ANÁLISIS DEL RETO

Andrés Felipe Rodriguez Acosta, af.rodrigueza12345@uniandes.edu.co, 202322292
Juan David Gutierrez Rodriguez, jd.gutierrezr123@uniandes.edu.co, 202316163
Samuel Rodriguez Robledo, s.rodriguezr234567@uniandes.edu.co, 202323878

Introducción

En este reto nuestro objetivo es aplicar los conocimientos aprendidos durante el módulo tres del curso Estructuras de datos y algoritmos. Nos centraremos en la implementación de árboles, tanto BST (árboles binarios de búsqueda) y RBT (árbol rojo-negro). En este documento llevaremos a cabo un análisis de la complejidad de los algoritmos empleados y su costo de ejecución.

Funciones

Requerimiento 3: *Andrés Felipe Rodriguez Acosta*

Requerimiento 4: *Juan David Gutierrez Rodriguez*

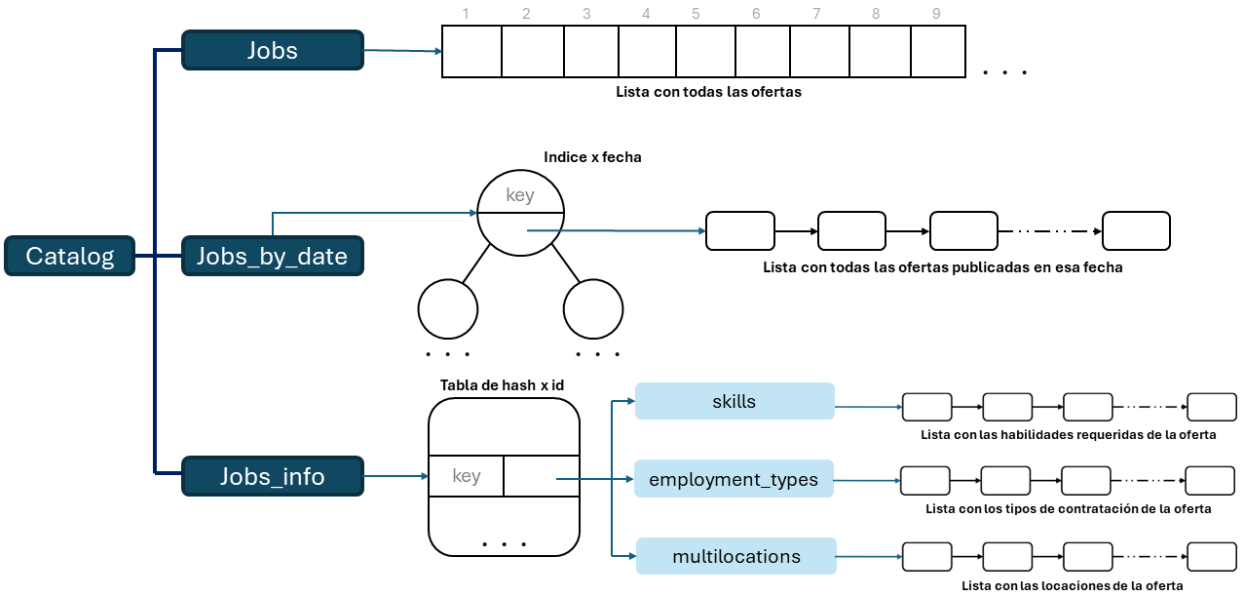
Requerimiento 5: *Samuel Rodriguez Robledo*

Índice

Carga de datos

Descripción

Para el manejo de nuestra carga de datos, hemos optado por implementar nuestras estructuras de datos siguiendo el diagrama siguiente.



Inicialmente, hemos definido la estructura de nuestro catálogo, que incluirá una lista de ofertas, un árbol de fechas organizadas de manera ascendente (de más antigua a más reciente) y una tabla hash. Esta tabla hash almacenará, para cada ID de oferta, un diccionario que contendrá listas correspondientes a cada información adicional de las ofertas.

```
def new_data_structs(data_structure, numelements):
    """
    Inicializa las estructuras de datos del modelo. Las crea de
    manera vacía para posteriormente almacenar la información.
    """
    catalog = {
        'jobs': None,
        'jobs_by_date': None,
        'jobs_info': None
    }

    # Lista con ofertas de trabajo
    catalog['jobs'] = lt.newList('ARRAY_LIST')
    # Arbol. Llave: fecha de publicacion; Valor: lista con ofertas publicadas en esa fecha
    catalog['jobs_by_date'] = om.newMap(omaptype=data_structure,
                                        cmpfunction=compare_dates)
    # Tabla de hash. Llave: id de la oferta; Valor: diccionario que contiene tres listas para la informacion adicional de la
    oferta
    catalog['jobs_info'] = mp.newMap(numelements,
                                     maptype='CHAINING',
                                     loadfactor=8)

    return catalog
```

Procedemos a agregar las ofertas a la lista. Luego, cada oferta se inserta en el árbol, utilizando como llave la fecha de publicación ("published_at"). Por último, en la tabla hash, asociamos el ID de la oferta con un diccionario que incluye tres llaves: "skills", "employment_types" y "multilocations", cada una con una lista vacía como valor inicial.

```
def new_job(catalog, data):
    """
    Crea una nueva estructura para modelar las ofertas
    """
    for key in data:
        if data[key] == 'Undefined':
            data[key] = 'Desconocido'

    # Agregar oferta a la lista 'jobs'
    add_lst(catalog['jobs'], data)
    # Agregar oferta al arbol 'jobs_by_date'
    add_job_by_date(catalog['jobs_by_date'], data['published_at'], data)
    # Crear estructura para modelar los datos
    job_info = {
        'skills': lt.newList(),
        'employment_types': lt.newList(),
        'multilocations': lt.newList()
    }
    # Agregar diccionario a la tabla de hash 'jobs_info'
    add_map(catalog['jobs_info'], mp, data['id'], job_info)
```

El último paso implica la adición de información adicional a las ofertas. Para ello, se busca el ID correspondiente en la tabla hash para obtener la lista donde se almacenará esta información adicional. En el caso específico de los tipos de contratación, se realiza una conversión del valor a dólares utilizando la función *exchange_money()*. Esta función multiplica el valor correspondiente por su tasa de cambio actual en el mercado.

```
def new_employment_type(catalog, type, data):
    """
    Crea una nueva estructura para modelar los tipos de contratacion
    """
    for key in data:
        if data[key] == '':
            data[key] = '0'

    data['salary'] = exchange_money(data['currency_salary'], (int(data['salary_from']) + int(data['salary_to'])) / 2)
    data['salary_from'] = exchange_money(data['currency_salary'], data['salary_from'])
    data['salary_to'] = exchange_money(data['currency_salary'], data['salary_to'])

    new_data(catalog, type, data)

def new_data(catalog, type, data):
    """
    Obtiene la estructura para almacenar los datos
    """
    lst = me.getValue(get_entry(catalog, mp, data['id']))[type]
    add_lst(lst, data)
```

```
def exchange_money(currency, value):
    """
    Retorna el valor en doláres
    """
    new_value = value

    if currency == 'pln':
        new_value = int(value) * 0.25
    elif currency == 'eur':
        new_value = int(value) * 1.07

    return str(new_value)
```

Entrada	Tipo de árbol a utilizar (RBT o BST), tamaño de los datos, medición de memoria.
Salidas	Primeras y últimas 3 ofertas, número de ofertas cargadas, altura del árbol, número de elementos, tiempo y memoria de ejecución.
Implementado (Sí/No)	Si fue implementado por Juan David .

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Inicializar catalogo	O(1)
Recorrer cada oferta del archivo	O(n)
Recorrer cada llave de la oferta	O(18)
Agregar a la lista, al árbol y a la tabla	O(1)
Recorrer los demás archivos	O(n)
Añadir a la lista	O(1)
Convertir salario a dólar	O(1)
TOTAL	***O(n)***

Pruebas Realizadas

Las pruebas fueron realizadas tanto con árboles BST y RBT. El algoritmo de ordenamiento para la lista de ofertas fue Merge Sort. La máquina utilizada cuenta con las siguientes capacidades.

Procesadores	Intel Core i7 -13700H
Memoria RAM (GB)	16 GB
Sistema operativo	Windows 11 Home

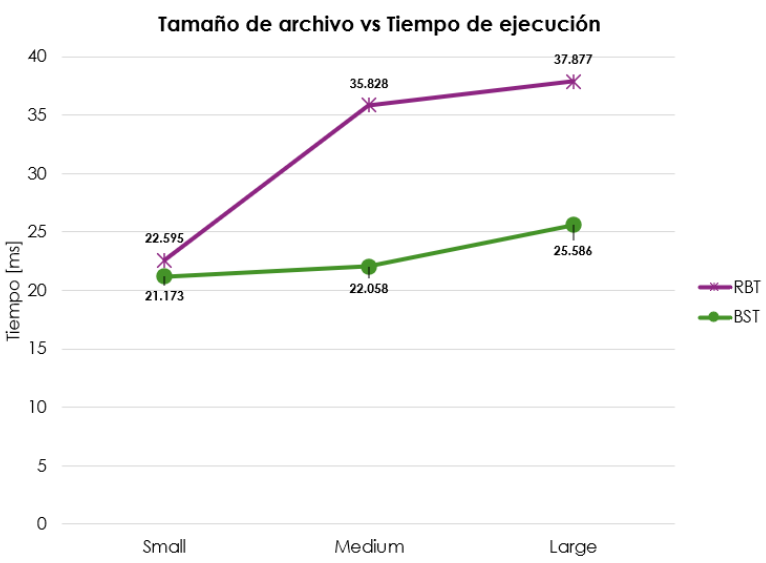
Tablas de datos

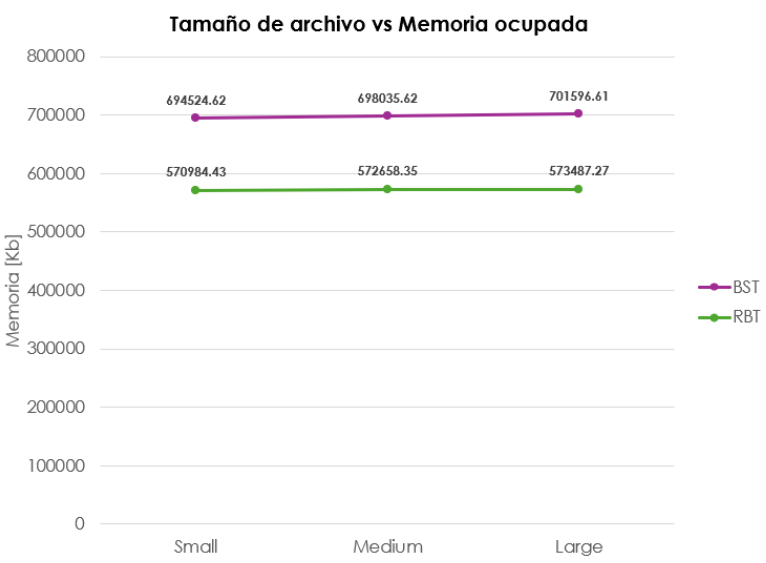
Las tablas con la recopilación de datos de las pruebas.

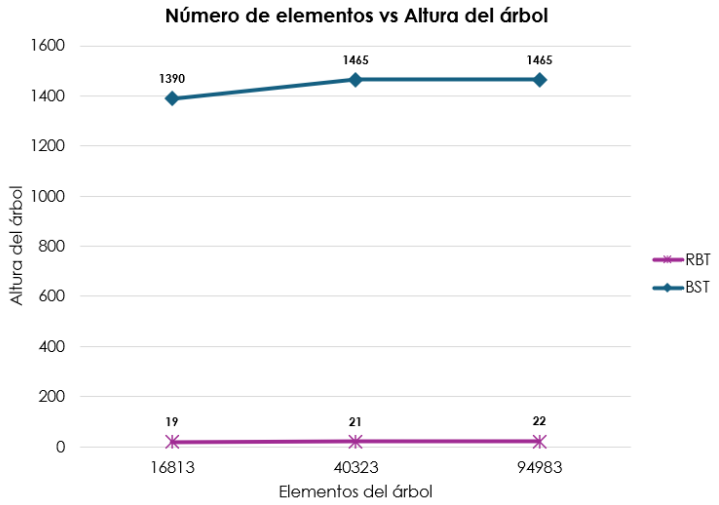
	Tamaño de archivo	Elementos	Altura	Salida	Memoria [Kb]	Tiempo [ms]
RBT	Small	16813	19	114.709 ofertas	570984.43	22.595
	Medium	40323	21	191.567 ofertas	572658.35	35.828
	Large	94983	22	203.562 ofertas	573487.27	37.877
BST	Small	16813	1390	114.709 ofertas	694524.62	21.173
	Medium	40323	1465	191.567 ofertas	698035.62	22.058
	Large	94983	1465	203.562 ofertas	701596.61	25.586

Graficas

Las gráficas con la representación de las pruebas realizadas.







Análisis

Nuestro manejo de datos funciona de manera excelente y es muy eficiente, lo que significa que podemos procesar grandes cantidades de información sin problemas con una complejidad ideal de O(n). Además, hemos organizado nuestros datos de una manera que hace que sea muy fácil cumplir con nuestras necesidades y requisitos. Esta estructura nos ayuda a trabajar de manera más rápida y eficaz, permitiéndonos adaptarnos fácilmente a cualquier cambio o nueva demanda en nuestro proyecto.

Volver al índice

Requerimiento 1

Descripción

```
def req_1(catalog, fehca_ini,fecha_fin):
    """
    Función que soluciona el requerimiento 1
    """
    jobs_by_date = catalog["jobs_by_date"]
    jobs = om.values(jobs_by_date,fehca_ini,fecha_fin)
    jobs_info = catalog['jobs_info']

    filtered_jobs = lt.newList('ARRAY_LIST')

    for joblist in lt.iterator(jobs):
        for job in lt.iterator(joblist):
            add_lst(filtered_jobs, job)
            info = me.getValue(get_entry(jobs_info, om, job['id']))
            skills = get_job_skills(info['skills'])
            job['skills'] = skills

            add_lst(filtered_jobs, job)
    req_8(filtered_jobs)
    if lt.isEmpty(filtered_jobs):

        # Si no se encuentra ninguna oferta devolver None
        return None, 0

    else:
        # Ordenar las ofertas, de mayor a menor, por la fecha de publicacion
        sort(filtered_jobs, compare_dates_samuel)

        # Obtener el numero total de ofertas publicadas segun los requisitos
        total_offers = data_size(filtered_jobs, lt)

        return filtered_jobs, total_offers
```

Para el requerimiento lo primero que se hizo fue asignar las variables, y gracias a como cargamos los datos pudimos recorrer el mapa donde las ofertas están organizadas por fecha. Posteriormente recorrimos el mapa donde se encuentras las habilidades y se tuvo que hacer un doble loop. Finalmente, se agregan las ofertas a una nueva lista y se sortean por fecha.

Entrada	• Catalogo. • Fecha inicial. • Fecha final.
Salidas	Respuesta esperada del algoritmo.
Implementado (Sí/No)	Si se implementó y lo hizo Samuel

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Asignación de variables.	O (1)
Creación de lista.	O (1)
Recorrido doble en listas.	O (n^2)
Aniadir a lista.	O (n)
TOTAL	***O (n^2)***

Pruebas Realizadas

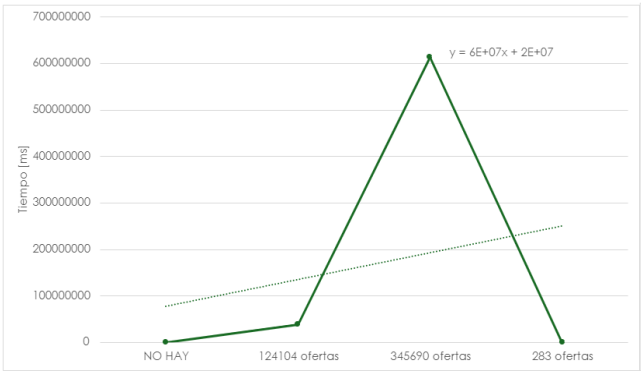
Las pruebas fueron realizadas tanto con un árbol RBT. El algoritmo de ordenamiento para la lista de ofertas fue Merge Sort. El tamaño de archivo fue large. La máquina utilizada cuenta con las siguientes capacidades.

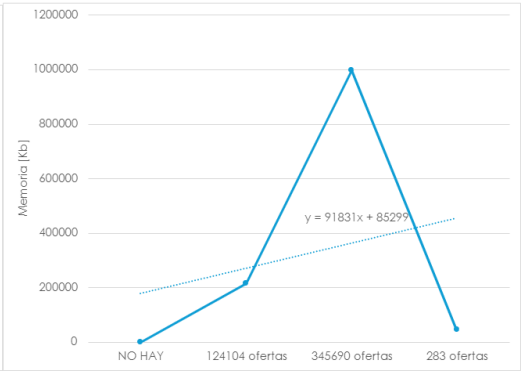
Procesadores	Intel Core i7 -13700H
Memoria RAM (GB)	16 GB
Sistema operativo	Windows 11 Home

Tablas de datos

Entradas	Salida	Memoria [Kb]	Tiempo [ms]
2020,03,0, 2021,06,01	NO HAY	0,00	0,000
2022, 07, 01, 2022,12,01	124104 ofertas	215987,17	38743470,000
2022,09,01,2024,01,01	345690 ofertas	998123,02	614326910,000
2023,05,01,2023,08,01	283 ofertas	45390,00	487980,000

Graficas





Análisis

Teniendo en cuenta las pruebas realizadas y la respuesta del programa consideramos que a pesar de que el requerimiento 1 tiene en su totalidad una complejidad de O (n^2) gracias a la manera en que se

cargaron los datos y el hecho de que no se tiene que recorrer el total de las ofertas, sino solo las que indique el usuario por parámetro el requerimiento tiene una eficiencia y respuesta muy positiva.

Volver al índice

Requerimiento 2

Descripción

Este requerimiento implicaba filtrar los datos dependiendo de un rango de salario ingresado por el usuario y devolver el total de ofertas encontradas y los trabajos filtrados. Para esto inicializamos un lista para posteriormente usar para guardar las ofertas filtradas ahí y además un contador . Para poder filtrar los datos iteramos sobre jobs, posteriormente obtenemos el salario para poder hacer la respectiva comparación, añadir al contador y filtrar los datos, después obtenemos las habilidades relacionadas a cada trabajo y añadimos todo en la lista para después modelar los datos

```python
def req_2(catalog, salario_ini, salario_fin):
    """
    Función que soluciona el requerimiento 2
    """
    jobs = catalog["jobs"]
    jobs_info = catalog['jobs_info']
    total_offers = 0
    rta = lt.newList("ARRAY_LIST")

    for job in lt.iterator(jobs):

        info = me.getValue(get_entry(jobs_info, om, job['id']))
        # Obtener el salario minimo de la oferta
        current_salary = get_min_salary(info['employment_types'])
        current_salary = get_min_salary(info['employment_types'])

        # Verificar si la oferta se encuentra entre el rango de salario
        if float(salario_ini) <= float(current_salary) <= float(salario
            total_offers += 1
            skills = get_job_skills(info['skills'])

            # Añadir datos a la oferta
            job['min_salary'] = current_salary
            job['skills'] = skills
            lt.addLast(rta, job)
    req_8(rta)

    return total_offers, rta
```

[Breve descripción de como abordaron la implementación del requerimiento]

Entrada	Rango de salarios
Salidas	Lista de trabajos filtrados y total de ofertas encontradas
Implementado (Sí/No)	Si, Andres

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Inicializar lista	O(1)
Recorrer las ofertas	O(n)
Realizar comparaciones	O(1)
Obtener habilidades y salario	O(1)

Añadir a la lista	O(1)
Obtener N ofertas y total ofertas	O(1)
TOTAL	***O(n)***

Pruebas Realizadas

Las pruebas fueron realizadas tanto con un árbol RBT. El algoritmo de ordenamiento para la lista de ofertas fue Merge Sort. El tamaño de archivo fue large. La maquina utilizada cuenta con las siguientes capacidades.

	maquina
procesador	Ryzen 7 5700u
RAM (GB)	16 GB
Sistema operativo	Windows 10 home 64 bits

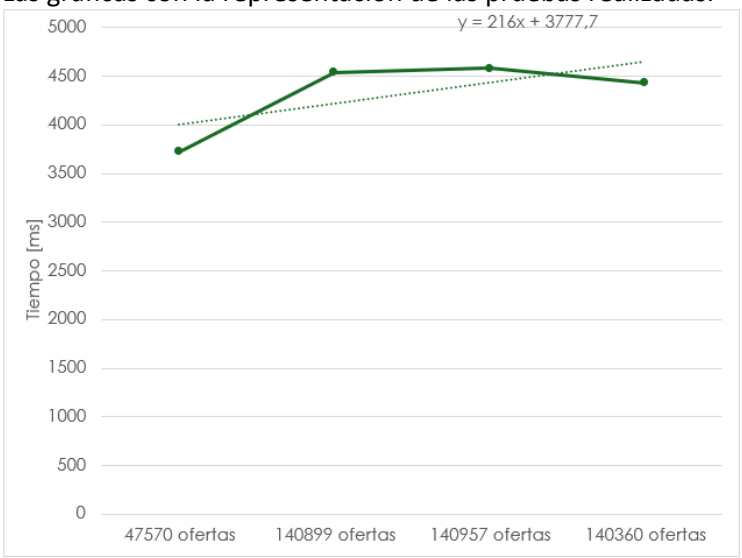
Tablas de datos

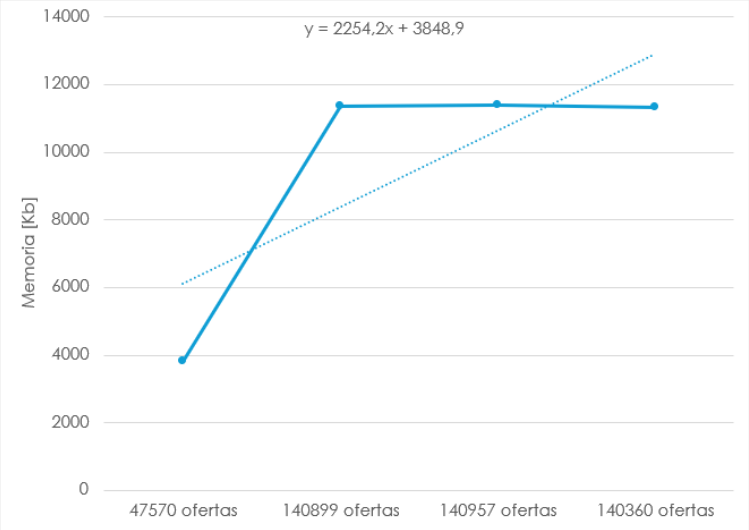
Las tablas con la recopilación de datos de las pruebas.

Entradas	Salida	Memoria [Kb]	Tiempo [ms]
400-3.000	47570 ofertas	3830,50	3722,500
600-12.000	140899 ofertas	11379,80	4538,500
400-12.500	140957 ofertas	11384,09	4581,700
1.000-20.000	140360 ofertas	11343,08	4428,100

Graficas

Las gráficas con la representación de las pruebas realizadas.





Análisis

Este requerimiento muestra muy buen rendimiento y eficacia, especialmente cuando se enfrenta a archivos de gran tamaño. Su complejidad O(n), asegura un procesamiento eficiente de los datos, lo que significa que su rendimiento no se ve afectado de manera significativa por el aumento en el tamaño del archivo manteniendo un buen tiempo de respuesta y un bajo uso de memoria . Además, su proceso de ejecución es directo y de fácil comprensión, lo que facilita su mantenimiento y entendimiento para cualquier persona que trabaje con él

Volver al índice

Requerimiento 3

Descripción

Este requerimiento implica filtrar los trabajos a partir de el país y el nivel de experticia ingresados por el usuario, además solo se requiere imprimir una cantidad de ofertas ingresada por el usuario. Para abordar este requerimiento inicializamos una lista y un contador. Posteriormente iteramos sobre jobs para poder hacer las comparaciones correspondientes a la experticia y el país y cada vez que se cumpla añadimos 1 al contador después obtenemos el salrio mínimo ofertado y las habilidades para cada respectiva oferta y lo añadimos a la lista para posteriormente agregar las ofertas filtradas a la lista. Por ultimo se hace una validación de la cantidad de ofertas a imprimir seleccionada por el usuario

```python
def req_3(catalog, n_ofertas, experticia, pais):
    """
    Función que soluciona el requerimiento 3
    """
    # TODO: Realizar el requerimiento 3
    jobs = catalog["jobs"]
    jobs_info = catalog['jobs_info']
    res = lt.newList("ARRAY_LIST")
    total_offers = 0

    for job in lt.iterator(jobs):
        if job["country_code"].lower() == pais.lower():
            if job["experience_level"].lower() == experticia.lower():

                total_offers += 1

                info = me.getValue(get_entry(jobs_info, om, job['id']))
                # Obtener el salario minimo de la oferta
                current_salary = get_min_salary(info['employment_types'])
                skills = get_job_skills(info['skills'])

                # Añadir datos a la oferta
                job['min_salary'] = current_salary
                job['skills'] = skills
                lt.addLast(res,job)

    req_8(res)
    if lt.isEmpty(res):
        # Si no se encuentra ninguna oferta devolver None
        return None, 0

    else:

        # Obtener el numero total de ofertas publicadas segun los requisitos
        total_offers = data_size(res, lt)

        # Verificar que la lista sea menor o igual al tamaño recibido por parametro
        if total_offers > n_ofertas:
            res = get_sublist(res, 1, n_ofertas)



    return res, total_offers
```

Entrada	Numero de ofertas a consultar, pais, experticia
Salidas	Lista de trabajos filtrados y total de ofertas
Implementado (Sí/No)	Si se implementó, Andres

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Inicializar lista	O(1)
Recorrer las ofertas	O(n)
Realizar comparaciones	O(1)
Obtener habilidades y salario	O(1)

Añadir a la lista	O(1)
Obtener N ofertas y total ofertas	O(1)
TOTAL	***O(n)***

Pruebas Realizadas

Las pruebas fueron realizadas tanto con un árbol RBT. El algoritmo de ordenamiento para la lista de ofertas fue Merge Sort. El tamaño de archivo fue large. La maquina utilizada cuenta con las siguientes capacidades.

	maquina
procesador	Ryzen 7 5700u
RAM (GB)	16 GB
Sistema operativo	Windows 10 home 64 bits

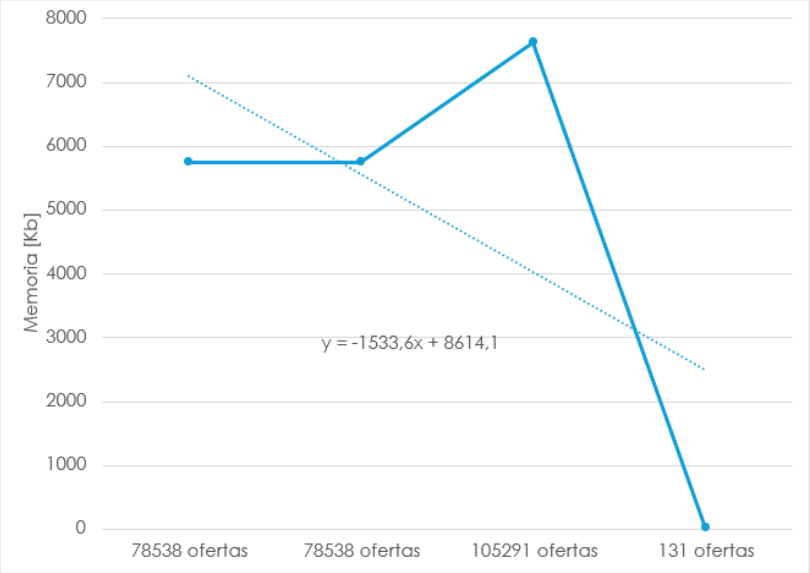
Tablas de datos

Las tablas con la recopilación de datos de las pruebas.

Entradas	Salida	Memoria [Kb]	Tiempo [ms]
2 ofertas, senior, PL	78538 ofertas	5746,30	2398,800
5 ofertas, senior , PL	78538 ofertas	5746,30	5441,200
50 ofertas, junior, PL	105291 ofertas	7617,30	6586,300
99 ofertas, mid, US	131 ofertas	10,80	1036,600

Graficas

Las gráficas con la representación de las pruebas realizadas.



Análisis

Este requerimiento muestra muy buen rendimiento y eficacia, especialmente cuando se enfrenta a archivos de gran tamaño, se puede notar que re. Su complejidad O(n), asegura un procesamiento eficiente de los datos, lo que significa que su rendimiento no se ve afectado o de manera significativa por el aumento en el tamaño del archivo manteniendo un buen tiempo de respuesta y un bajo uso de memoria . Además, su proceso de ejecución es directo y de fácil comprensión, lo que facilita su mantenimiento y entendimiento para cualquier persona que trabaje con él y el usuario puede seleccionar que tantas ofertas quiere ver en pantalla

Volver al índice

Requerimiento 4

Descripción

Para cumplir con este requerimiento, se busca identificar las N ofertas más recientes publicadas en una ciudad específica y con un tipo de ubicación determinado (remote, partly_remote u office). Para abordar este desarrollo, implementamos una variable que utiliza una lista vacía para almacenar las ofertas filtradas.

```
# Inicializar lista donde se guardaran las ofertas filtradas
filtered_jobs = lt.newList('ARRAY_LIST')

# Recorrer la lista de ofertas
for job in lt.iterator(jobs):

    # Verificar si la ciudad coincide
    if job['city'].lower() == city.lower():

        # Verificar si la ubicacion de la oferta coincide
        if job['workplace_type'] == workplace.lower():

            info = me.getValue(get_entry(jobs_info, om, job['id']))
            # Obtener el salario minimo de la oferta
            min_salary = get_min_salary(info['employment_types'])
            # Obtener las habilidades solicitadas
            skills = get_job_skills(info['skills'])

            # Añadir datos a la oferta
            job['min_salary'] = min_salary
            job['skills'] = skills

            # Añadir oferta a la lista
            add_lst(filtered_jobs, job)
```

Posteriormente, iteramos a través de cada una de las ofertas en la lista "jobs" del catálogo. Durante este proceso, comprobamos si la ciudad y el tipo de ubicación de la oferta coinciden con los parámetros especificados. Si esto se cumple, extraemos las habilidades requeridas y el salario mínimo ofrecido, agregándolos a la oferta actual. Finalmente, añadimos la oferta filtrada a la lista resultante.

```
    # Ordenar las ofertas, de mayor a menor, por la fecha de publicacion
    sort(filtered_jobs, compare_date_and_salary)

    # Obtener el numero total de ofertas publicadas segun los requisitos
    total_offers = data_size(filtered_jobs, lt)

    # Verificar que la lista sea menor o igual al tamaño recibido por parametro
    if total_offers > num_offers:
        filtered_jobs = get_sublist(filtered_jobs, 1, num_offers)

    return filtered_jobs, total_offers
```

Luego, ordenamos estas ofertas según su fecha de publicación, de la más reciente a la más antigua, y utilizamos la función get_sublist() para obtener únicamente las N ofertas más recientes.

Entrada	Ciudad, Tipo de ubicación de la oferta y numero de ofertas a listar.
Salidas	Las N ofertas filtradas por ciudad y tipo de ubicación. Total de ofertas que cumplen con los parámetros de búsqueda.
Implementado (Sí/No)	Si fue implementado por Juan David.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Inicializar lista	O(1)
Recorrer las ofertas	O(n)
Realizar comparaciones	O(1)
Obtener valor en la tabla de hash	O(1)
Obtener habilidades y salario	O(1)
Añadir a la lista	O(1)
Obtener N ofertas y total ofertas	O(1)
TOTAL	***O(n)***

Pruebas Realizadas

Las pruebas fueron realizadas tanto con un árbol RBT. El algoritmo de ordenamiento para la lista de ofertas fue Merge Sort. El tamaño de archivo fue large. La maquina utilizada cuenta con las siguientes capacidades.

Procesadores	Intel Core i7 -13700H
Memoria RAM (GB)	16 GB
Sistema operativo	Windows 11 Home

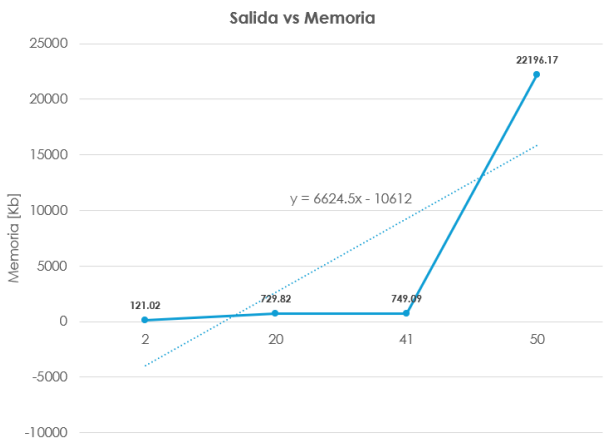
Tablas de datos

Las tablas con la recopilación de datos de las pruebas.

Entradas	Salida [ofertas]	Memoria [Kb]	Tiempo [ms]
Bogota, remote, 4	2	121.02	272.574
Krakow, partly_remote, 20	20	729.82	266.639
New York, remote, 60	41	749.09	298.089
Warszawa, office, 50	50	22196.17	4364.458

Graficas

Las gráficas con la representación de las pruebas realizadas.



Análisis

Este requerimiento presenta un excelente rendimiento y eficacia, incluso al tratar con archivos de gran tamaño. Su complejidad óptima de O(n) asegura un procesamiento eficiente de los datos, lo que significa que su rendimiento no se ve afectado significativamente por el incremento en el tamaño del archivo. Además, sus pasos de ejecución son relativamente simples y directos, lo que facilita su comprensión y mantenimiento.

Volver al índice

Requerimiento 5

Descripción

```python
def req_5(catalog, num_offers, lim_inf_comp, lim_sup_com, skill_name, lim_inf_hab, lim_sup_hab):
    """
    Función que soluciona el requerimiento 5
    """
    jobs = catalog['jobs']
    jobs_info = catalog['jobs_info']

    filtered_jobs = lt.newList('ARRAY_LIST')

    # Recorrer la lista de ofertas
    for job in lt.iterator(jobs):

        # Verificar que el tamanio de la compania no sea Undefined.
        if job["company_size"].isdigit():
            company_size = int(job["company_size"])
            # Verificar que el tamanio de la compania este dentro del rango
            if (company_size >= lim_inf_comp) and (company_size <= lim_sup_com):

                #Sacar las skills de la oferta y anadirlo a jobs
                info = me.getValue(get_entry(jobs_info, om, job['id']))

                # Obtener las skills.
                skills = get_job_skills_2(info['skills'])
                skills_2 = get_job_skills(info['skills'])


                # Obtener el salario minimo de la oferta.
                min_salary = get_min_salary(info['employment_types'])

                # Añadir datos a la oferta
                job['min_salary'] = min_salary



                job['skills'] = skills_2
```

```
            # Verificar que la skill por parametro este en la oferta
            if mp.get(skills,skill_name) is not None:

                # Asignar el nivel de la experiencia a una variable
                skill_value = me.getValue(mp.get(skills, skill_name))

                # Verificar que el nivel de experiencia se encuentre dentro del rango
                if (int(skill_value) >= lim_inf_hab) and (int(skill_value) <= lim_sup_hab):

                    add_lst(filtered_jobs, job)

    req_8(filtered_jobs)

    if lt.isEmpty(filtered_jobs):

        # Si no se encuentra ninguna oferta devolver None
        return None, 0

    else:
        # Ordenar las ofertas, de mayor a menor, por la fecha de publicacion
        sort(filtered_jobs, compare_date_and_salary_samuel)

        # Obtener el numero total de ofertas publicadas segun los requisitos
        total_offers = data_size(filtered_jobs, lt)

        # Verificar que la lista sea menor o igual al tamaño recibido por parametro
        if total_offers > num_offers:
            filtered_jobs = get_sublist(filtered_jobs, 1, num_offers)

        return filtered_jobs, total_offers
```

Básicamente se siguió un camino muy parecido al del requerimiento 4 donde se revisaron las ofertas en base al tamaño de la compañía. La gran diferencia es que en este caso para añadir las habilidades al diccionario de la oferta se usó un mapa en el cual se guardaba el nombre de la habilidad y el nivel para así poder filtrar posteriormente las habilidades por nivel. Por último, se filtraron las n ofertas (n entrando por parámetro) y se devolvió una tupla con las ofertas y con el total de estas.

Entrada	• Numero de ofertas • Límite inferior del tamaño de la compañía • Límite superior del tamaño de la compañía. • Habilidad solicitada • Límite inferior del nivel de la habilidad. • Límite superior del nivel de la habilidad.
Salidas	las N ofertas más antiguas con un nivel mínimo y máximo para una habilidad solicitada para empresas en un rango de tamaño
Implementado (Sí/No)	Si se implementó y lo hizo Samuel

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Asignar variables	O (1)

Inicializar lista	O (1)
Recorrer las ofertas	O (n)
Realizar comparaciones	O (1)
Obtener valor en la tabla de hash	O (1)
Obtener habilidades y salario	O (1)
Añadir a la lista	O (1)
Obtener N ofertas y el total de ofertas	O (1)
TOTAL	***O(n)***

Pruebas Realizadas

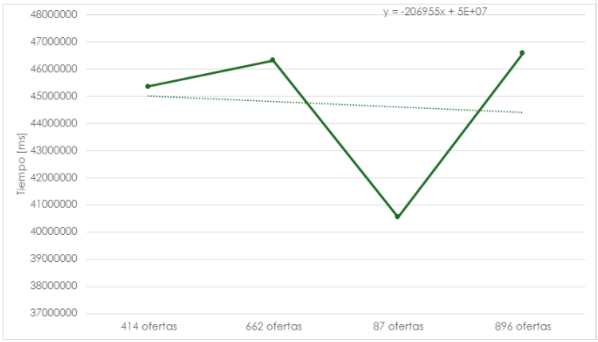
Las pruebas fueron realizadas tanto con un árbol RBT. El algoritmo de ordenamiento para la lista de ofertas fue Merge Sort. El tamaño de archivo fue large. La máquina utilizada cuenta con las siguientes capacidades.

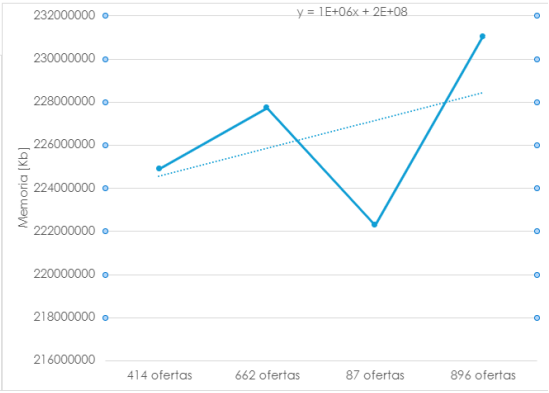
Procesadores	Intel Core i7 -13700H
Memoria RAM (GB)	16 GB
Sistema operativo	Windows 11 Home

Tablas de datos

Entradas	Salida	Memoria [Kb]	Tiempo [ms]
50, 100, 700, JAVA, 2, 3	414 ofertas	224891325,00	45366034,000
30, 0, 1000, PYTHON, 0,4	662 ofertas	227741402,00	46320112,000
100, 900, 2000, JAVASCRIPT, 1,3	87 ofertas	222282967,00	40554361,000
1000, 200, 5000, PYTHON 0, 4	896 ofertas	231024549,00	46598102,000

Graficas





Análisis

Considerando las pruebas realizadas y el resultado de la implementación podríamos decir que este código se implementó con complejidad adecuada y de manera muy efectiva, de modo que no involucra tiempos de respuesta ni mucho espacio en memoria.

Volver al índice

Requerimiento 6

Descripción

Para abordar este requerimiento, debemos identificar las N ciudades que cuentan con el mayor número de ofertas dentro de un rango específico de fechas y salarios. Para lograr esto, optamos por utilizar una tabla de hash para almacenar las ciudades junto con sus respectivas ofertas. Para filtrar las ofertas dentro del rango de fechas, empleamos el árbol "jobs_by_date", el cual, mediante la función values(), nos proporciona las ofertas dentro del intervalo de fechas especificado.

```
# Inicializar la tabla donde se guardaran las ciudades filtradas
cities = mp.newMap(1000,
                   maptype='PROBING',
                   loadfactor=0.1)

total_offers = lt.newList()


# Descomponer las tuplas de los rangos
min_date, max_date = dates[0], dates[1]
min_salary, max_salary = salary_range[0], salary_range[1]

# Obtener las ofertas en un rango de fechas
jobs = om.values(jobs_by_date, min_date, max_date)
```

Luego, evaluamos si el salario de estas ofertas está dentro del rango aceptable. Si es así, procedemos a modelar la estructura para almacenar las ciudades, la cual consiste en un diccionario que contiene el nombre de la ciudad junto con la cantidad de ofertas que tiene.

```
for lst_jobs in lt.iterator(jobs):

    for job in lt.iterator(lst_jobs):

        info = me.getValue(get_entry(jobs_info, om, job['id']))
        # Obtener el salario minimo de la oferta
        current_salary = get_min_salary(info['employment_types'])

        # Verificar si la oferta se encuentra entre el rango de salarios
        if float(min_salary) <= float(current_salary) <= float(max_salary):

            # Obtener las habilidades solicitadas
            skills = get_job_skills(info['skills'])

            # Añadir datos a la oferta
            job['min_salary'] = current_salary
            job['skills'] = skills

            # Agregar 1 a las ofertas que cumplen con los criterios de busqueda
            add_lst(total_offers,job)
            # Obtener la pareja (llave, valor) de la ciudad. Si no existe es None
            city = get_entry(cities, mp, job['city'])

            if city is None:
                # Crear estructura para modelar los datos de las ciudades
                city = {
                    'name': job['city'],
                    'offers': lt.newList('ARRAY_LIST')
                }

                # Agregar oferta a la lista de ofertas de la ciudad
                add_lst(city['offers'], job)
                # Agregar ciudad al mapa de ciudades
                add_map(cities, mp, city['name'], city)

            else:
                # Obtener valor de la ciudad
                city = me.getValue(city)
                # Agregar oferta a la lista de ofertas de la ciudad
                add_lst(city['offers'], job)
```

Una vez que hemos almacenado todas las ciudades junto con su número respectivo de ofertas, las ordenamos de mayor a menor según la cantidad de ofertas. Con esta información, podemos identificar las N ciudades con el mayor número de ofertas y hacer uso de la función get_sublist() para obtener solo las primeras N ciudades de la lista ordenada.

Entrada	Rango de fechas, rango de salarios y número de ciudades a listar.
Salidas	Ciudades filtradas. Cantidad de ofertas y de ciudades que cumplen con los criterios de búsqueda. Ciudad con mayor número de ofertas
Implementado (Sí/No)	Si fue implementado por Juan David.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Inicializar variables	O(1)
Recorrer ofertas	$O(n^2)$
Realizar comparaciones	O(1)
Obtener valor de la tabla de hash	O(1)
Inicializar diccionario para la ciudad	O(1)
Obtener valor de la tabla de hash	O(1)
Añadir al mapa	O(1)
Añadir a la lista	O(1)

TOTAL	***O(n^2)***

Pruebas Realizadas

Las pruebas fueron realizadas tanto con un árbol RBT. El algoritmo de ordenamiento para la lista de ofertas fue Merge Sort. El tamaño de archivo fue large. La maquina utilizada cuenta con las siguientes capacidades.

Procesadores	Intel Core i7 -13700H
Memoria RAM (GB)	16 GB
Sistema operativo	Windows 11 Home

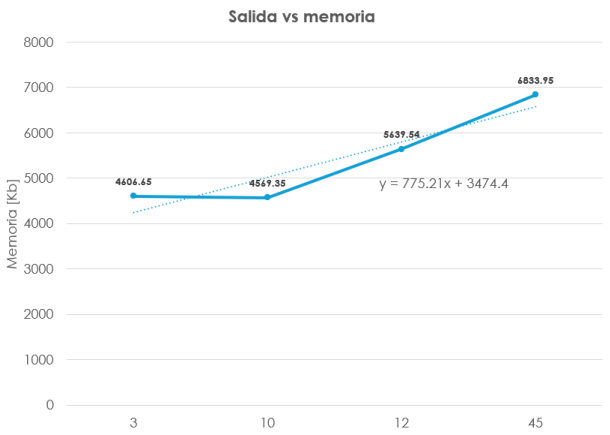
Tablas de datos

Las tablas con la recopilación de datos de las pruebas.

Entradas	Salida [ciudades]	Memoria [Kb]	Tiempo [ms]
2021-05-01, 2022-05-01, 6000, 7000, 3	3	4606.65	1345.398
2022-05-01, 2022-12-31, 1000, 5000, 10	10	4569.35	2698.369
2023-01-01, 2024-01-01, 5000, 15000, 15	12	5639.54	2869.185
2022-01-01, 2023-01-01, 10000, 50000, 50	45	6833.95	3013.331

Graficas

Las gráficas con la representación de las pruebas realizadas.



Análisis

Aunque la complejidad aparente de este requerimiento pueda ser de $O(n^2)$, es esencial recordar que no estamos procesando todos los datos en su totalidad. La función values() nos permite reducir el conjunto de ofertas a un intervalo específico de fechas, lo que disminuye significativamente la cantidad de datos a manejar. Incluso si consideramos un escenario en el que debemos abarcar todas las ofertas disponibles, estaríamos tratando esencialmente con la misma cantidad de ofertas, solo que organizadas en sublistas dentro de la estructura de datos. Aunque se requiera una doble iteración para procesar estas sublistas, se comportaría igual que recorrer todas las ofertas en la lista principal "jobs".

Volver al índice

Requerimiento 7

Descripción

Para cumplir con el requerimiento 7, que consiste en graficar la cantidad de ofertas en función de una propiedad específica (nivel de experiencia, tipo de ubicación o habilidades requeridas), en un año y país específicos proporcionados como parámetro, hemos optado por utilizar un árbol binario de búsqueda (BST).

```
jobs_by_date = catalog['jobs_by_date']
jobs_info = catalog['jobs_info']

filtered_jobs = om.newMap(omaptype='BST',
                          cmpfunction=compare_dates)
graphical_offers = lt.newList()
total_offers = 0

# Obtener el intervalo del año de busqueda
min_date, max_date = year + '-01-01', year + '-12-31'
# Obtener las ofertas en un rango de fechas
jobs = om.values(jobs_by_date, min_date, max_date)
```

En este enfoque, agregamos un diccionario al árbol para cada propiedad. Para las habilidades requeridas, el diccionario contendrá el nivel de habilidad requerido y el número de ofertas que solicitan ese nivel de habilidad. Para los tipos de ubicación y niveles de experiencia, el diccionario almacenará el nombre de la ubicación o del nivel de experiencia y la cantidad de ofertas asociadas a estas propiedades.

```
if search_area == 'skills':

    for skill in lt.iterator(job['skills']):

        entry = get_entry(filtered_jobs, om, skill['level'])

        if entry is None:
            # Crear estructura para modelar los datos de cada propiedad
            property = {
                'name': skill['level'],
                'offers': 1
            }
            # Añadir propiedad al mapa
            add_map(filtered_jobs, om, property['name'], property)

        else:
            # Obtener propiedad y agregarle 1 oferta
            property = me.getValue(entry)
            property['offers'] += 1

else:

    entry = get_entry(filtered_jobs, om, job[search_area])

    if entry is None:
        # Crear estructura para modelar los datos de cada propiedad
        property = {
            'name': job[search_area],
            'offers': 1
        }
        # Añadir propiedad al mapa
        add_map(filtered_jobs, om, property['name'], property)
```

Al finalizar el proceso, inicializamos dos listas: una para el eje x, que contendrá los nombres de las propiedades, y otra para el eje y, que contendrá el número de ofertas correspondientes a cada propiedad.

```
# Obtener los valores
properties = om.valueSet(filtered_jobs)
x = []
y = []

for property in lt.iterator(properties):
    # Descomponer los ejes X (nombre de la propiedad) y Y (numero de ofertas)
    x.append(property['name'])
    y.append(property['offers'])
```

Estas listas se utilizarán para generar la gráfica solicitada, proporcionando una visualización clara y concisa de la distribución de ofertas según la propiedad especificada para el país y año dados.

Entrada	País, año de búsqueda y propiedad para graficar.
Salidas	Los valores de los ejes. La cantidad de ofertas usadas para graficar. El numero total de ofertas. El mayor y menor valor de la gráfica.
Implementado (Sí/No)	Si fue implementado por Juan David.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Inicializar variables	O(1)

Recorrer ofertas	$O(n^2)$
Realizar comparaciones	O(1)
Obtener valor del árbol	O(1)
Inicializar diccionario para la propiedad	O(1)
Obtener valor de la tabla de hash	O(1)
Añadir al árbol	O(1)
Recorrer árbol	O(5)
Añadir a la lista de X y Y	O(1)
TOTAL	***$O(n^2)$***

Pruebas Realizadas

Las pruebas fueron realizadas tanto con un árbol RBT. El algoritmo de ordenamiento para la lista de ofertas fue Merge Sort. El tamaño de archivo fue large. La maquina utilizada cuenta con las siguientes capacidades.

Procesadores	Intel Core i7 -13700H
Memoria RAM (GB)	16 GB
Sistema operativo	Windows 11 Home

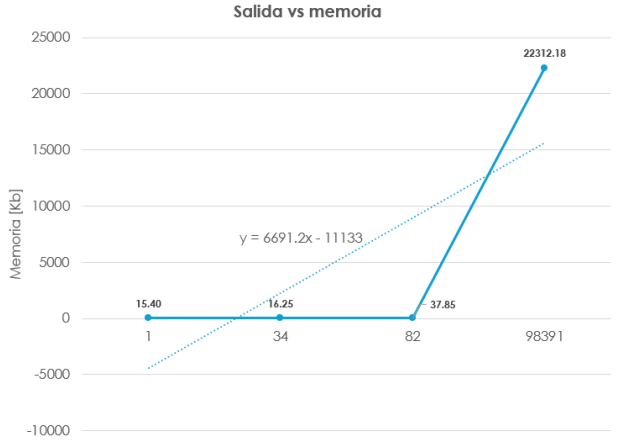
Tablas de datos

Las tablas con la recopilación de datos de las pruebas.

Entradas	Salida [ofertas]	Memoria [Kb]	Tiempo [ms]
CO, 2023, Ubicacion	1	15.40	1742.205
CH, 2023, Experiencia	34	16.25	1790.591
FR, 2022, Habilidades	82	37.85	1836.696
PL, 2022, Experiencia	98391	22312.18	1963.020

Graficas

Las gráficas con la representación de las pruebas realizadas.



Análisis

Al igual que en el requerimiento anterior, aunque la complejidad aparente de esta tarea podría ser de $O(n^2)$, consideramos que no estamos procesando la totalidad de los datos en cada iteración. La función values() nos permite reducir el conjunto de ofertas a un intervalo específico de fechas, lo que disminuye de manera significativa la carga de datos que debemos manejar.

Volver al índice

Requerimiento 8

Descripción

Para llevar a cabo el requerimiento 8, empleamos la extensión Folium, la cual nos permite crear mapas interactivos y añadir marcadores. El código comienza creando un mapa utilizando la función Map().

```
# Crear mapa
map = folium.Map()

for job in lt.iterator(jobs):
    # Obtener las coordenadas como una tupla (latitud, longitud)
    coordenates = (job["latitude"],job["longitude"])
    # Mensaje que se mostrara al hacer click sobre un marcador
    message = f"""
    {job["company_name"]}

    {job['country_code']}
    """
```

Luego, iteramos sobre la lista de ofertas recibida como parámetro para obtener las coordenadas (latitud, longitud) de cada una. Posteriormente, añadimos un mensaje que se mostrará al hacer clic sobre un marcador, el cual incluye el nombre de la empresa y el país de la oferta.

```
    # Añadir marcador al mapa
    folium.Marker(location=coordenates,
                  tooltip=job["city"],
                  popup=message).add_to(map)

# Guardar mapa para poder visualizarlo
map.save("mapa_req_8.html")
```

Finalmente, guardamos el mapa en formato HTML para poder visualizarlo posteriormente en nuestro navegador. Cada uno de nuestros requerimientos llama a la función req_8() para así poder crear el mapa interactivo.

Entrada	Lista de ofertas.
Salidas	Mapa en formato .html
Implementado (Sí/No)	Si fue implementado por Juan David y Samuel.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**

Crear mapa	O(1)
Iterar sobre las ofertas	O(n)
Obtener coordenadas	O(1)
Inicializar mensaje	O(1)
Añadir marcador	O(1)
Guardar mapa	O(1)
TOTAL	***O(n)***

Análisis

Para el requerimiento 8, las pruebas realizadas son las mismas que para cada uno de los requerimientos anteriores, dado que esta función está implementada en todos ellos. Respecto a la complejidad de nuestro algoritmo, presenta una complejidad ideal de O(n), lo que garantiza su eficacia. Sin embargo, es importante destacar que al intentar agregar un gran número de ofertas al mapa interactivo, nos enfrentamos al problema de que el navegador no puede cargarlo. Esto puede deberse a la sobrecarga de datos y recursos que implica la visualización de un gran conjunto de marcadores en el mapa.

Volver al índice