ANÁLISIS DEL RETO

Franklin Smith Fernandez Romero, 202215104, F.fernandezr@uniandes.edu.co
Manuel Santiago Prieto Hernández, 202226947, m.prietoh@uniandes.edu.co
Pablo Arango Muriel, 202220340, p.arangom@uniandes.edu.co

Requerimiento 0 - Carga de datos

```python
def load_data(control, filename):
    """
    Carga los datos del reto
    """
    # TODO: Realizar la carga de
    start = get_time()
    data=crear_lista_datos(filename)
    control["model"]= model.crear_datastructs_fecha(control["model"], data)
    control["model"] = model.crear_datastructs_casos(control["model"], data)
    control["model"] = model.agregar_datas(control["model"], data)
    end= get_time()
    deltatime = delta_time(start, end)
    print(deltatime)
    return control

def crear_lista_datos(filename):
    lista=model.new_list()
    file = cf.data_dir + filename
    input_file = csv.DictReader(open(file, encoding='utf-8'))
    for data in input_file:
        model.add_data(lista, data)

    return lista
```

Descripción

En la carga de datos primero se obtiene todos los datos de un csv y se guardan en un Array_list, para luego ser divididos por fecha y caso, primero creando un árbol por cada año, y uno por cada mes, y por cada día hay un Array_list con los accidentes de ese día ordenados del más reciente al más antiguo. En el caso del árbol casos, es un árbol binario con los diferentes casos, que contiene Array_List de los accidentes con ese caso.

Entrada	Control, FileName del archivo
Salidas	Tabla hash de árboles binarios y rojo negros

Implementado (Sí/No)	Si

Análisis de complejidad

Pasos	**Complejidad**
Crear Array List de los datos	O(n)
Crear arboles a utilizar	O(n)
Dividir por fechas	O(n)
Dividir por casos	O(n)
Ordenar ascendentemente por fecha	O(nlogn)
Total	O(nlogn)

Pruebas Realizadas

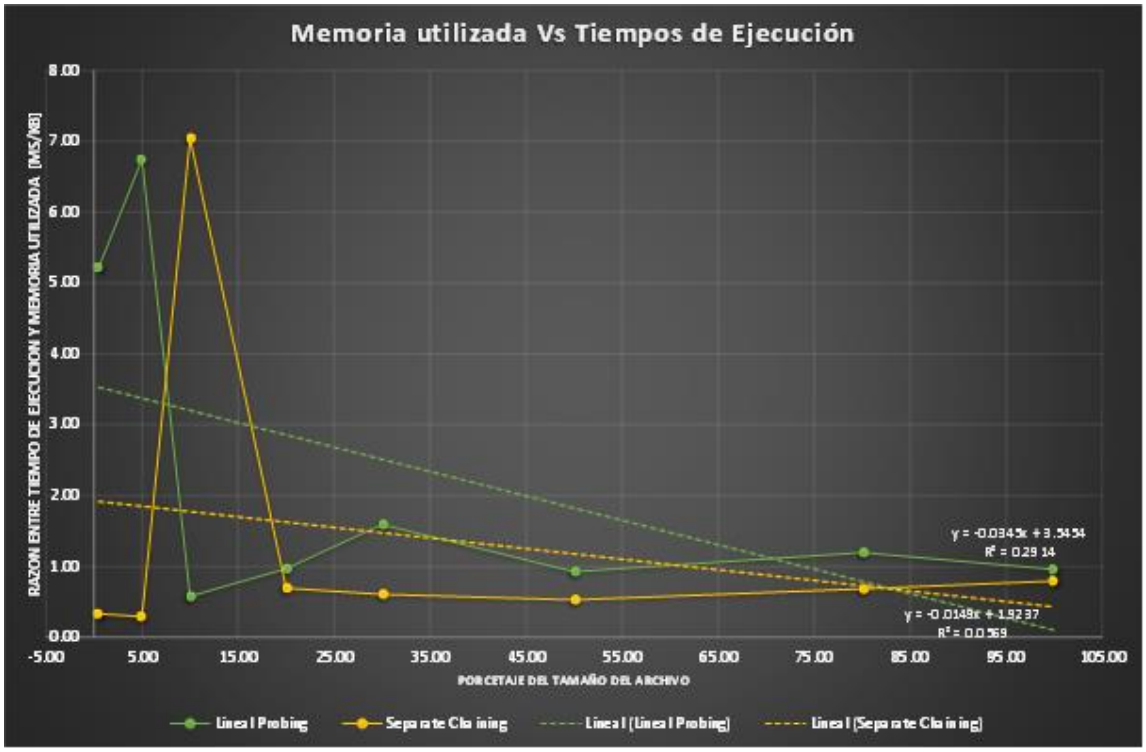
Las pruebas realizadas fueron realizadas en una maquina con las siguientes especificaciones. Los datos de entrada fueron el data structs del modelo.

Procesadores	**11th Gen Intel(R) Core(TM) i3-1115G4 @ 3.00GHz**
Memoria RAM	12 GB
Procesador	Core i3
Sistema Operativo	Windows 10

Entrada	**Tiempo (s)**
small	1700.72
5 pct	2210.16
10 pct	3806.9
20 pct	1.08
30 pct	1.20
50 pct	2.35
80 pct	3.47
large	6.87

Graficas

Las gráficas con la representación de las pruebas realizadas.



Análisis

Análisis de resultados de la implementación, tener cuenta las pruebas realizadas y el análisis de complejidad.

Según el cálculo de complejidad se obtiene que la complejidad en O(n) y analizando la gráfica, podemos ver que se acerca bastante a la linealidad teniendo pequeñas variaciones debido a procesos corriendo en segundo plano en el computador
En conclusión, si se ve una relación entre el cálculo teórico y el que se aprecia experimentalmente.

Requerimiento 1

```python
def RBT_rango_fechas(mapa, inicio, final):
    arbol=om.newMap("RBT",
                    compare_arbol_caso)
    dia_in, mes_in, anio_in= divir_fecha(inicio)
    dia_fin, mes_fin, anio_fin= divir_fecha(final)
    if anio_in==anio_fin:
        arb_meses=om.get(mapa, anio_in)
        arb_meses=me.getValue(arb_meses)
        if mes_in==mes_fin:
            mes= om.get(arb_meses, mes_in)
            mes= me.getValue(mes)
            agregar=om.values(mes, dia_in, dia_fin)
            arbol= agregar_arbol(agregar, arbol)

        else:
            meses= om.values(arb_meses, mes_in, mes_fin)
            first_mes=lt.firstElement(meses)
            last_mes=lt.lastElement(meses)
            meses2=meses
            for mes in lt.iterator(meses2):
                if mes == first_mes:
                    max = om.maxKey(mes)
                    agregar= om.values(mes, dia_in, max)
                    arbol= agregar_arbol(agregar, arbol)
                elif mes == last_mes:
                    min = om.minKey(mes)
                    agregar= om.values(mes, min, dia_fin)
                    arbol= agregar_arbol(agregar, arbol)
                else:
                    agregar= om.valueSet(mes)
                    arbol= agregar_arbol(agregar, arbol)

    else:
        anios=om.values(mapa, anio_in, anio_fin)
        first_anio=lt.firstElement(anios)
        last_anio=lt.lastElement(anios)
        for anio in lt.iterator(anios):
            if anio == first_anio:
                max = om.maxKey(anio)
                meses_min= om.values(anio, mes_in, max)
                first_mes=lt.firstElement(meses_min)
                last_mes=lt.lastElement(meses_min)
                meses2=meses_min
                for mes in lt.iterator(meses2):
                    if mes == first_mes:
                        max = om.maxKey(mes)
                        agregar= om.values(mes, dia_in, max)
                        arbol= agregar_arbol(agregar, arbol)
                    elif mes == last_mes:
                        min = om.minKey(mes)
                        agregar= om.values(mes, min, dia_fin)
                        arbol= agregar_arbol(agregar, arbol)
                    else:
                        agregar= om.valueSet(mes)
                        arbol= agregar_arbol(agregar, arbol)
            elif last_anio== anio:
                min = om.minKey(anio)
                meses_max= om.values(anio, min, mes_fin)
                first_mes=lt.firstElement(meses_max)
                last_mes=lt.lastElement(meses_max)
                meses2=meses_max
                for mes in lt.iterator(meses2):
                    if mes == first_mes:
                        max = om.maxKey(mes)
                        agregar= om.values(mes, dia_in, max)
                        arbol= agregar_arbol(agregar, arbol)
                    elif mes == last_mes:
                        min = om.minKey(mes)
                        agregar= om.values(mes, min, dia_fin)
                        arbol= agregar_arbol(agregar, arbol)
                    else:
                        agregar= om.valueSet(mes)
                        arbol= agregar_arbol(agregar, arbol)

            else:
                meses=om.valueSet(anio)
                for mes in lt.iterator(meses):
                    dias= om.valueSet(mes)
                    arbol= agregar_arbol(agregar, arbol)
    return arbol
```

```python
def req_1(data_structs, inicio, final):
    """
    Función que soluciona el requerimiento 1
    """
    # TODO: Realizar el requerimiento 1
    arbol= mp.get(data_structs, "fechas")
    arbol= me.getValue(arbol)
    RBT= RBT_rango_fechas(arbol, inicio, final)
    altura= om.height(RBT)
    nodos= om.keySet(RBT)
    nodos= lt.size(nodos)
    elementos= om.valueSet(RBT)
    elementos_size= lt.size(elementos)
    elementos= sort(elementos, 5)
    return elementos, altura, nodos, elementos_size
```

Descripción

Este requerimiento obtiene todos los accidentes en un rango de fecha, primero divide la fecha inicial y final en maño, mes y día, y obtiene el rango de esas fechas con una función llamada RBT_rango_fechas, que recibe las dos fechas y retorna un árbol con todos los valores de ese rango.

Entrada	Arbol de fechas, fecha inicial, fecha final
Salidas	Arbol RBT con el rango de fechas, la altura, los nodos y los elementos
Implementado (Sí/No)	Si

Análisis de complejidad

Pasos	**Complejidad**
Traer rango años	O(log n)
Traer rango mes fecha inicial	O(log n)
Traer rango mes fecha final	O(log n)
Crear árbol	O(n)
Traer rango días fecha inicial	O(log n)
Traer rango días fecha final	O(log n)
Recorrido para insertar en el árbol	O(n)
Total	O(n)

Pruebas Realizadas

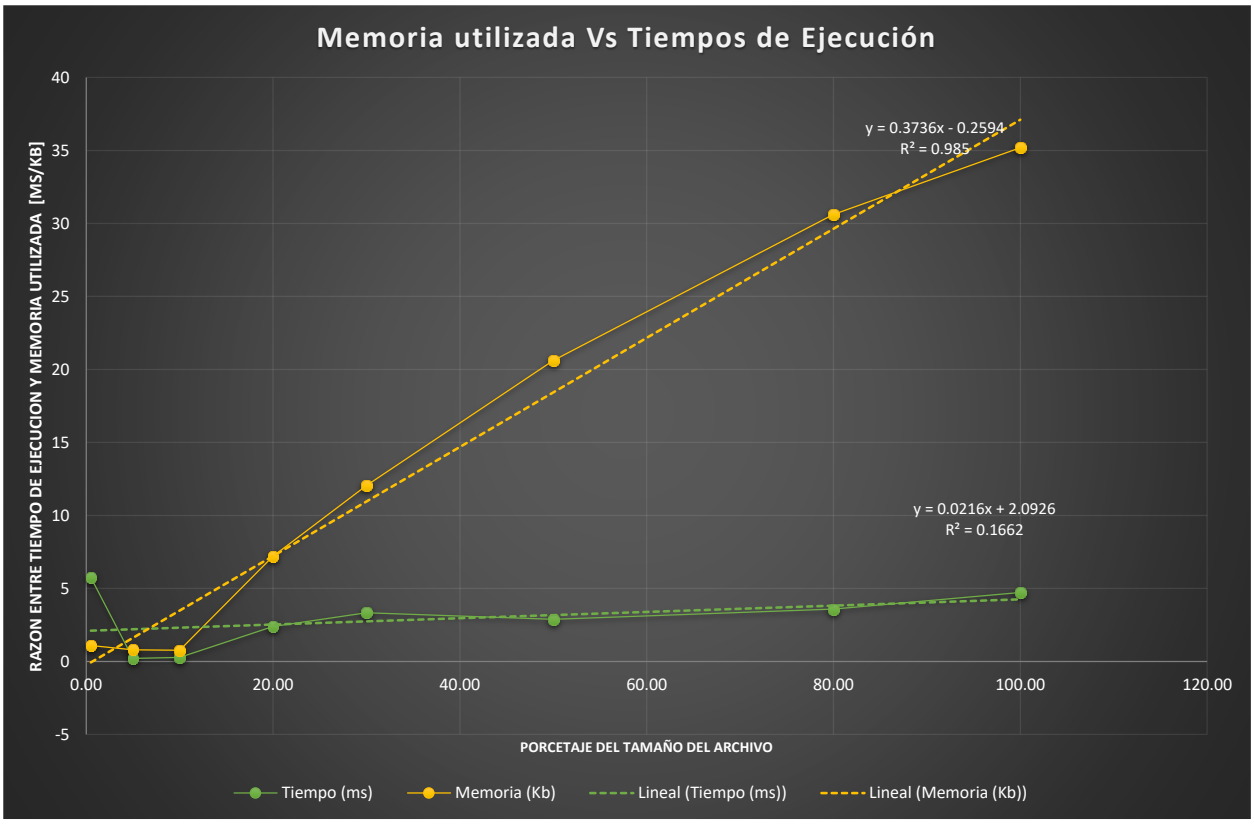
Las pruebas realizadas fueron realizadas en una maquina con las siguientes especificaciones. Los datos de entrada fueron el data structs del modelo.

Procesadores	**11th Gen Intel(R) Core(TM) i3-1115G4 @ 3.00GHz**
Memoria RAM	12 GB
Procesador	Core i3
Sistema Operativo	Windows 10

Entrada	**Tiempo (ms)**	**Memoria (Kb)**
small	5.738	1.085
5 pct	0.208	0.796
10 pct	0.268	0.765
20 pct	2.388	7.203
30 pct	3.334	12.04
50 pct	2.878	20.61
80 pct	3.591	30.61
large	4.721	35.18

Graficas

Las gráficas con la representación de las pruebas realizadas.



Análisis

Análisis de resultados de la implementación, tener cuenta las pruebas realizadas y el análisis de complejidad.

Según el cálculo de complejidad se obtiene que la complejidad en O(n) y analizando la gráfica, podemos ver que se acerca bastante a la linealidad teniendo pequeñas variaciones debido a procesos corriendo en segundo plano en el computador
En conclusión, si se ve una relación entre el cálculo teórico y el que se aprecia experimentalmente.

Requerimiento 2

```python
def req_2(data_structs, hora_inc, hora_fin, anio, mes):
    """
    Función que soluciona el requerimiento 2
    """
    arbol= mp.get(data_structs, "fechas")
    arbol= me.getValue(arbol)
    arbol= om.get(arbol, int(anio))
    arbol= me.getValue(arbol)
    arbol= om.get(arbol, mes)
    arbol=me.getValue(arbol)
    hora_inc= hora_inc.replace(":", "")
    hora_fin= hora_fin.replace(":", "")
    dias= om.valueSet(arbol)
    respuesta= om.newMap("RBT",
                    compare_arbol_caso)
    for dia in lt.iterator(dias):
        for data in lt.iterator(dia):
            hora=data["HORA_OCURRENCIA_ACC"].replace(":", "")
            if int(hora)>= int(hora_inc) and int(hora)<=int(hora_fin):
                om.put(respuesta, data["FORMULARIO"], data)

    return sort(om.valueSet(respuesta), 5)
```

Descripción

Este requerimiento obtiene los intervalos de horas para todos los días de un mes y un año en específico, primero obtiene los datos del año y el mes ingresado por parámetro, luego compara en cada lista de cada día, los accidentes que estén en este intervalo.

Entrada	Árbol de fechas, hora inicial, hora final
Salidas	Array list con todos los datos en este rango de horas
Implementado (Sí/No)	Si

Análisis de complejidad

Pasos	**Complejidad**
Traer año	O(log n)
Traer mes	O(log n)
Recorrer lista de cada día	O(n)
Agregar a la lista	O(n)
Total	O(n)

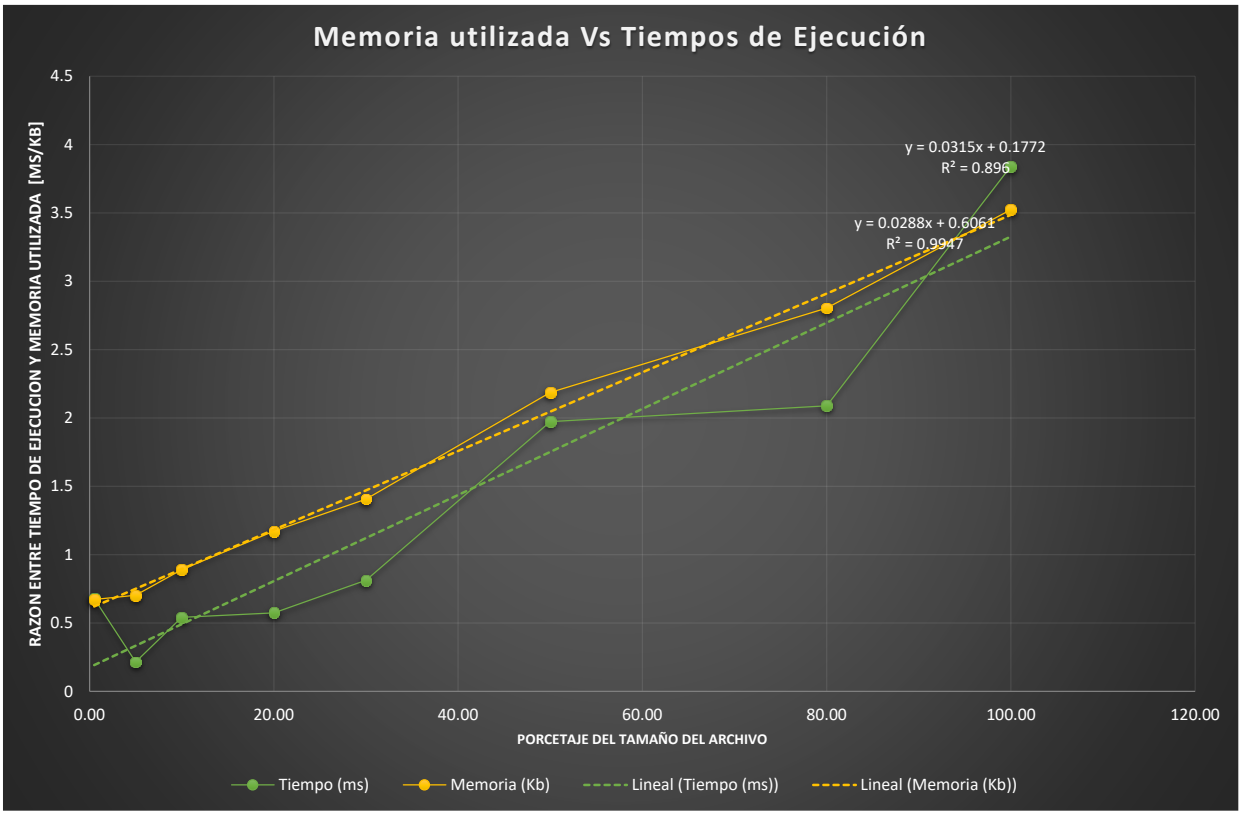
Pruebas Realizadas

Procesadores	11th Gen Intel(R) Core(TM) i3-1115G4 @ 3.00GHz
Memoria RAM	12 GB
Procesador	Core i3
Sistema Operativo	Windows 10

Entrada	Tiempo (ms)	Memoria (Kb)
small	0.678	0.671
5 pct	0.214	0.703
10 pct	0.541	0.890
20 pct	0.574	1.171
30 pct	0.813	1.406
50 pct	1.973	2.187
80 pct	2.089	2.804
large	3.840	3.523

Graficas

Las gráficas con la representación de las pruebas realizadas.



Análisis

Según el cálculo de complejidad se obtiene que la complejidad en O(n) y analizando la gráfica, podemos ver que se acerca bastante a la linealidad teniendo pequeñas variaciones debido a procesos corriendo en segundo plano en el computador
En conclusión, si se ve una relación entre el cálculo teórico y el que se aprecia experimentalmente.

Requerimiento 3

```python
def req_3(data_structs,clase_acc,Direccion):
    """
    Función que soluciona el requerimiento 3
    """
    # TODO: Realizar el requerimiento 3
    arbol_key_val= mp.get(data_structs, "casos")
    arbol_array= me.getValue(arbol_key_val)
    arbol_hash= om.get(arbol_array, clase_acc)
    registro= me.getValue(arbol_hash)


    respuesta= om.newMap("RBT",
                    compare_arbol_caso)

    for data in lt.iterator(registro):
        Direccion_data= data["DIRECCION"]
        if  Direccion.lower() in Direccion_data.lower():
            om.put(respuesta, data["FORMULARIO"], data)

    return sort(om.valueSet(respuesta), 5)
```

Descripción

Esta función busca el subsector económico que tuvo el menor total de retenciones para un año específico. Primero, obtiene los valores correspondientes a ese año del mapa, y los ordena según su subsector económico y sus retenciones tributarias, de menor a mayor. Esta lista se procesa a través de una función que divide los datos según la categoría ingresada por parámetro y los guarda en un mapa donde la clave es el subsector y el valor es un ArrayList con todos los datos correspondientes a ese subsector. Luego, estos datos se suman por subsector mediante otra función. Finalmente, se encuentra el subsector con el menor total de retenciones y se agregan los datos correspondientes a una lista.

Entrada	árbol de casos, tipo de caso, y la calle especifica
Salidas	Una Array List con todos los datos de esa calle y tipo de daño
Implementado (Sí/No)	Si, implementado por Franklin Romero

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Traer casos	O(log n)
Traer caso especifico	O(log n)
Recorrido de cada lista de cada registro	O(n)

Comparar direcciones “calles o vías etc”	O(n)
Agregar a la lista	O(n)
Ordenar por MergeSort	O(NLogn)
TOTAL	O(NLogn

Pruebas Realizadas

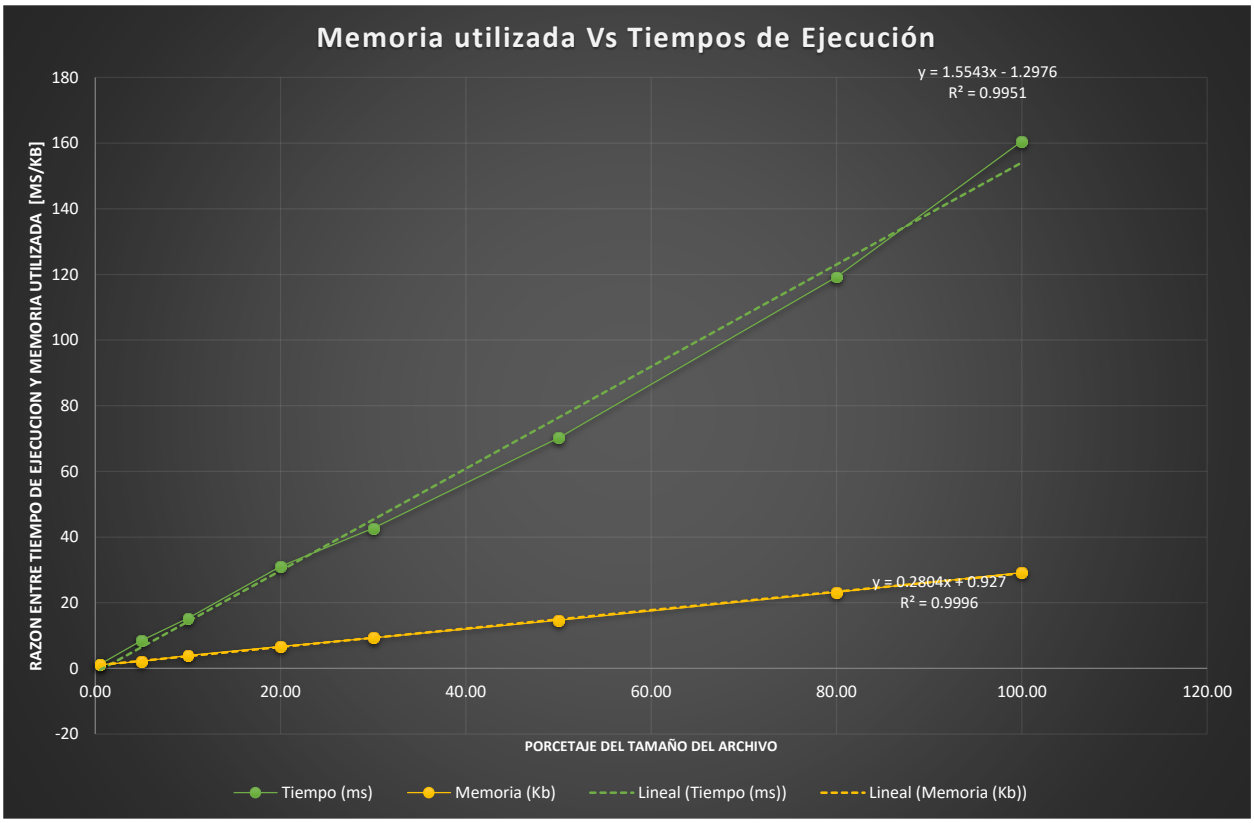
Descripción de las pruebas de tiempos de ejecución y memoria utilizada. Incluir descripción del procedimiento, las condiciones, las herramientas y recursos utilizados (librerías, computadores donde se ejecutan las pruebas, entre otros).

Procesadores	**11th Gen Intel(R) Core(TM) i3-1115G4 @ 3.00GHz**
Memoria RAM	12 GB
Procesador	Core i3
Sistema Operativo	Windows 10

Entrada	Tiempo (ms)	Memoria (Kb)
small	1.247	1.140
5 pct	8.523	2.140
10 pct	15.30	3.921
20 pct	31.04	6.703
30 pct	42.74	9.312
50 pct	70.22	14.65
80 pct	119.2	23.18
large	160.5	29.21

Graficas

Las gráficas con la representación de las pruebas realizadas.



Análisis

Análisis de resultados de la implementación, tener cuenta las pruebas realizadas y el analisis de complejidad.

Aunque la complejidad calculada fue de O(Nlogn), en la gráfica se puede evidenciar que el código tiende a ser este mismo, por tal motivo se cree que el código en los mejores casos puede ser O(Nlogn)

Requerimiento 4

```python
def req_4(data_structs, fecha_ini, fecha_fin, clase_acc):
    """
    Función que soluciona el requerimiento 4
    """
    # TODO: Realizar el requerimiento 4
    respuesta = lt.newList()
    arbol= mp.get(data_structs, "fechas")
    arbol= me.getValue(arbol)
    rbt = RBT_rango_fechas(arbol, fecha_ini, fecha_fin)
    rbt_size = om.size(RBT_rango_fechas_1(arbol, fecha_ini, fecha_fin))
    i = 0
    dia = me.getValue(mp.get(rbt, om.maxKey(rbt)))

    while i < 5:

        if lt.size(dia) == 0:
            om.deleteMax(rbt)
            if om.isEmpty(rbt):
                return respuesta, rbt_size
            dia = me.getValue(mp.get(rbt, om.maxKey(rbt)))
        min = lt.lastElement(dia)
        if min['GRAVEDAD'] == clase_acc:
            lt.addLast(respuesta, min)
            i+=1
        lt.removeLast(dia)
```

La función reporta los 5 accidentes más recientes dada una gravedad y un rango de fechas. En primer lugar, crea una lista vacía llamada "respuesta" que almacenará los 5 accidentes más recientes que cumplen con los criterios especificados. Después obtiene el árbol de fechas y crea un árbol rojo-negro que contiene las fechas de los accidentes que caen dentro del rango de fechas especificado, que se creó mediante la función RBT_rango_fechas. Inicializa un contador llamado "i" en 0 que se utilizará para contar los accidentes más recientes y entra en un bucle que se ejecuta mientras el contador "i" es menor que 5 (es decir, hasta que se hayan encontrado los 5 accidentes más recientes que cumplen con los criterios especificados). Si la lista de accidentes asociada al día más reciente está vacía, elimina el nodo con la clave máxima (es decir, la fecha más reciente) del árbol rojo-negro y obtiene la estructura de datos asociada a la nueva fecha máxima. Si el árbol rojo-negro está vacío después de la eliminación, la función devuelve la lista "respuesta" y el tamaño del árbol rojo-negro. Si la gravedad del accidente coincide con la gravedad especificada, agrega el accidente a la lista "respuesta" y aumenta el contador "i" en 1. Finalmente elimina el último elemento de la lista de accidentes asociada al día más reciente e itera hasta que se hayan encontrado los 5 accidentes más recientes que cumplen con los criterios especificados.

Descripción

Esta función retorna los 5 accidentes más recientes en un periodo de tiempo.

Entrada	La estructura de datos principal, fecha inicial, fecha final, gravedad del accidente
Salidas	Una lista con los 5 accidentes más recientes y la cantidad de todos lo accidentes en el rango
Implementado (Sí/No)	Si, implementado por Pablo Arango

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Crear una lista vacía	O(1)
Crear árbol rango	O(n)
Obtener el tamaño de un árbol rojo-negro	O(1)
Recorrer la lista	O(n)
Acceder y eliminar un nodo (en ciclo)	O(n logn)
TOTAL	***O(n logn)***

Pruebas Realizadas

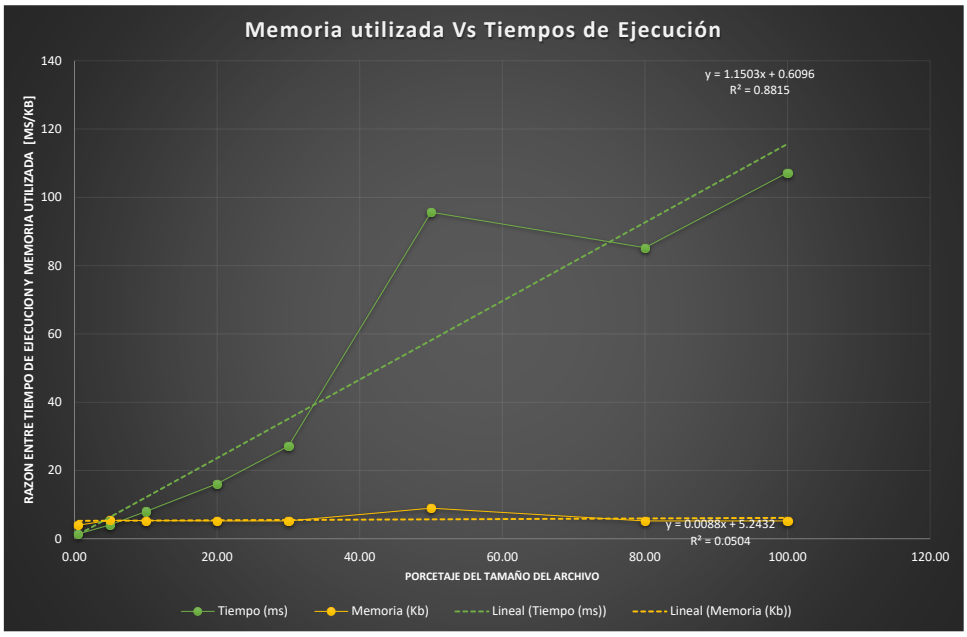
Las pruebas realizadas fueron realizadas en una maquina con las siguientes especificaciones. Los datos de entrada fueron el data structs del modelo.

Procesadores	**11th Gen Intel(R) Core(TM) i3-1115G4 @ 3.00GHz**
Memoria RAM	12 GB
Procesador	Core i3
Sistema Operativo	Windows 10

Entrada	**Tiempo (ms)**	**Memoria (Kb)**
small	1.392	3.992
5 pct	4.060	5.429
10 pct	7.972	5.335
20 pct	16.12	5.210
30 pct	27.14	5.210
50 pct	95.63	8.945
80 pct	85.24	5.210
large	107.2	5.210

Graficas

Las gráficas con la representación de las pruebas realizadas.



Análisis

Aunque la complejidad calculada fue de O(nlogn), en la gráfica se puede evidenciar que el código tiende a ser lineal, por tal motivo se cree que el código en los mejores casos puede ser O(n). Es evidente que el n log(n) no tiene este impacto en el caso promedio, pues en este caso solo deberá tomar en cuenta los datos de los primeros 5 y normalmente no se iteraran todas las fechas.

Requerimiento 5

```python
def req_5(data_structs, localidad, anio, mes):
    """
    Función que soluciona el requerimiento 5
    """
    # TODO: Realizar el requerimiento 5
    arbol= mp.get(data_structs, "fechas")
    arbol= me.getValue(arbol)
    arbol= om.get(arbol, int(anio))
    arbol= me.getValue(arbol)
    arbol= om.get(arbol, mes)
    arbol=me.getValue(arbol)
    respuesta= om.newMap("RBT",
                    compare_arbol_caso)
    dias= om.valueSet(arbol)
    for dia in lt.iterator(dias):
        for data in lt.iterator(dia):
            localidad_data=data["LOCALIDAD"]
            if localidad.lower()==localidad_data.lower():
                om.put(respuesta, data["FORMULARIO"], data)

    return sort(om.valueSet(respuesta), 5)
```

Descripción

Esta función obtiene los 10 accidentes menos recientes para una localidad, mes y año en específico, primero obtiene los datos del año y mes ingresado por parámetro, y luego hace un recorrido de cada día de ese mes y compara los que tengan la misma localidad y los agrega a una Array_list, luego obtiene los 10 últimos de la lista y los imprime.

Entrada	Arbol de fechas, localidad, mes y año
Salidas	Una Array List con todos los datos de esa localidad, mes y año
Implementado (Sí/No)	Si, implementado por Manuel Prieto

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Traer año	O(log n)
Traer mes	O(log n)
Recorrido de cada lista de cada día	O(n)
Comparar localidades	O(n)
Agregar a la lista	O(n)
Obtener los 10 últimos de la lista	O(1)
TOTAL	***O(n)***

Pruebas Realizadas

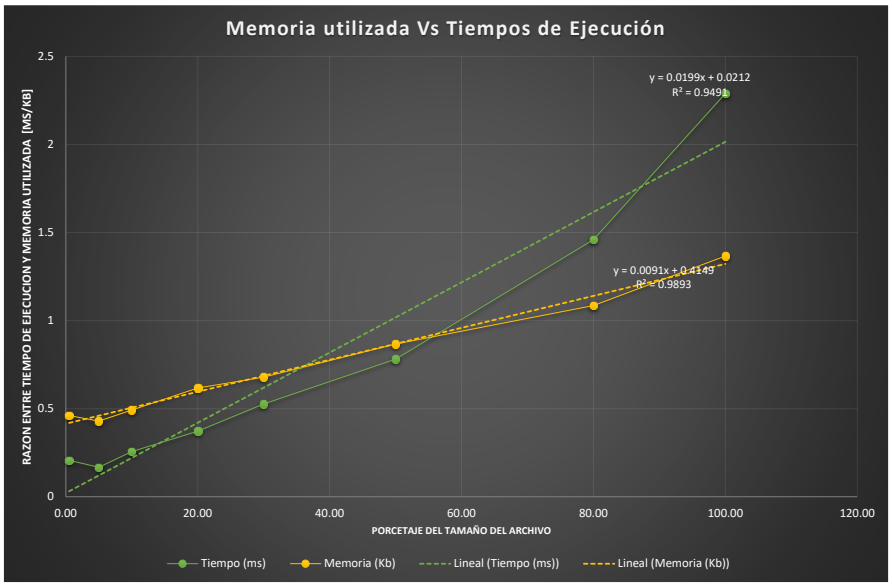
Las pruebas realizadas fueron realizadas en una maquina con las siguientes especificaciones. Los datos de entrada fueron el data structs del modelo.

Procesadores	**11th Gen Intel(R) Core(TM) i3-1115G4 @ 3.00GHz**
Memoria RAM	12 GB
Procesador	Core i3
Sistema Operativo	Windows 10

Entrada	**Tiempo (ms)**	**Memoria (Kb)**
small	0.207	0.460
5 pct	0.167	0.429
10 pct	0.256	0.492
20 pct	0.372	0.617
30 pct	0.527	0.679
50 pct	0.781	0.867
80 pct	1.459	1.085
large	2.288	1.367

Graficas

Las gráficas con la representación de las pruebas realizadas.



Análisis

Aunque la complejidad calculada fue de O(n), en la gráfica se puede evidenciar que el código tiende a ser lineal, por tal motivo se cree que el código en los mejores casos puede ser O(logn)

Requerimiento 6

```python
def req_6(data_structs, top, lat, lon, rad, anio, mes):
    """
    Función que soluciona el requerimiento 6
    """
    arbol= mp.get(data_structs, "fechas")
    arbol= me.getValue(arbol)
    arbol= om.get(arbol, int(anio))
    arbol= me.getValue(arbol)
    arbol= om.get(arbol, mes)
    arbol=me.getValue(arbol)
    respuesta= om.newMap("RBT",
                    compare_arbol_caso)
    dias= om.valueSet(arbol)
    for dia in lt.iterator(dias):
        for data in lt.iterator(dia):
            lat2=float(data["LATITUD"])
            lon2=float(data["LONGITUD"])
            distancia=calcular_haversine(lat,lon,lat2,lon2)
            if distancia<= float(rad):
                if int(om.size(respuesta))<int(top):
                    om.put(respuesta, distancia, data)
                else:
                    max= om.maxKey(respuesta)
                    if distancia<max:
                        om.deleteMax(respuesta)
                        om.put(respuesta,distancia, data)

    return om.valueSet(respuesta)
```

Descripción

Esta función obtiene un número de accidentes ocurridos en una zona, mes y año en específico, primero obtiene los datos del año y mes ingresado por parámetro, y luego hace un recorrido de cada día de ese mes y calcula su distancia a partir del punto central y los agrega a un mapa a retornar, primero agrega el número de valores correspondiente al ingresado por parámetro, y luego va obteniendo la mayor llave que es la distancia del punto central y compara hasta encontrar una menor y la intercambia.

Entrada	Árbol de fechas, numero de datos, latitud, longitud, distancia, mes, año
Salidas	Lista de los valores de un árbol con los datos
Implementado (Sí/No)	Si

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Traer año	O(log n)
Traer mes	O(log n)
Crear RBT	O(n)
Recorrido de cada lista de cada día	O(n)
Calcular distancia	O(n)
Comparar e ingresar al arbol	O(n)
TOTAL	***O(n)***

Pruebas Realizadas

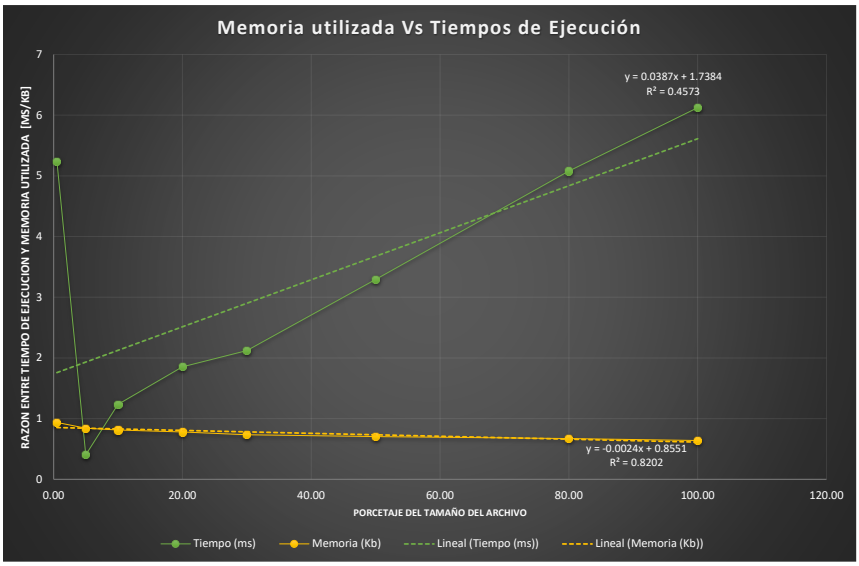
Las pruebas realizadas fueron realizadas en una maquina con las siguientes especificaciones. Los datos de entrada fueron el data structs del modelo.

Procesadores	**11th Gen Intel(R) Core(TM) i3-1115G4 @ 3.00GHz**
Memoria RAM	12 GB
Procesador	Core i3
Sistema Operativo	Windows 10

Entrada	**Tiempo (ms)**	**Memoria (Kb)**
small	5.241	0.937
5 pct	0.404	0.843
10 pct	1.236	0.812
20 pct	1.854	0.781
30 pct	2.122	0.734
50 pct	3.294	0.703
80 pct	5.079	0.671
large	6.122	0.640

Graficas

Las gráficas con la representación de las pruebas realizadas.



Análisis

La gráfica se comporta según lo calculado en la complejidad del código, ya que podemos ver que, con respecto a los tiempos, la gráfica tiende a ser nlogn, y lo calculado en la complejidad es O(logn)

Requerimiento 7

```python
def req_7(data_structs, anio, mes):
    """
    Función que soluciona el requerimiento 7
    """
    # TODO: Realizar el requerimiento 7
    arbol= mp.get(data_structs, "fechas")
    arbol= me.getValue(arbol)
    arbol= om.get(arbol, int(anio))
    arbol= me.getValue(arbol)
    arbol= om.get(arbol, mes)
    arbol=me.getValue(arbol)
    lista=lt.newList(datastructure="ARRAY_LIST")
    queue= qu.newQueue()
    lista_hora=lt.newList(datastructure="ARRAY_LIST")

    dias= om.valueSet(arbol)
    for dia in lt.iterator(dias):
        dia= sort(dia, 3)
        first=lt.firstElement(dia)
        dia_invertido=sort(dia, 4)
        last= lt.firstElement(dia_invertido)
        lt.addLast(lista, first)
        lt.addLast(lista, last)
        lista= sort(lista, 2)
        qu.enqueue(queue, lista)
        lista=lt.newList(datastructure="ARRAY_LIST")
        for data in lt.iterator(dia):
            hora=sacar_hora(data)
            lt.addLast(lista_hora, hora)

    lista_hora= sort(lista_hora, 6)

    return queue, lista_hora
```

Descripción

Esta función reporta los accidentes más tempranos y más tardes de cada día y grafica un histograma con los datos, primero obtiene los datos del año y mes ingresado por parámetro, y luego hace un recorrido de cada día de ese mes, crea una cola donde se va a guardan las listas con los accidentes ocurridos más temprano y tarde de cada día, y también una lista con todas las veces que se repite una hora, para después en la vista graficarlos

Entrada	El mapa del modelo, el top, año y subsector
Salidas	Una lista con el top
Implementado (Sí/No)	Si

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Traer año	O(log n)
Traer mes	O(log n)
Recorrido de cada lista de cada día	O(n)
Agregar a la lista	O(n)
Ordenar los datos	O(nlogn)
Agregar a la cola	O(31)
Agregar a la lista_horas	O(n)
TOTAL	***O(nlogn)***

Pruebas Realizadas

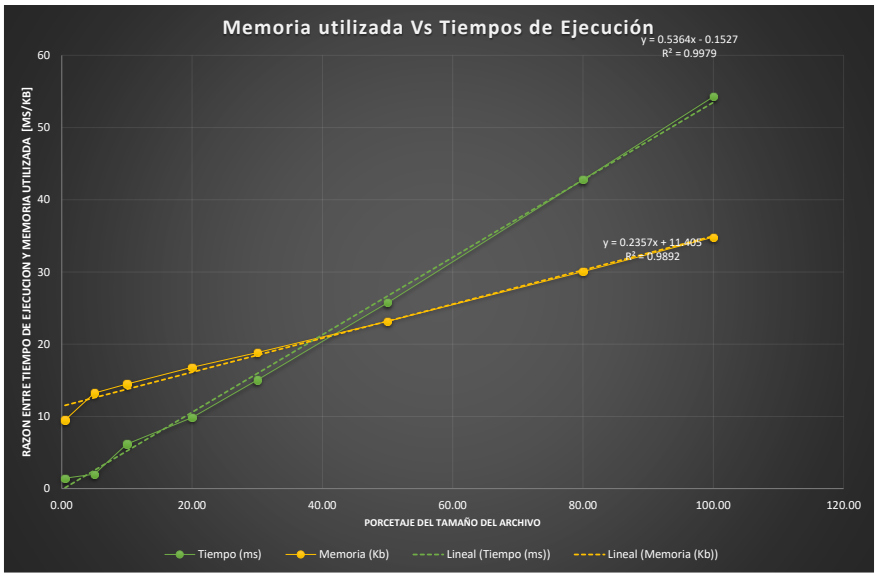
Las pruebas realizadas fueron realizadas en una maquina con las siguientes especificaciones. Los datos de entrada fueron el data structs del modelo.

Procesadores	**11th Gen Intel(R) Core(TM) i3-1115G4 @ 3.00GHz**
Memoria RAM	12 GB
Procesador	Core i3
Sistema Operativo	Windows 10

Entrada	**Tiempo (ms)**	**Memoria (Kb)**
small	1.438	9.5
5 pct	1.996	13.24
10 pct	6.168	14.49
20 pct	9.842	16.78
30 pct	15.04	18.83
50 pct	25.74	23.16
80 pct	42.78	30.04
large	54.26	34.82

Graficas

Las gráficas con la representación de las pruebas realizadas.



Análisis

A pesar de que la complejidad calculada fue de O(n), en la gráfica se puede evidenciar que la función tiende a ser nlogn , lo que da por conclusión que la función en casos promedios suele ser nlogn.

Requerimiento 8

```python
def req_8(data_structs, inicio, final, clase):
    """
    Función que soluciona el requerimiento 8
    """
    # TODO: Realizar el requerimiento 8
    color = lt.newList(datastructure="ARRAY_LIST")
    lista = lt.newList(datastructure="ARRAY_LIST")
    arbol= mp.get(data_structs, "fechas")
    arbol= me.getValue(arbol)
    rbt= RBT_rango_fechas_1(arbol, inicio, final)
    for i in lt.iterator(om.valueSet(rbt)):
        if i["CLASE_ACC"]==clase:
            lt.addLast(lista, i)
            if i["GRAVEDAD"] == "CON MUERTOS":
                lt.addLast(color, "red")
            elif i["GRAVEDAD"] == "CON HERIDOS":
                lt.addLast(color, "orange")
            else:
                lt.addLast(color, "green")
    return lista, color
```

Descripción

La función se encarga de visualizar todos los accidentes de una clase particular para un rango de fechas en el mapa de Bogotá. Primero, la función crea dos listas vacías, color y lista, que se utilizarán más adelante para almacenar los accidentes y los colores correspondientes. Luego, se obtiene un árbol de búsqueda a partir de la estructura de datos que se le pasó como parámetro, y se utiliza una función para obtener un árbol rojo-negro con elementos en el rango de fechas especificado. Se itera sobre los elementos del árbol rojo-negro y se verifica si el accidente cumple con la clase. Si el accidente cumple

con estas condiciones, se agrega a la lista, y se agrega el color correspondiente a la otra lista. El color del accidente se determina según su gravedad: "CON MUERTOS" se representa con el color rojo, "CON HERIDOS" con naranja, y los accidentes con "SOLO DAÑOS" con verde. Finalmente, la función devuelve las dos listas que contienen los accidentes que cumplen con las condiciones especificadas y sus colores correspondientes.

Entrada	La estructura de datos principal, fecha inicial, fecha final, gravedad del accidente
Salidas	Lista de accidentes y lista de colores
Implementado (Sí/No)	Si.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Crear dos listas vacías	O(1)
Crear un árbol rojo-negro a partir de un rango de fechas	O(n)
Iniciar ciclo	O(n)
Acceder a un elemento del arbol (En ciclo)	O(n logn)
TOTAL	***O(n logn)***

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).

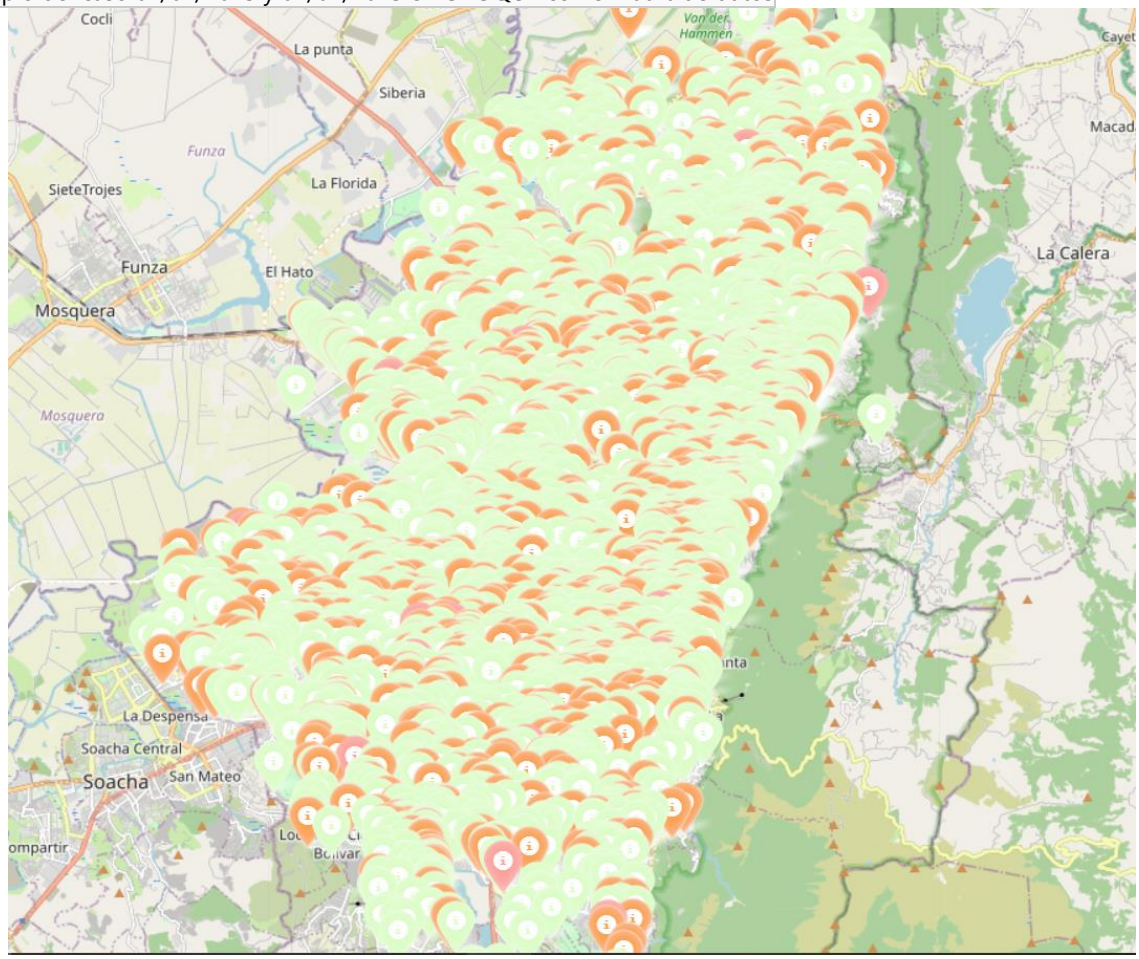
Procesadores	11th Gen Intel(R) Core(TM) i3-1115G4 @ 3.00GHz
Memoria RAM	12 GB
Procesador	Core i3
Sistema Operativo	Windows 10

Entrada	**Tiempo (ms)**	**Memoria (Kb)**
small	0.637	3.007
5 pct	3.393	10.67
10 pct	11.52	13.85
20 pct	20.00	19.48
30 pct	20.48	27.04
50 pct	30.96	39.16
80 pct	66.62	58.35
large	81.50	75.36

Salida (con el 100% de datos)

Las tablas con la recopilación de datos de las pruebas.

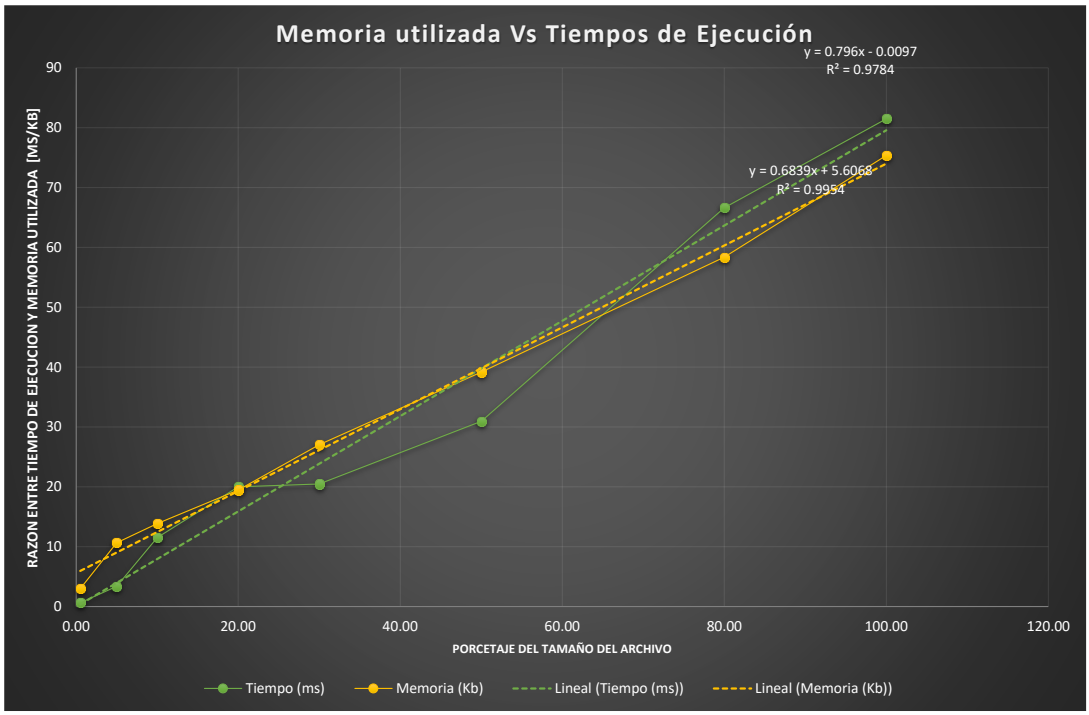
Ejemplo del caso 02/02/2015 y 02/04/2015 en CHOQUE con el 100% de datos



PD: que montón de accidentes en solo 2 meses

Graficas

Las gráficas con la representación de las pruebas realizadas.



Análisis

El algoritmo se acerca a un tiempo de ejecución n log n y se beneficia del uso de rangos pequeños.