ordenar la lista de la N ciudades alfabéticamente	O(NlogN)
Paso 5 (FUNCION AUXILIAR 1): ``` def count_city_offers(salary_tree, start_date, end_date, min_salary, max_salary): """ Cuenta las ofertas de trabajo por ciudad que cumplen con las especificaciones de fecha y salario. Args: salary_tree (RBT): Árbol rojo-negro con la información de salarios y fechas. start_date (str): Fecha de inicio para filtrar ofertas. end_date (str): Fecha de finalización para filtrar ofertas. min_salary (int): Salario mínimo para filtrar ofertas. max_salary (int): Salario máximo para filtrar ofertas. Returns: int: Número total de ofertas que cumplen con las especificaciones. """ offer_count = 0 # Itera a través de los rangos de salario que cumplen con las especificaciones. salary_range = om.values(salary_tree, max_salary, min_salary) TODO: Dani: tocaba cambiar el orden de max salary y min salary cuando se invoca la función porque tiene como #cmpfunction el defaultfunction_invertido #Ten cuidado entonces con eso cuando vayas hacer values en get_city_offer_details for date_tree in lt.iterator(salary_range): # Itera a través de las fechas que cumplen con las especificaciones dentro de cada rango de salario. date_range = om.values(date_tree, end_date, start_date) for job_list in lt.iterator(date_range): # Cada nodo de este árbol contiene una lista de trabajos, por lo que sumamos la longitud de cada lista al conteo. offer_count += lt.size(job_list) return offer_count ``` Esta función cuenta los trabajos en una ciudad que cumplen con ciertos rangos de salario y fecha especificados, utilizando un árbol rojo-negro (RBT) para organizar los datos por salario y, anidado dentro, otro árbol por fecha.	O(C)

Paso 6 (FUNCION AUXILIAR 2): Obtiene detalles de los trabajos de la ciudad con el mayor número de ofertas, recolectando información relevante dentro de los rangos de salario y fecha especificados.	O(C)
Total	O(Nlog(N))

```
def get_city_offer_details(data_struct, city_name, start_date, end_date,min_salary, max_salary):
    """
    Esta función recupera los detalles de las ofertas laborales de la ciudad
    con el mayor número de ofertas publicadas.
    """
    city_entry = mp.get(data_struct['tablaReq6'], city_name)
    all_offers = lt.newList('ARRAY_LIST')

    if city_entry:      You, 2 days ago • req6 finished-falta tabulate
        salary_tree = me.getValue(city_entry)
        salary_range = om.values(salary_tree, max_salary, min_salary)


    for date_tree in lt.iterator(salary_range):
        date_range = om.values(date_tree, end_date, start_date)

        for hour_salary_id_tree in lt.iterator(date_range):

            for job_key in lt.iterator(om.keySet(hour_salary_id_tree)):
                job = om.get(hour_salary_id_tree, job_key)['value']
                lt.addLast(all_offers, job)

    if lt.size(all_offers) > 10:
        first_five = lt.subList(all_offers, 1, 5)
        last_five = lt.subList(all_offers, lt.size(all_offers) - 4, 5)
        # nueva lista con ultimos y primeros 5
        top_city_details = lt.newList('ARRAY_LIST')
        for offer in lt.iterator(first_five):
            lt.addLast(top_city_details, offer)
        for offer in lt.iterator(last_five):
            lt.addLast(top_city_details, offer)
    else:
        top_city_details = all_offers

    return top_city_details
```

Pruebas Realizadas

Descripción de las pruebas de tiempos de ejecución y memoria utilizada. Incluir descripción del procedimiento, las condiciones, las herramientas y recursos utilizados (librerías, computadores donde se ejecutan las pruebas, entre otros).

Los datos de entrada fueron:

N de ciudades: 8

Fecha inicial: 2022-12-09

Fecha final: 2023-12-09

Salario mínimo (en USD): 0

Salario máximo (en USD): 123.456

Procesadores	**Apple Silicon Chip M2**
Memoria RAM	8 GB
Sistema Operativo	MacOS Sonoma 14.4.1

Tablas de datos

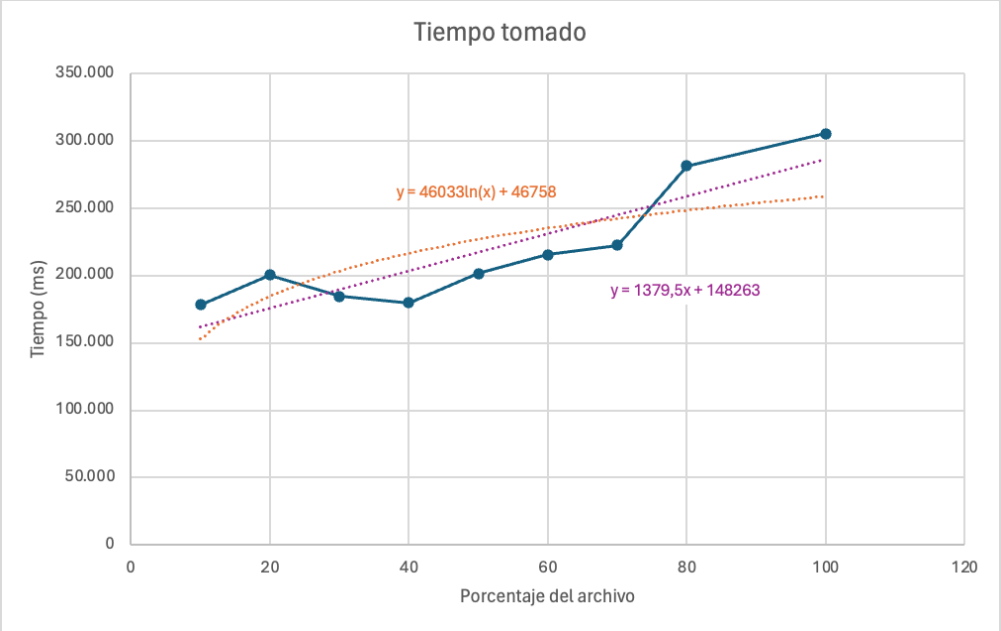
Las tablas con la recopilación de datos de las pruebas.

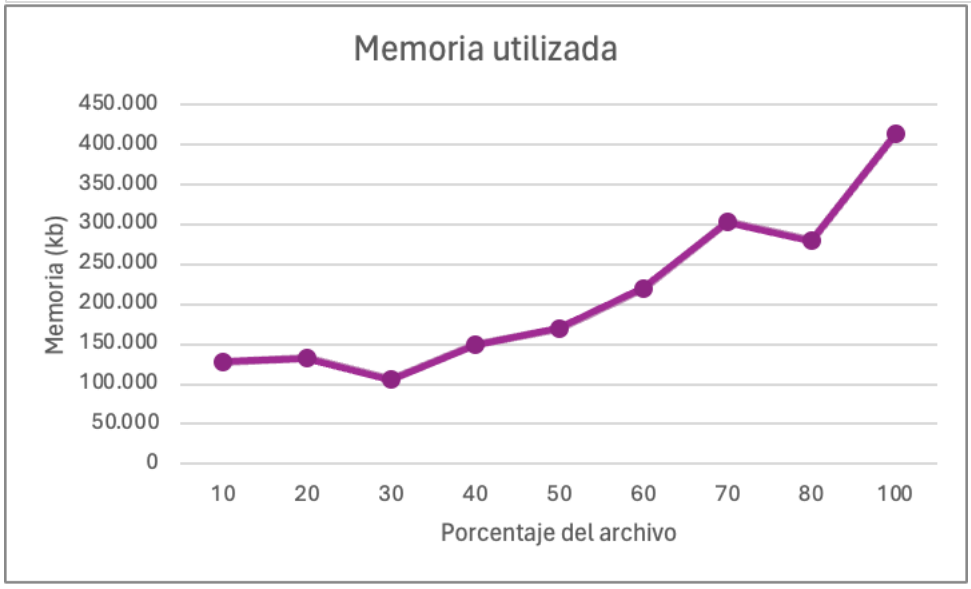
Entrada	**Tiempo (ms)**	**Memoria (kB)**
10 pct	178.248	128.345
20 pct	200.301	132.122

30 pct	184.560	105.482
40 pct	179.712	148.734
50 pct	201.345	170.035
60 pct	215.485	220.198
70 pct	222.341	302.451
80 pct	281.532	279.011
large	305.409	413.098

Graficas

Las gráficas con la representación de las pruebas realizadas.





Análisis

El comportamiento observado en las gráficas puede ser influenciado por múltiples factores técnicos, como la implementación específica de las estructuras de datos y las funciones utilizadas. Si bien la complejidad teórica nos da una expectativa de O(NlogN), las optimizaciones y la gestión efectiva de memoria atenúan el impacto visible en las pruebas prácticas. Se puede concluir que, mientras que el tiempo de ejecución muestra un crecimiento que podría estar en línea con un componente logarítmico, el uso de memoria refleja una gestión que se incrementa linealmente con el tamaño del archivo.

Requerimiento 7

Descripción

Este código contabiliza las ofertas laborales publicadas para un país y un año específico según alguna propiedad de interés como lo son el nivel de experticia requerido, el tipo de ubicación del trabajo, o habilidad específica.

Entrada	El año relevante (en formato "%Y"), el código del país para la consulta (ej.: PL, CO, ES, etc), y la propiedad de conteo (experticia, ubicación, o habilidad).
Salidas	Retorna una lista que contiene: NumJobsYear= número de ofertas en ese año NumJobsGrafica = número de ofertas utilizadas para hacer el gráfico InfoGrafica = información para poder plot el gráfico de barras MaxValue = valor máximo de la propiedad consultada MinValue = valor mínimo de la propiedad consultada JobsList = lista de ofertas laborales
Implementado (Sí/No)	Si, grupal

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
640 #Saco el año que pidio el usuario 641 yearPareja=mp.get(data_structs["tablaReq37"], year) 642 yearValue=me.getValue(yearPareja) 643 #Consigo el número de ofertas totales del año 644 numJobsYear=yearValue["ofertas_totales"] 645 646 #Consigo la info de ese pais 647 paisPareja=mp.get(yearValue["paises"], pais) 648 paisValue=me.getValue(paisPareja) 649 En este paso se sacan los valores del año y del país pedido por el usuario que están guardados en una tabla de hash. Además, se guarda el número de ofertas totales del país (es calculado en la carga de datos) en la variable que será retornada.	O(C) Conseguir estos valores tiene complejidad constante porque son sacados de una tabla de hash. Si los datos están distribuidos equitativamente, esto es O(1). Si hay clusters, la complejidad aumenta un poco más que O(1).
925 #Depende de la propiedad edito el título del gráfico y escojo los arboles que voy acceder 926 if propiedad=="ubicacion": 927 xLabel="Tipos de ubicaciones" 928 title="Cantidad de ofertas por ubicaciones en "+pais+" "+year 929 nombre_arbolPropiedad="arbol_workplace_type" 930 numJobsGrafica=paisValue["ofertas_totales"] 931 skillTF=False 932 933 elif propiedad=="experticia": 934 xLabel="Niveles de experticia" 935 title="Cantidad de ofertas por niveles de experticia en "+pais+" "+year 936 nombre_arbolPropiedad="arbol_experience_levels" 937 numJobsGrafica=paisValue["ofertas_totales"] 938 skillTF=False 939 940 else: 941 xLabel="Habilidades" 942 title="Cantidad de ofertas por habilidades en "+pais+" "+year 943 nombre_arbolPropiedad="arbol_skills" 944 numJobsGrafica=paisValue["ofertas_totales_skills"] 945 skillTF=True	O(C) porque son solamente asignaciones de string y un True o False.

En este paso se determina el título de la gráfica, el título del eje x de esta gráfica, la cantidad de ofertas que seran utilizadas para la gráfica (toda oferta tiene nivel de experticia y ubicación por ende estos dos valores seran los mismos entre si), el nombre del árbol que se va a acceder dependiendo de la propiedad pedida por el usuario, y skillTF que sera útil más adelante.	
 Se crea un nuevo mapa que se utilizará para guardar los trabajos ordenadamente. Además se sacan los valores máximo y mínimo del mapa de la propiedad. Para saber el número de ofertas de esa propiedad y su nombre, se divide la llave compuesta.	Creación de mapa: O(C) Búsqueda de llave maxima y mínima: O(logN)
 Se invoca la función valueKeySet. Esta función recorre el arbol de manera ordenada para: • Sacar todas las llaves, descomponerlas y guardarlas en un diccionario cuyas llaves son cada propiedad y el valor el número de ofertas de la propiedad. Este diccionario será utilizado por matplot lib para graficar (por ende no puede ser un ADT de disclib). • Sacar todos los valores que son listas de trabajos y recorrer esas listas para guardar cada trabajo en un arbol (los trabajos están guardados en la lista como una tupla, donde el primer valor de la tupla es un string que sirve como llave compuesta de su fecha, salario y ID y el segundo valor es el trabajo en sí).	O(N). La explicación de la complejidad de esta función depende de si se escogió ubicación, experiencia o habilidad por el usuario. Si se pidio ubicación o experiencia, cada una de estas propiedades apenas tiene tres distintas categorías y cada trabajo solo tiene una sola ubicación o experiencia. Por ende, la complejidad resulta siendo O(N) , con N igual a la cantidad de trabajos en el año y país pedido, porque solo se recorre cada trabajo una sola vez. La inserción en el arbol es O(LogN), que es menor que O(N). Si se pide habilidad, van haber muchas más habilidades que solo tres. Además, muchos trabajos tienen más de una habilidad entonces estarían guardados en más de una lista. Por ende, en la carga de datos se creó una lista aparte donde se guardó cada trabajo con al menos una habilidad una sola vez. Si el usuario quiere

○ Si la propiedad escogida fue habilidad (si SkillTF es True), en vez de recorrer los trabajos dentro de los valores del árbol, los trabajos son recorridos en una lista externa que está guardada en la tabla del data structure.	entonces ver los trabajos de habilidades, se recorre está lista lo cual es O(N) y se añade al arbol lo cual es O(LogN).
687 jobsList=lt.newList("ARRAY_LIST") 688 valueSetList(jobsArbol, jobsList) Se crea una nueva lista para guardar todos los trabajos en una lista para impresión. Se pasan esos trabajos del arbol a la lista usando la función de valueSetList (mirar funciones auxiliares para explicación de esta función).	O(N), siendo N la cantidad de trabajos en el año del país pedido.
690 #Saco los valores para el eje x y y del gráfico 691 propiedades = list(data.keys()) 692 totales_propiedades = list(data.values()) 693 694 infoGrafica=[propiedades, totales_propiedades, xLabel, title] 695 696 answer=[numJobsYear, numJobsGrafica, infoGrafica, maxValue, minValue, jobsList] 697 698 return answer Saco una lista de las llaves y otra de los valores del diccionario que tenía propiedad: número de ofertas. Estos serán los valores para el eje x y el eje y de la gráfica. Guardo esto, el título del eje x, y el título de la gráfica en una lista para crear la gráfica en el view.	O(2*P), siendo P la cantidad de distintas categorías de la propiedad. Si se pidió ubicación o nivel de experticia, el diccionario solo tendrá 3 valores entonces será O(6). Si se pidió habilidades, el número será más alto.
Total: O(N) o O(P) , siendo N el número de trabajos en el país y el año escogidos y P el número de distintas propiedades de ese país en ese año. Si el usuario pidió habilidades, es O(P) porque este es un número mayor que O(N). Si el usuario pidió experticia o ubicación, O(N).	

Pruebas Realizadas y tablas de datos:

Descripción de las pruebas de tiempos de ejecución y memoria utilizada. Incluir descripción del procedimiento, las condiciones, las herramientas y recursos utilizados (librerías, computadores donde se ejecutan las pruebas, entre otros).

Los datos de entrada fueron países: año: 2022, país: PL, propiedad: ubicacion

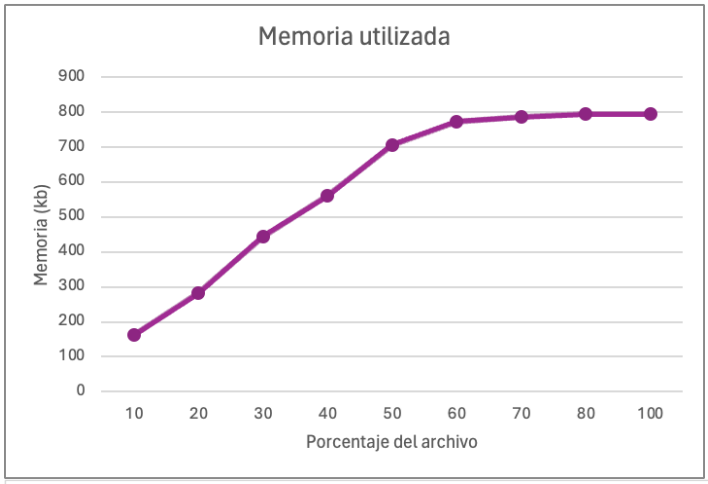
Procesadores	**Apple M2**
Memoria RAM	8 GB
Sistema Operativo	Sonoma 14.4.1

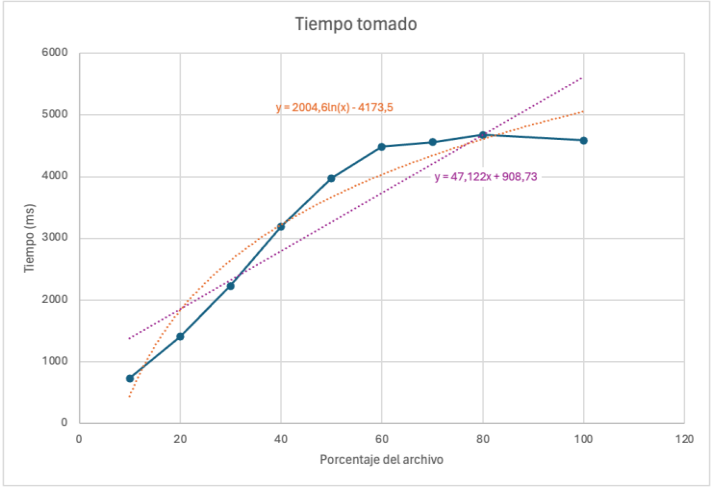
Entrada	**Tiempo (ms)**	**Memoria (kB)**
10 pct	734.135	161.177
20 pct	1407.591	281.990
30 pct	2229.721	445.115
40 pct	3192.172	560.514
50 pct	3975.812	706.428

60 pct	4488.228	773.364
70 pct	4559.189	785.466
80 pct	4677.827	793.365
large	4589.959	793.420

Graficas

Las gráficas con la representación de las pruebas realizadas.





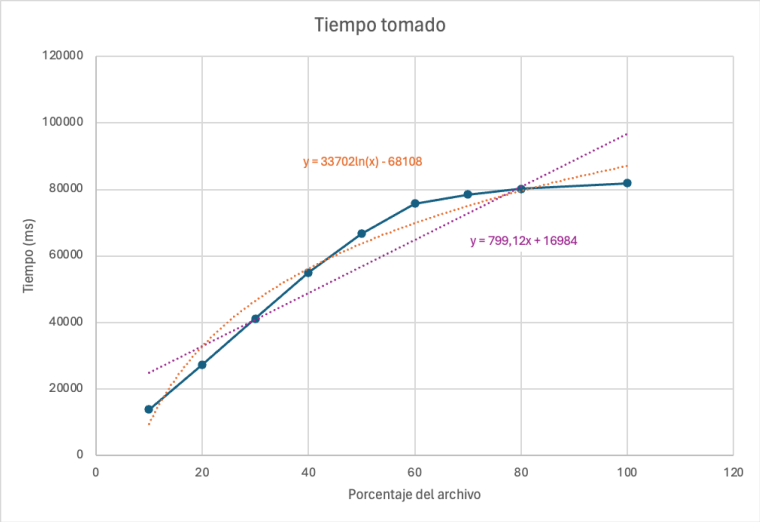
Análisis

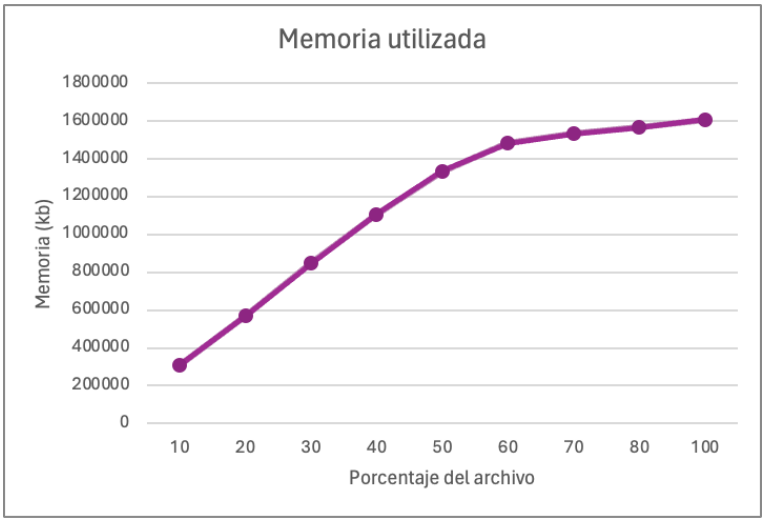
A pesar de que nuestro algoritmo tenía una complejidad teórica de O(N) , la gráfica de complejidad de tiempo es más similar a una complejidad logarítmica. Considerando que funciones como ValueKeySetReq7 no es como Values que dependen de un rango puesto por el usuario, pensamos que de pronto la razón por la cual la función se ve logarítmicas es porque la cantidad de trabajos de Polonia no cambia mucho entre los archivos más grandes, si no que lo que cambia entre esos archivos es que hay más trabajos de otros países.

Carga de datos

Pruebas Realizadas y tablas de datos:

Entrada	Tiempo (ms)	Memoria (kB)
10 pct	13836.937	309253.837
20 pct	27372.042	568744.107
30 pct	41153.530	847630.175
40 pct	54982.118	1106315.875
50 pct	66674.142	1334119.539
60 pct	75787.667	1482772.866
70 pct	78532.727	1530917.851
80 pct	80174.330	1565628.161
large	81942.261	1608319.279





Descripción

Para nuestra carga de datos hicimos:

1. ArbolReq1: El requerimiento 1 hace uso de dos árboles rojo negro anidados. Se tiene un árbol dividido por fechas, en el que las llaves son las fechas de publicación y el valor de cada uno es otro árbol. Aquí las fechas solo tienen año, mes y día dado que eso es lo que

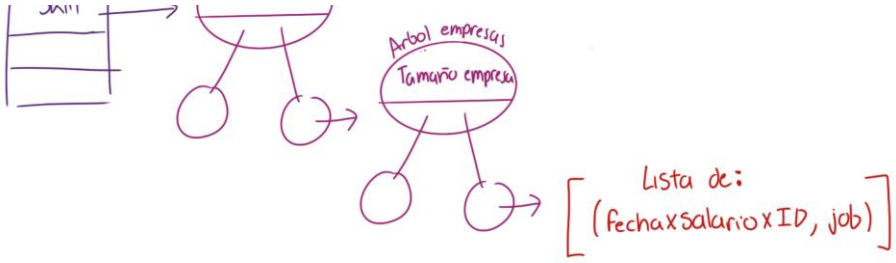
se le pide el usuario. En el segundo árbol se hace uso de una llave compuesta donde el valor es la hora de publicación, el salario y el id (el id es para que cada nodo sea una oferta). De esta forma, en el segundo árbol, las ofertas con la misma fecha y hora quedarán ordenadas según salario.

2. ArbolReq2: Para el requerimiento 2 se implementaron árboles dentro de un árbol. Primero, se creó un árbol cuyas llaves son los salarios mínimos de las ofertas. Luego, el valor de cada nodo es otro árbol, cuyas llaves son la fecha de publicación con hora y el id del trabajo. Esto busca que las ofertas que tengan el mismo salario estén ordenadas por fecha de publicación.

3. TablaReq37: Una tabla de hash que tiene como llave los años de las ofertas. Como valor tiene una tabla de hash que tiene como llave los países de ese año. Los valores de esa tabla de hash es la información que contiene:
    - Un contador de ofertas totales (usado para el requisito 7 si el usuario pide ubicación o nivel de experticia).
    - Un contador de ofertas totales con habilidades (usado para el requisito 7 si el usuario pide habilidad).
    - Una tabla de hash que contiene como llave el nivel de experiencia y como valor un arbol que contiene los trabajos (guardados con la llave compuesta de published_atXsalarioXid) y una lista que contiene los trabajos de ese año en ese país en ese nivel de experticia.
    - Un arbol que contiene como llave un string compuesto por el número de ofertas del nivel de experiencia y el nombre del nivel de experiencia. Como valor tiene un arbol que contiene los trabajos (guardados con la llave compuesta de published_atXsalarioXid) y una lista que contiene los trabajos de ese año en ese país en ese nivel de experticia.
    - Una tabla de hash que contiene como llave la ubicación y como valor una lista que contiene los trabajos de ese año en ese país y con esa ubicación.
    - Un arbol que contiene como llave un string compuesto por el número de ofertas de la ubicación y el nombre de la ubicación. Como valor tiene una lista que contiene los trabajos de ese año en ese país y con esa ubicación.
    - Una tabla de hash que contiene como llave el nombre de la habilidad y como valor una lista que contiene los trabajos de ese año en ese país y con esa habilidad.
    - Un arbol que contiene como llave un string compuesto por el número de ofertas de la habilidad y el nombre de la habilidad. Como valor tiene la palabra “cuervo” (para el requisito nos importa el orden de las llaves, no necesitamos guardar nada en el valor).
    - Una lista que contiene todos los trabajos que tienen una oferta.

Los arboles tienen una llave compuesta: cantidad de ofertas ∧ propiedad

4. TablaReq4: Esta tabla de hash se compone de llaves correspondientes a ciudades cuyos valores son otras tablas. En la segunda tabla la llave es la ubicación del trabajo (remote, partly_remote, office) y su valor es un árbol. Los árboles tienen llaves compuestas que constan de la fecha, el salario y el id. Esto se hace con el objetivo de que las ofertas se ordenen por fecha, y en caso de tener la misma, por salario.

5. 

ıbilidades y veles de la tamaño de como valor ese tamaño or un string quisito 5

6. TablaReq6: La estructura inicia con una tabla hash donde cada ciudad es almacenada como una clave, facilitando el acceso rápido a los datos asociados con cada ciudad particular. Esta tabla está vinculada a un árbol de salarios, el cual organiza las ofertas de empleo dentro de cada ciudad según rangos de salario, permitiendo filtrar eficientemente las ofertas que se encuentran dentro de un rango de salario específico. A su vez, cada nodo en el árbol de salarios se conecta a un árbol de fechas organizado por año, mes y día, lo que posibilita un filtrado adicional por fecha específica, ideal para identificar ofertas activas o listadas en periodos concretos. Dentro de cada fecha, los datos se estructuran aún más detalladamente en un tercer nivel mediante otro árbol que organiza la información por hora, salario exacto y un identificador único para cada oferta. Este nivel de detalle permite operaciones como actualizaciones, eliminaciones o consultas muy precisas sobre ofertas individuales. Finalmente, los nodos terminales de esta estructura contienen la información específica de cada oferta de trabajo, como la descripción del puesto, requisitos, y duración del contrato, almacenados en nodos de trabajo.

7. Tabla_employmentTypesID: Una tabla de hash que contiene como llaves los IDs de cada trabajo y como valor un diccionario del trabajo con su mayor y menor salario posible.
8. Tabla_multilocationID: Una tabla de hash que contiene como llaves los IDs de los trabajos que tenían más de una localización y como valor un diccionario del trabajo.
9. Tabla_skillsID: Una tabla de hash que contiene como llave los IDs de cada trabajo y como valor una lista de sus skills.

En la carga de datos primero se cargan las tablas 7-9 de los archivos auxiliares porque la información en ellas es utilizada para guardar cada trabajo con la información de sus salarios y hablidades en las tablas y arboles de los requisitos. Después se carga el archivo jobs y se guarda cada trabajo en cada estructura de cada requisito.

```
55  def load_data(control,muestra, memflag=True):
56      """
57      Carga los datos del reto
58      """
59      # TODO: Realizar la carga de datos
60      # toma el tiempo al inicio del proceso
61      start_time = getTime()
62
63      # inicializa el proceso para medir memoria
64      if memflag is True:
65          tracemalloc.start()
66          start_memory = getMemory()
67
68      data_struct = control['model']
69
70      load_multilocations(data_struct,muestra)
71      load_skills(data_struct, muestra)
72      load_employmentTypes(data_struct, muestra)
73      numjobs=load_jobs(data_struct, muestra)
74      model.loadArbolReq7(data_struct)
75
101 def load_jobs(data_struct, muestra):
102     jobfile = cf.data_dir + muestra+'-jobs.csv'
103     input_file = csv.DictReader(open(jobfile, encoding='utf-8'),delimiter=';')
104     sumatoria=0
105
106     for job in input_file:
107         salarioJobPareja=mp.get(data_struct["tabla_employmentTypesID"], job["id"])
108         salarioJobValue=me.getValue(salarioJobPareja)
109         job["salary_from"]=salarioJobValue["salary_from"]
110         job["salary_to"]=salarioJobValue["salary_to"]
111
112         skillsJobPareja=mp.get(data_struct["tabla_skillsID"], job["id"])
113         skillsJobValue=me.getValue(skillsJobPareja)
114         job["skills"]=skillsJobValue["skills"]
115
116         job["skills_required"] = []
117         for skill in lt.iterator(job["skills"]):
118             job["skills_required"].append(skill["name"])
119
120         model.add_arbolReq1(data_struct, job)
121         model.add_arbolReq2(data_struct, job)
122         model.add_tablaReq4(data_struct, job)
123         model.add_tablaReq5(data_struct, job)
124         model.add_tablaReq6(data_struct, job)
125         model.add_tablaReq37(data_struct,job)
126
127     return sumatoria
```

Para el arbolReq1, primero se obtiene el árbol de data_struct y luego se llama a la función updateArbolDias. Esta función obtiene la fecha sin hora de cada publicación y la asigna como llave. Así mismo, si la llave ya existe, se obtiene su valor; si no existe, se crea. Luego la función de addJobInd_arbolHoraXSalario se encarga de crear otro subárbol como el valor de cada llave. Este árbol tendrá como llave compuesta la hora, el salario y el id.

```
132 def add_arbolReq1(data_struct, job):
133     mapa=data_struct["arbolReq1"]
134     updateArbolDias(mapa, job)
135     pass
```

```
433  def updateArbolDias(map, job):
434
435      fechaHora = job["published_at"].split("T")
436      fechaYMD=fechaHora[0]
437      #Busco el día en el arbol
438      entry = om.get(map, fechaYMD)
439
440      if entry is None:
441          #Si el día no existo añado el arbol del día
442          arbolJobs = om.newMap(omaptype="RBT", cmpfunction=defaultfunction_invertida)
443          om.put(map, fechaYMD, arbolJobs)
444      else:
445          arbolJobs = me.getValue(entry)
446
447      #Le añado a ese arbol el trabajo
448      addJobInd_arbolHoraxSalario(arbolJobs, job)
449
450      return map
451
571  def addJobInd_arbolHoraxSalario(arbol, job):
572      fecha=job["published_at"].split("T")
573      hora=fecha[1]
574      salarioKey=str(job["salary_from"])
575
576      #Cambio el salario para que quede con 6 cifras para la llave
577      while len(salarioKey)<6:
578          salarioKey="0"+salarioKey
579
580      #Creo la llave que primero tiene hora, después el salario, y por último el id del trabajo
581      key=hora+"X"+salarioKey+"X"+job["id"]
582
583      #Cada hoja tiene un solo trabajo
584      om.put(arbol, key, job)
585
586      pass
```

Para cargar los datos del arbolReq2, se sigue un proceso similar al del requerimiento 1. La función updateArbolSalario, crea nodos cuyas llaves sean el salario mínimo de las ofertas. Si ya existe, solo se obtiene el valor de la llave. Luego la función addSubArbol_fechas, hace que cada valor sea otro árbol cuyos nodos tengan como llave la fecha de publicación y el id.

```
136
137  def add_arbolReq2(data_struct, job):
138      mapa=data_struct["arbolReq2"]
139      updateArbolSalario(mapa, job)
140      pass
```

```
451
452     def updateArbolSalario (map, job):
453         salarioKey=str(job["salary_from"])
454
455         #Cambio el salario para que quede con 6 cifras para la llave
456
457         salarioKey=salarioKey.zfill(6)
458
459         entry = om.get(map, salarioKey)
460
461         if entry is None:
462             #Si el árbol del salario existe
463             arbolJobs = om.newMap(omaptype="RBT", cmpfunction=defaultfunction_invertida)
464             om.put(map, salarioKey, arbolJobs)
465         else:
466             arbolJobs = me.getValue(entry)
467
468         addSubArbol_fechas(arbolJobs, job)
469         return map
def addSubArbol_fechas(arbol, job):
    fechaKey = job["published_at"]
    key = fechaKey + "X" + job["id"]

    om.put(arbol, key, job)
    pass
```

Para la tabla 3 (la del requisito 3 y 7) primero se sacan todos los valores del trabajo que necesitan conversión para crear llaves con el formato indiciado (244-256). Después se saca el valor del año y del país del trabajo de sus tablas de hash. Si no existe el valor de ese año o de ese país, se crean. Durante esto tambíen se añade a los contadores de ofertas totales del año y país que serán usados en el requisito 7 (258-285). Después, se añade el trabajo en la tabla de hash de nivel de experiencia (289-299) y se hace los mismo para la tabla de habilidades (302-315) y de las ubicaciones (319-326). Esto se hace en la función del model .add_tablaReq37.

Después de haber hecho lo anterior para todos los trabajos, se llama la función del model loadArbolReq7. Esta función recorre todos los años y todos los países para añadirle a los árboles de ese país (habilidad, experiencia y ubicación) la información necesaria para graficar. Ya que estos árboles tienen como llaves el número de ofertas de cada propiedad y el nombre de la propiedad, se necesitaba primero contar cuantas ofertas tenía cada propiedad (por ende, la tabla de hash). Por eso, se llama a la función tablaToArbolReq7 que recorre todos los valores de la tabla de la propiedad, mira cuantos trabajos tiene cada propiedad, y añade al árbol de ese tipo de propiedad usando una llave compuesta del número de ofertas y el nombre de la propiedad.

```python
def add_tablaReq37(data_struct, job):

    salaryKey=str(job["salary_from"])
    #Cambio el salario para que quede con 3 cifras para la llave
    while len(salaryKey)<6:
        salaryKey="0"+salaryKey
    jobKey=job["published_at"]+"X"+salaryKey+"X"+job["id"]

    fecha=job["published_at"].split("T")
    fecha2=fecha[0].split("-")
    yearKey=fecha2[0]
    countryKey=job["country_code"]
    #Para que no vaya a sacar error con los trabajos que no tienen nada en pais
    if countryKey == "":
        countryKey="Undefined"

    #Saco el valor del año
    entry = mp.get(data_struct["tablaReq37"], yearKey)
    #Si si existe cojo el value de ese año
    if entry is None:
        yearValue = newAnioValue(yearKey)
        mp.put(data_struct["tablaReq37"], yearKey, yearValue)
    #Si no existe creo una nueva key,value pair
    else:
        yearValue = me.getValue(entry)
    #Le sumo 1 a las ofertas totales del años
    yearValue["ofertas_totales"]+=1

    #Saco el valor del pais dentro de ese año
    tablaPaises=yearValue["paises"]
    paisPareja = mp.get(tablaPaises, countryKey)
    #Si si existe cojo el value de ese año
    if paisPareja is None:
        paisValue = newPaisValue(countryKey)
        mp.put(tablaPaises, countryKey, paisValue)
    #Si no existe creo una nueva key,value pair
    else:
        paisValue = me.getValue(paisPareja)
    #Le sumo 1 a las ofertas totales del país
    paisValue["ofertas_totales"]+=1

    if lt.size(job["skills"])>0:
        #Le sumo 1 a las ofertas totales que seran usadas para los skills
        paisValue["ofertas_totales_skills"]+=1


    #Añado a la tabla de experiencia de ese país en ese año en el trabajo
    expPareja = mp.get(paisValue["tabla_experience_levels"], job["experience_level"])
    if expPareja is None:
        expValue = newExpValue(job["experience_level"])
        mp.put(paisValue["tabla_experience_levels"], job["experience_level"], expValue)
    else:
        expValue = me.getValue(expPareja)

    arbolJobs=expValue["jobs"]
    om.put(arbolJobs, jobKey, job)
    lt.addLast(expValue["jobs_lista"], (jobKey,job))

    #Añado a la tabla de habilidades
    for skill in lt.iterator(job["skills"]):
        skillName=skill["name"]

        skillPareja = mp.get(paisValue["tabla_skills"], skillName)
        if skillPareja is None:
            skillValue = newSkillValueReq7(skillName)
            mp.put(paisValue["tabla_skills"],skillName, skillValue)
        else:
            skillValue = me.getValue(skillPareja)

        lt.addLast(skillValue["jobs_lista"], (jobKey,job))

    if lt.size(job["skills"])>0:
        lt.addLast(paisValue["lista_jobs_skills"], (jobKey,job))


    #Añado a la tabla de ubicacion
    parejaUbicacion = mp.get(paisValue["tabla_workplace_type"], job["workplace_type"])
    if parejaUbicacion is None:
        ubicacionValue = newUbicacionValueReq7(job["workplace_type"])
        mp.put(paisValue["tabla_workplace_type"], job["workplace_type"], ubicacionValue)
    else:
        ubicacionValue = me.getValue(parejaUbicacion)

    lt.addLast(ubicacionValue["jobs_lista"], (jobKey,job))
```

Funciones que pasan de tabla a árbol:

```
471  def tablaToArbolReq7(mapa, arbol, jobInd):
472
473      #Recorro cada uno de los valores de categoria del pais
474      for pos2 in range(lt.size(mapa['table'])):
475
476          entryCategoria = lt.getElement(mapa['table'], pos2+1)
477
478          if (entryCategoria['key'] is not None and entryCategoria['key'] != '__EMPTY__'):
479
480              size=str(lt.size(entryCategoria["value"]["jobs_lista"]))
481              name=entryCategoria['key']
482
483              #Cambio la cantidad de trabajos para que quede con 7 cifras para la llave
484              size=size.zfill(8)
485
486              size_name=size+"X"+name
487
488              if jobInd:
489                  om.put(arbol, size_name, entryCategoria["value"])
490              else:
491                  om.put(arbol, size_name, "Cuervo")
492
493      pass
494
495  def loadArbolReq7(data_struct):
496      for year in lt.iterator(mp.valueSet(data_struct["tablaReq37"])):
497          for paisValue in lt.iterator(mp.valueSet(year["paises"])):
498              tablaToArbolReq7(paisValue["tabla_experience_levels"], paisValue["arbol_experience_levels"], True)
499              tablaToArbolReq7(paisValue["tabla_workplace_type"], paisValue["arbol_workplace_type"], True)
500              tablaToArbolReq7(paisValue["tabla_skills"], paisValue["arbol_skills"], False)
501
```

Para la carga de la tabla para el requerimiento 4 primero se obtiene la ciudad y tipo de ubicación de la oferta. Luego, se revisa si la ciudad ya existe en la tabla. Si ya existe, se obtiene su valor; si no existe, se añade con la función newCityValueWorkType. Esta función hace que el valor de la ciudad sea un diccionario donde la primera llave es la ciudad y la segunda es workplace_type. La segunda llave tendrá con valor otra tabla. Luego, se obtiene esta segunda tabla y se busca si en sus llaves ya está el tipo de ubicación de la nueva oferta. Si ya está, se obtiene el valor; si no está se crea con la función newWorkTypeValue. Esta función hace que el valor de cada workplace type sea un diccionario, cuya primera llave es el workplace_type y la segunda llave contiene como valor un arbol. Finalmente, se llama a la función addJob_arbolFechaXSalario. La función se encarga de añadir al arbol nodos cuyas llaves sean la fecha de publicación, el salario y el id y sus valores sean las ofertas de trabajo.

```python
def add_tablaReq4(data_struct, job):

    ciudad = job["city"]
    work_type  = job["workplace_type"]

    if ciudad == "":
        ciudad="Undefined"

    #Chequeo si existe la ciudad en la tabla
    existPais=mp.contains(data_struct["tablaReq4"], ciudad)
    #Si ya existe el pais cojo el value de ese pais
    if existPais:
        entryCity = mp.get(data_struct["tablaReq4"], ciudad)
        cityValue = me.getValue(entryCity)
    #Si no existe creo un nuevo key, value pair
    else:
        cityValue = newCityValueWorkType(ciudad)
        mp.put(data_struct["tablaReq4"], ciudad, cityValue)


    #Chequeo si el nivel que quiero añadir existe dentro de ese pais
    existWorkType = mp.contains(cityValue["workplace_type"], work_type)
    #Si si existe cojo ese experience level
    if existWorkType:
        entryWorkType = mp.get(cityValue["workplace_type"], work_type)
        #Cojo el value de ese entry
        workTypeValue = me.getValue(entryWorkType)
    #Si no existe creo un nuevo key, value pair
    else:
        workTypeValue = newWorkTypeValue(work_type)
        mp.put(cityValue["workplace_type"], work_type, workTypeValue)
    #Añado a la lista de trabajos de ese experience level


    addJob_arbolFechaxSalario(workTypeValue["arbol"], job)

    pass
```

```
181  def newCityValueWorkType(ciudad):
182      entry = {'city': "", "workplace_type": None}
183      entry['city'] = ciudad
184      entry['workplace_type'] = mp.newMap(numelements=3, maptype="PROBING", loadfactor=0.5,
185                                          cmpfunction=compare_mapId)
186      return entry
187
188  def newWorkTypeValue(work_type):
189      entry = {"workplace_type": "", "arbol": None}
190      entry["workplace_type"] = work_type
191      entry["arbol"] = om.newMap(omaptype='RBT', cmpfunction=defaultfunction_invertida)
192
193      return entry
588  def addJob_arbolFechaxSalario(arbol, job):
589      fecha=job["published_at"]
590      salarioKey=str(job["salary_from"])
591
592      #Cambio el salario para que quede con 6 cifras para la llave
593
594      salarioKey = salarioKey.zfill(6)
595
596      #Creo la llave que primero tiene hora, después el salario, y por último el id del trabajo
597      key = fecha+"X"+salarioKey+"X"+job["id"]
598
599      #Cada hoja tiene un solo trabajo
600      om.put(arbol, key, job)
601
602      pass
```

Para la tabla del requisito 6:

```
    return tree
```

La función principal, **add_tablaReq6**, es responsable de agregar cada oferta de trabajo a esta estructura. Comienza extrayendo información clave de cada trabajo, como la ciudad, el salario

(formateado a seis dígitos para estandarización), la fecha de publicación y un identificador único del trabajo.

Una vez extraída esta información, la función procede a organizarla dentro de la estructura de datos. Utiliza funciones auxiliares como **get_or_create_rbt_mp** y **get_or_create_rbt_om** para asegurar que cada nivel de la estructura de datos tenga el árbol rojo-negro correspondiente listo para su uso. En el primer nivel, se maneja un árbol para salarios dentro de cada ciudad. Si un árbol para un salario específico no existe dentro de la ciudad correspondiente, se crea uno nuevo. Este proceso se repite en el siguiente nivel para las fechas dentro del árbol de salarios, y luego para los detalles más finos dentro del árbol de fechas, donde se combina la hora, el salario y el ID para formar una clave única.

Las funciones auxiliares **get_or_create_rbt_mp** y **get_or_create_rbt_om** son fundamentales para esta estructura. **get_or_create_rbt_mp** se encarga de manejar los árboles dentro de la tabla de hash, creando un nuevo árbol rojo-negro si la clave específica no existe ya. Por otro lado, **get_or_create_rbt_om** hace lo mismo, pero dentro de un mapa ordenado, lo que es crucial para mantener la organización y eficiencia en el manejo de las fechas y otros atributos que requieren ordenamiento específico.

Para la tabla 5 (la del requisito 5), al principio se saca la información del trabajo que será usada para sus llaves (líneas 197-209). Después se recorre sus habilidades. Primero se chequea si la habilidad ya está en la tabla de hash, en el caso que no se ingresa en la tabla de has la nueva habilidad, y se saca el valor. Ese valor es el arbol de niveles, del cual se saca el árbol de empresas usando om.get() con el skill level de la habilidad. Si no existe, se crea este arbol y se ingresa en el arbol de niveles. Por último, se saca la lista de trabajos de ese nivel y tamaño de empresas y se añade la tupla del trabajo a esa lista (si no existe esa lista, se crea).

```python
def add_tablaReq5(data_struct, job):

    salaryKey=str(job["salary_from"])
    #Cambio el salario para que quede con 3 cifras para la llave
    while len(salaryKey)<6:
        salaryKey="0"+salaryKey
    jobKey=job["published_at"]+"X"+salaryKey+"X"+job["id"]

    #Cambio el company size para que tenga 7 cifras o ---- si es undefined
    if job["company_size"]=="Undefined":
        comp_size="------"
    else:
        comp_size=job["company_size"].zfill(7)

    empresaKey=comp_size
    empresaKey=comp_size

    for skill in lt.iterator(job["skills"]):
        skillName=skill["name"]
        skill_level=skill["level"]

        if mp.contains(data_struct["tablaReq5"], skillName):
            parejaSkill = mp.get(data_struct["tablaReq5"], skillName)
            arbolNiveles = me.getValue(parejaSkill)
        else:
            arbolNiveles=om.newMap(omaptype="RBT")
            mp.put(data_struct["tablaReq5"], skillName, arbolNiveles)

        parejaArbolEmpresas=om.get(arbolNiveles, skill_level)

        if parejaArbolEmpresas is None:
            arbolEmpresas=om.newMap(omaptype="RBT")
            om.put(arbolNiveles, skill_level, arbolEmpresas)
        else:
            arbolEmpresas=me.getValue(parejaArbolEmpresas)

        pareja_listaJobs=om.get(arbolEmpresas, empresaKey)

        if pareja_listaJobs is None:
            listaJobs=lt.newList("ARRAY_LIST")
            om.put(arbolEmpresas, empresaKey, listaJobs)
        else:
            listaJobs=me.getValue(pareja_listaJobs)

        lt.addLast(listaJobs, (jobKey, job))

    pass
```

Para la tabla 7 nos aseguramos de que antes de que añadieramos el trabajo, en la función add_tabla_employmentTypesID() cambiaramos su divisa a USD para que hubiera una comparación justa entre los salarios en el caso que tuviera una divisa distinta. Además, previamente nos aseguramos de que estuvieramos añadiendo el trabajo con su mayor salary_to y menor salary_from en el caso que estuviera ofrecido para b2b y permanent en la función load_employmentTypes(). Esto último tomo provecho de que el archivo viene ordenado de tal manera que las ofertas que tienen el mismo ID van una tras la otra y que la oferta b2b tenía los salario más altos.

```python
def load_employmentTypes(data_struct, muestra):
    skillsFile = cf.data_dir + muestra + '-employments_types.csv'
    input_file = csv.DictReader(open(skillsFile, encoding='utf-8'),delimiter=';')
    prev=None

    for employmentType in input_file:
        current=employmentType
        if prev==None:
            model.add_tabla_employmentTypesID(data_struct, employmentType)
        elif prev["id"]==employmentType["id"]:
            NewEmploymentType=employmentType
            if prev["type"]=="b2b":
                NewEmploymentType["salary_to"]=prev["salary_to"]
            else:
                NewEmploymentType["salary_from"]=prev["salary_from"]
            model.add_tabla_employmentTypesID(data_struct, NewEmploymentType)
        else:
            model.add_tabla_employmentTypesID(data_struct, employmentType)

        prev=current


    return None
def add_tabla_employmentTypesID(data_struct, employType):
    #Cambio de divisa

    if  employType["currency_salary"] == "pln":
        employType["salary_from"] = round(float(employType["salary_from"]) * 0.25)
        employType["salary_to"] = round(float(employType["salary_to"]) * 0.25)

    elif  employType["currency_salary"] == "gbp":
        employType["salary_from"] = round(float(employType["salary_from"]) * 1.24)
        employType["salary_to"] = round(float(employType["salary_to"]) * 1.24)


    elif  employType["currency_salary"] == "eur":
        employType["salary_from"] =round(float(employType["salary_from"]) * 1.07)
        employType["salary_to"] = round(float(employType["salary_to"]) * 1.07)


    elif  employType["currency_salary"] == "chf":
        employType["salary_from"] =round(float(employType["salary_from"]) * 1.10)
        employType["salary_to"] = round(float(employType["salary_to"]) * 1.10)

    mp.put(data_struct['tabla_employmentTypesID'], employType['id'], employType)

```

Para la tabla 8 nos aseguramos que solo se añadan los trabajos que aparecen más de una vez en el archivo de multilocations y además de que solo se añadan una sola vez tomando provecho que si un trabajo tenía más de una multilocation sus otras locations estarían inmediatamente después.

```python
def load_multilocations(data_struct, muestra):
    skillsFile = cf.data_dir + muestra + '-multilocations.csv'
    input_file = csv.DictReader(open(skillsFile, encoding='utf-8'),delimiter=';')

    pre_prev={"id":None}
    prev={"id":None}

    for multilocation in input_file:
        current=multilocation

        #añado el elemento a multilocation si tiene más de una location y solo lo añado una vez
        if prev["id"]==pre_prev["id"] and current["id"]!=prev["id"] and prev["id"]!=None:
            model.add_tabla_multilocationID(data_struct, prev)

        pre_prev=prev
        prev=current

    return None
```

```python
def add_tabla_multilocationID(data_struct, job):
    mp.put(data_struct['tabla_multilocationID'], job['id'], job)
```

Para la tabla 9 primero se chequea si ya existe el ID. En el caso que sí, se conseguí la pareja y valor de ese ID y se añade la habilidad a su lista. En el caso que no, se crea una nueva pareja, se inserta en la tabla y se añade la habilidad a su lista.

```python
def load_skills(data_struct, muestra):
    skillsFile = cf.data_dir + muestra + '-skills.csv'
    input_file = csv.DictReader(open(skillsFile, encoding='utf-8'),delimiter=';')
    for skill in input_file:
        model.add_tabla_skillsID(data_struct, skill)

    return None
```

```python
def add_tabla_skillsID(data_struct, skill):
    id = skill["id"]

    if mp.contains(data_struct["tabla_skillsID"], id):
        entrySkill = mp.get(data_struct["tabla_skillsID"], id)
        listSkills = me.getValue(entrySkill)
        lt.addLast(listSkills["skills"], skill)
    else:
        listSkills=newSkillsValue(id)
        mp.put(data_struct["tabla_skillsID"], id, listSkills)
        lt.addLast(listSkills["skills"], skill)
```

```python
def newSkillsValue(id):
    entry = {'id': "", "skills": None}
    entry['id'] = id

    entry['skills'] = lt.newList('SINGLE_LINKED')

    return entry
```