ANÁLISIS DEL RETO

Juan Andrés Vargas Bolaños, 202110398, ja.vargasb1@uniandes.edu.co

Juan Camilo Lyons Bustamante, 201913100, jc.lyons@uniandes.edu.co

Jorge Eduardo Solórzano Díaz, 202115798, j.solorzanod@uniandes.edu.co

Carga de datos

Descripción

```
def new_data_structs():
    """
    Inicializa las estructuras de datos del modelo. Las crea de
    manera vacía para posteriormente almacenar la información.
    """
    #TODO: Inicializar las estructuras de datos
    data_structs = {"list_accidents":None,
                    "dates_index": None,
                    "acc_class_av":None,
                    }

    data_structs["list_accidents"] = lt.newList("ARRAY_LIST")
    data_structs["dates_index"] = om.newMap(omaptype="RBT",comparefunction=compareDates)
    data_structs["acc_class_av"] = om.newMap(omaptype="RBT",comparefunction=compare)

    return data_structs
```

```
def add_data(data_structs, accident):
    """
    Función para agregar nuevos elementos a la lista
    """
    #TODO: Crear la función para agregar elementos a una lista
    lt.addLast(data_structs["list_accidents"],accident)
    updateDateIndex(data_structs["dates_index"], accident)
    updateClassAvIndex(data_structs["acc_class_av"], accident)
    return data_structs
```

```python
def updateClassAvIndex(map, accident):
    """
    Crea una nueva estructura para modelar los datos
    """
    #TODO: Crear la función para estructurar los datos
    if "AV" in accident["DIRECCION"]:
        acc_class_av = (accident["DIRECCION"].split("-"))[0],accident["CLASE_ACC"]
        entry = om.get(map, acc_class_av)
        if entry is None:
            dataentry = newClassAvEntry(accident)
            lst = me.getValue(dataentry)
            om.put(map, acc_class_av, lst)
        else:
            dataentry = entry
        addClassAvIndex(dataentry, accident)
        return map

def newClassAvEntry(accident):
    entry = {"ClassAvIndex": None, "value": None}
    entry["ClassAvIndex"] = (accident["DIRECCION"].split("-"))[0],accident["CLASE_ACC"]
    entry["value"] = lt.newList("ARRAY_LIST")
    return entry

def addClassAvIndex(dataentry, accident):
    lst = me.getValue(dataentry)
    lt.addLast(lst, accident)
    return dataentry
```

```python
def updateDateIndex(map, accident):
    """
    Crea una nueva estructura para modelar los datos
    """
    #TODO: Crear la función para estructurar los datos
    occurreddate = accident["FECHA_OCURRENCIA_ACC"]
    entry = om.get(map, occurreddate)
    if entry is None:
        dataentry = newDateEntry(accident)
        lst = me.getValue(dataentry)
        om.put(map, occurreddate, lst)
    else:
        dataentry = entry
    addDateIndex(dataentry, accident)
    return map

def newDateEntry(accident):
    entry = {"dateIndex": None, "value": None}
    entry["dateIndex"] = accident["FECHA_OCURRENCIA_ACC"]
    entry["value"] = lt.newList("ARRAY_LIST")
    return entry

def addDateIndex(dataentry, accident):
    lst = me.getValue(dataentry)
    lt.addLast(lst, accident)
    return dataentry
```

Entrada	Archivo csv con toda la información de los accidentes.
Salidas	Diccionario con tres llaves.
Implementado (Sí/No)	Si. Implementado por Juan Andrés Vargas B.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Creación de la lista	O(N)
Creación del mapa tipo RBT (x2)	O(Log N)
TOTAL	***O(N)***

Pruebas Realizadas

AMD Ryzen 5 4600H with Radeon Graphics
3.00GHz

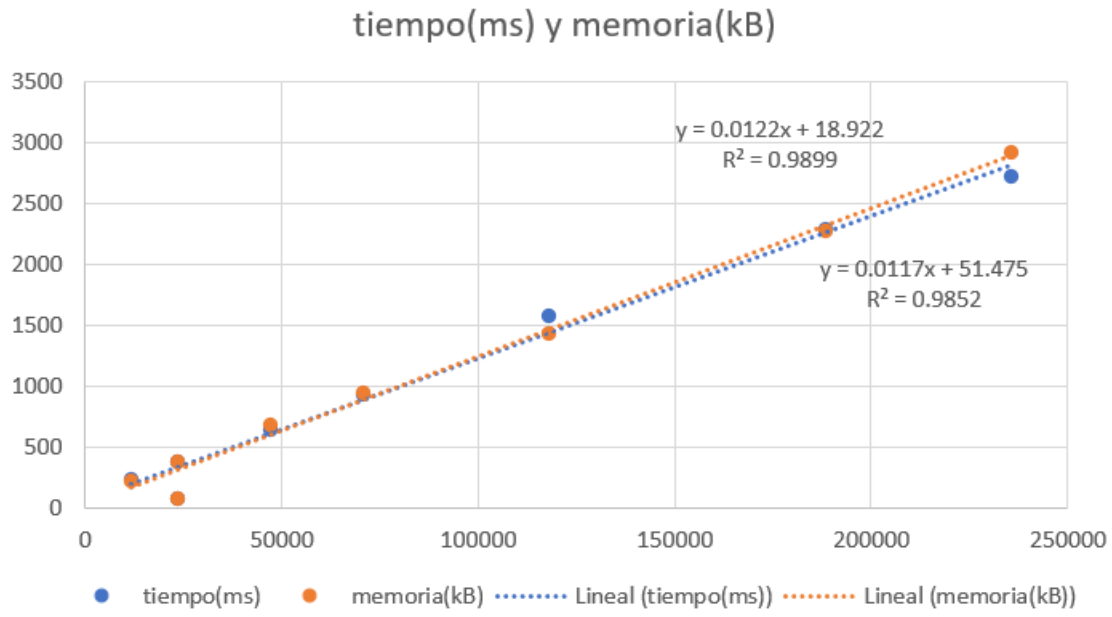
8,00 GB (7,36 GB utilizable)

Windows 11 Sistema operativo de 64 bits, procesador x64

Entrada	Tiempo (ms)	Memoria(kB)
small	76.91	79.14
5 pct	238.62	221.82
10 pct	371.82	378.86
20 pct	636.94	681.18
30 pct	935.69	938.38
50 pct	1576.17	1426.79
80 pct	2287.99	2280.49
large	2721.59	2925.59

Graficas

Las gráficas con la representación de las pruebas realizadas.



Análisis

Observamos de la gráfica que como se concluyó del análisis de complejidad. Esta tiene una complejidad lineal. Además, vemos que la memoria también crece linealmente. Esto tiene sentido con lo implementado ya que en todos los casos es necesario recorrer todos los datos para así cargarlos en la estructura de datos.

Aunque en la implementación haya un orden de “Log N”, para la complejidad de implementación completa se toma el de mayor orden por lo que se confirma que es lineal.

Requerimiento 1

Descripción

```python
def req_1(data_structs, fecha_i, fecha_f):
    """
    Función que soluciona el requerimiento 1
    """
    # TODO: Realizar el requerimiento 1
    values = om.values(data_structs['dates_index'], fecha_i, fecha_f)
    list_sis = lt.newList("ARRAY_LIST")
    for value in lt.iterator(values):
        for val in lt.iterator(value):
            lt.addFirst(list_sis, val)

    param = lt.newList("ARRAY_LIST")
    lt.addLast(param, 'CODIGO_ACCIDENTE')
    lt.addLast(param, 'DIA_OCURRENCIA_ACC')
    lt.addLast(param, 'DIRECCION')
    lt.addLast(param, 'GRAVEDAD')
    lt.addLast(param, 'CLASE_ACC')
    lt.addLast(param, 'LOCALIDAD')
    lt.addLast(param, 'FECHA_HORA_ACC')
    lt.addLast(param, 'LATITUD')
    lt.addLast(param, 'LONGITUD')

    final_list = lt.newList("ARRAY_LIST")

    for siniestro in lt.iterator(list_sis):
        aux_list = []
        for par in lt.iterator(param):
            aux_list.append(siniestro[par])
        lt.addLast(final_list, aux_list)

    f_list = merg.sort(final_list, compareDatesandhour)

    return param, f_list
```

Entrada	Recibe el data structure junto con un rango inicial-final de una fecha.
Salidas	Retorna todos los accidentes ocurridos en el rango dado.
Implementado (Sí/No)	Si, implementado por Juan Camilo Lyos y Jorge Solórzano

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**

Sacar los valores en el rango dado	O(LogN)
Crear la lista de parámetros e insertar	O(1)
Iterar la lista para sacar los parámetros	O(N)
Ordenar la lista	O(NlogN)
TOTAL	***O(NlogN)***

Pruebas Realizadas

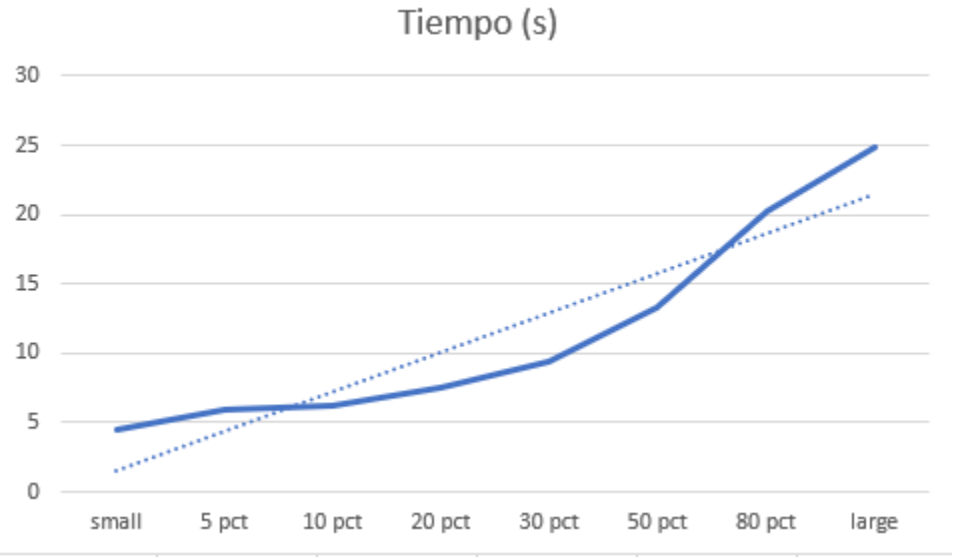
Procesadores	**AMD Ryzen 5 3400G with Radeon Vega Graphics 3.70 GHz**
Memoria RAM	16 GB
Sistema Operativo	Windows 11

Entrada	**Tiempo (s)**
small	4,422
5 pct	5,891
10 pct	6,275
20 pct	7,591
30 pct	9,394
50 pct	13,337
80 pct	20,295
large	24,877

Tablas de datos

Muestra	**Salida**	**Tiempo (ms)**
small	Accidentes Entre 01/11/2016 y 08/11/2016	4,422
5 pct	Accidentes Entre 01/11/2016 y 08/11/2016	5,891
10 pct	Accidentes Entre 01/11/2016 y 08/11/2016	6,275
20 pct	Accidentes Entre 01/11/2016 y 08/11/2016	7,591
30 pct	Accidentes Entre 01/11/2016 y 08/11/2016	9,394
50 pct	Accidentes Entre 01/11/2016 y 08/11/2016	13,337
80 pct	Accidentes Entre 01/11/2016 y 08/11/2016	20,295
large	Accidentes Entre 01/11/2016 y 08/11/2016	24,877

Graficas



Análisis

Teniendo en cuenta los resultados y la complejidad, depende de que tan grande sea el rango de fechas y el tamaño del archivo varía el tiempo, pero siempre se mantendrá por debajo de NlogN por cómo está hecho el algoritmo. Además, el espacio en memoria trata de ser el mínimo posible al ejecutar el req.

Requerimiento 2

Descripción

```python
def req_2(data_structs,horamin_initial,horamin_final,anio,mes):
    """
    Función que soluciona el requerimiento 2
    """
    # TODO: Realizar el requerimiento 2
    mes = MonthNameToNumber(mes)

    horamin_initial = horamin_initial + ":00"
    horamin_final = horamin_final + ":00"

    horamin_initial = horamin_initial.split(":")
    horamin_final = horamin_final.split(":")
    for i in range(0,3):
        horamin_initial[i] = int(horamin_initial[i])
        horamin_final[i] = int(horamin_final[i])


    rbt = data_structs["dates_index"]
    fecha_initial = "{0}/{1}/01".format(anio,mes)
    fecha_final = "{0}/{1}/31".format(anio,mes)

    values = om.values(rbt,fecha_initial,fecha_final)
    array = lt.newList("ARRAY_LIST")
    for element in lt.iterator(values):
        for accident in lt.iterator(element):
            lt.addLast(array,accident)
```

```
ranged_array = lt.newList("ARRAY_LIST")
for accident in lt.iterator(array):
    horamin = accident["HORA_OCURRENCIA_ACC"]

    horamin = horamin.split(":")
    for i in range(0,3):
        horamin[i] = int(horamin[i])

    if (horamin >= horamin_initial) and (horamin <= horamin_final):
        lt.addLast(ranged_array,accident)

ranged_sorted_array = merg.sort(ranged_array,compareHours)

param = ["CODIGO_ACCIDENTE","HORA_OCURRENCIA_ACC","FECHA_OCURRENCIA_ACC",
         "DIA_OCURRENCIA_ACC","LOCALIDAD","DIRECCION","GRAVEDAD","CLASE_ACC",
         "LATITUD","LONGITUD"]

lista_tabulate = lt.newList("ARRAY_LIST")
lt.addLast(lista_tabulate,param)

for accident in lt.iterator(ranged_sorted_array):
    lista_aux = lt.newList("ARRAY_LIST")
    for x in param:
        lt.addLast(lista_aux,accident[x])
    lt.addLast(lista_tabulate,lista_aux["elements"])

return lista_tabulate["elements"]
```

```
def compareHours(acc1,acc2):
    hour1 = acc1["HORA_OCURRENCIA_ACC"].split(":")
    hour2 = acc2["HORA_OCURRENCIA_ACC"].split(":")

    for i in range(0,3):
        hour1[i] = int(hour1[i])
        hour2[i] = int(hour2[i])

    if (hour1 == hour2):
        return acc1["FECHA_OCURRENCIA_ACC"] < acc2["FECHA_OCURRENCIA_ACC"]
    else:
        return hour1 < hour2
```

```
def MonthNameToNumber(mes):
    if not mes.isdecimal():
        mes = mes.upper()
        if mes == "ENERO":
            mes = "01"
        elif mes == "FEBRERO":
            mes = "02"
        elif mes == "MARZO":
            mes = "03"
        elif mes == "ABRIL":
            mes = "04"
        elif mes == "MAYO":
            mes = "05"
        elif mes == "JUNIO":
            mes = "06"
        elif mes == "JULIO":
            mes = "07"
        elif mes == "AGOSTO":
            mes = "08"
        elif mes == "SEPTIEMBRE":
            mes = "09"
        elif mes == "OCTUBRE":
            mes = "10"
        elif mes == "NOVIEMBRE":
            mes = "11"
        elif mes == "DICIEMBRE":
            mes = "12"
    if len(mes) == 1:
        mes = "0"+mes

    return mes
```

En este requerimiento se busca reportar todos los accidentes en un intervalo de horas del día para un mes y año dados. Primero se busca ajustar los valores de entrada para poder utilizarlos para buscar en la estructura de datos. Pasamos la hora de hh/mm/ss a una lista para poder compararlo posteriormente.

Se obtienen de la estructura de datos el árbol RBT y se obtienen en una lista enlazada todas las llaves que se encuentren en el año y mes especificado. Posteriormente, se convierte la lista enlazada en un arreglo para simplificar la implementación. Anteriormente solo habíamos creado el rango por año y mes, ahora los vamos a definir para un rango de horas y minutos del tipo hh/mm/ss. De aquí obtenemos una lista con todos los accidentes que ocurrieron en este rango de horas especifico. Procedemos a sortear este arreglo desde el más reciente al más antiguo.

Ya con la lista sorteada, extraemos toda la información que queremos mostrar y diseñamos una lista de listas compatible con la función tabulate.

Entrada	Estructura de datos Hora y minutos iniciales del intervalo de tiempo Hora y minutos finales del intervalo de tiempo. Año de consulta Mes de consulta.
Salidas	Lista de listas diseñado para mostrarlos con tabulate con todos los accidentes ocurridos en el periodo de tiempo especificado.
Implementado (Sí/No)	Si. Implementado por Juan Andrés Vargas B.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Búsqueda en el RBT	O(2*Log N)
Creación del arreglo	O(M)
Iteración de arreglo	O(M)
Comparaciones	O(M)
Sorteamiento del arreglo	O(R Log R) con R<<M
TOTAL	***O(M)***

Pruebas Realizadas

AMD Ryzen 5 4600H with Radeon Graphics
3.00GHz
8,00 GB (7,36 GB utilizable)
Windows 11 Sistema operativo de 64 bits, procesador x64

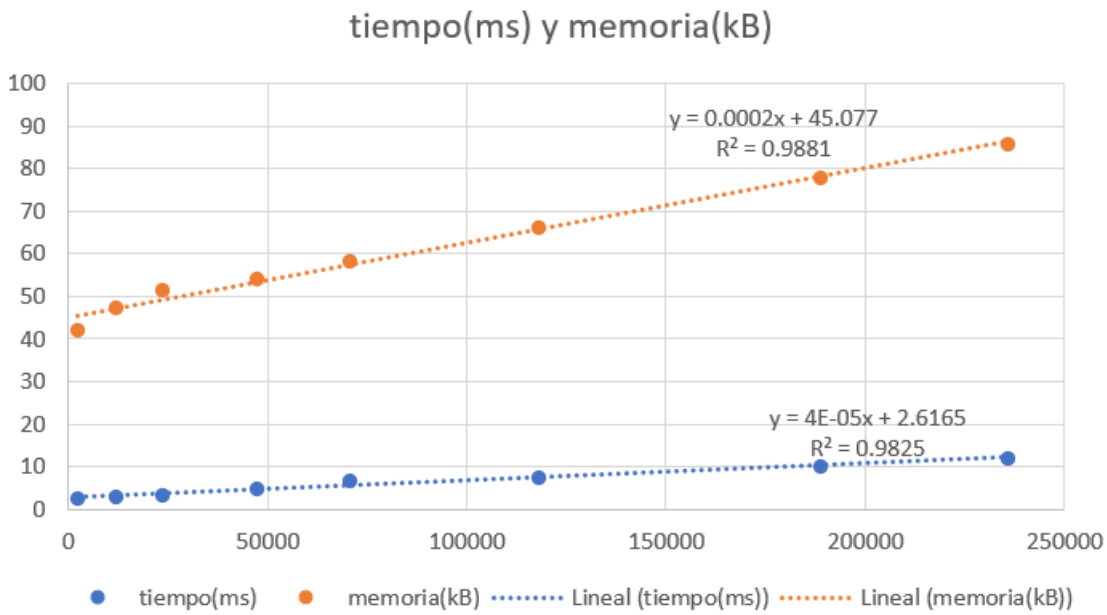
Entrada	**Tiempo (ms)**	**Memoria(kB)**
small	2.39	42.00
5 pct	2.89	47.28
10 pct	3.09	51.58
20 pct	4.87	54.18
30 pct	6.47	58.10
50 pct	7.51	66.04
80 pct	10.06	78.01
large	12.05	85.88

Tablas de datos

Las tablas con la recopilación de datos de las pruebas.

Graficas

Las gráficas con la representación de las pruebas realizadas.



Análisis

Observamos que la que el orden de complejidad en los tiempos es de orden lineal como se afirmó anteriormente. Observamos también que la complejidad en el uso en memoria también es lineal. Esto tiene sentido con la implementación porque en todos los casos se crea un arreglo y es necesario recorrer todo el arreglo. Aunque la búsqueda en el RBT y el sorteamiento sean de orden logarítmico ambos, esto no afecta la complejidad general de la implementación. Más específicamente en el sorteamiento no afecta ya que se hace sobre un arreglo de mucho menor tamaño que el arreglo original.

Requerimiento 3

Descripción

```
def req_3(data_structs,clase_accidente,nombre_via):
    """
    Función que soluciona el requerimiento 3
    """
    # TODO: Realizar el requerimiento 3

    acc_class_av = data_structs["acc_class_av"]

    nombre_via = nombre_via.upper()
    clase_accidente = clase_accidente.upper()

    if "AV " in nombre_via:
        llave = om.get(acc_class_av,(nombre_via,clase_accidente))
    else:
        nombre_via = "AV " + str(nombre_via)
        llave = om.get(acc_class_av,(nombre_via,clase_accidente))

    lista_acc = llave["value"]
    merg.sort(lista_acc,compareDatesreq3)
    sublista = lt.subList(lista_acc,1,3)
    param = ["CODIGO_ACCIDENTE","FECHA_HORA_ACC","DIA_OCURRENCIA_ACC",
             "LOCALIDAD","DIRECCION","GRAVEDAD","CLASE_ACC",
             "LATITUD","LONGITUD"]

    lista_tabulate = lt.newList("ARRAY_LIST")
    lt.addLast(lista_tabulate,param)

    for accident in lt.iterator(sublista):
        lista_aux = lt.newList("ARRAY_LIST")
        for x in param:
            lt.addLast(lista_aux,accident[x])
        lt.addLast(lista_tabulate,lista_aux["elements"])

    return lt.size(lista_acc),lista_tabulate["elements"]
```

```python
def compareDatesreq3(acc1,acc2):
    date1 = acc1["FECHA_OCURRENCIA_ACC"]
    date2 = acc2["FECHA_OCURRENCIA_ACC"]

    if date1 == date2:
        hour1 = acc1["HORA_OCURRENCIA_ACC"].split(":")
        hour2 = acc2["HORA_OCURRENCIA_ACC"].split(":")
        for i in range(0,3):
            hour1[i] = int(hour1[i])
            hour2[i] = int(hour2[i])
        return hour1 > hour2
    else:
        return date1 > date2
```

En este requerimiento nos piden reportar los 3 accidentes más recientes de una clase particular ocurridos a lo largo de una vía. Para esto utilizamos el mapa creado en la carga de datos donde las llaves son una tupla con la clase de accidente y el nombre de la avenida donde ocurrió el accidente.

Primero ajustamos los valores de entrada con el formato presente en la estructura de datos. Después usamos la función get para obtener la pareja llave-valor cuya llave sea la misma que la ingresada por parámetros. Obtenemos la lista con todos los accidentes y la sorteamos del más reciente al más antiguo.

Creamos una sublista con los 3 accidentes más recientes. Extraemos toda la información que queremos mostrar y diseñamos una lista de listas compatible con la función tabulate

Entrada	Estructura de datos, Clase del accidente, Nombre de la vía de la ciudad.
Salidas	Lista de listas diseñado para mostrarlos con tabulate con los 3 accidentes de la clase de accidente especificado, más recientes ocurridos en la vía especificada.
Implementado (Sí/No)	Si. Implementado por Juan Andrés Vargas B.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Búsqueda en el RBT	O(Log N)
Sorteo con mergesort de la lista	O(N*Log N)
TOTAL	***O(N*Log N)***

Pruebas Realizadas

AMD Ryzen 5 4600H with Radeon Graphics
3.00GHz

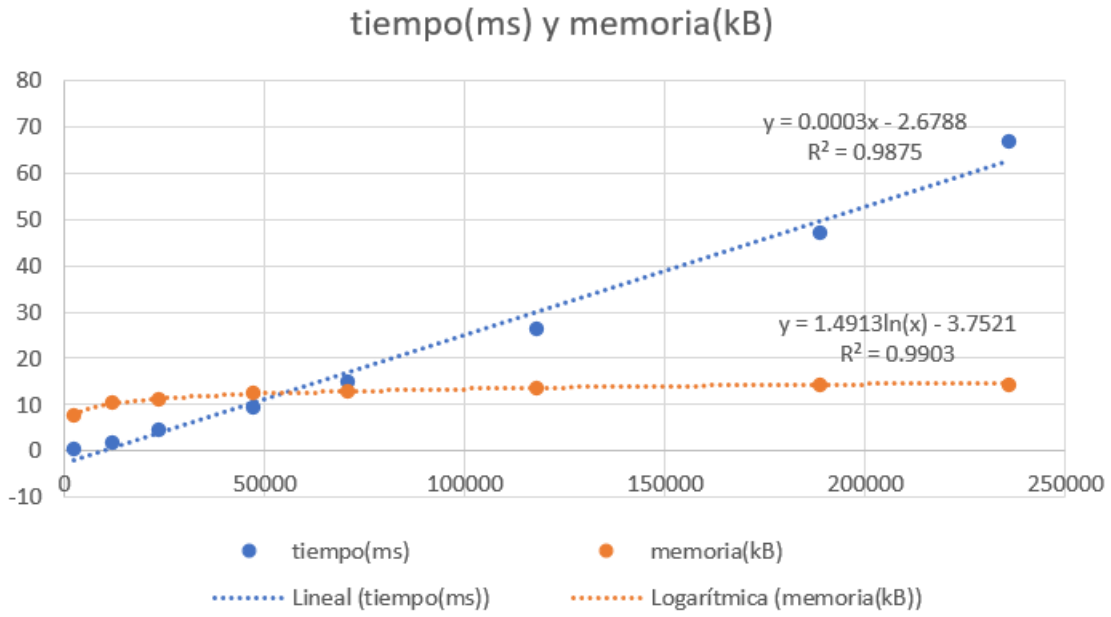
8,00 GB (7,36 GB utilizable)

Windows 11 Sistema operativo de 64 bits, procesador x64

Entrada	Tiempo (ms)	Memoria(kB)
small	0.57	7.63
5 pct	1.99	10.51
10 pct	4.48	11.02
20 pct	9.39	12.55
30 pct	14.83	13.05
50 pct	26.57	13.72
80 pct	47.19	14.38
large	66.98	14.38

Graficas

Las gráficas con la representación de las pruebas realizadas.



Análisis

Aunque en el análisis de complejidad se afirmó que el orden es linearitmico, también se puede aproximar se cómo si tuviera complejidad lineal ya que una función linearitmica combina tanto un crecimiento lineal como un crecimiento logarítmico. La función debería mostrar un crecimiento más lento a medida que el valor de x aumenta, debido a la influencia logarítmica. Por otro lado, observamos que el orden de complejidad del uso en memoria parece ser de orden logarítmico y esto se puede deber a que la complejidad de uso de memoria de un árbol rojo-negro depende del número de nodos en el árbol, y la altura del árbol está limitada por una función logarítmica en el número de nodos. Por lo tanto,

la complejidad de uso de memoria de un árbol rojo-negro puede ser de orden logarítmico en el peor de los casos.

Requerimiento 4

Descripción

```
def req_4(data_structs, fecha_i, fecha_f, g_accidente):
    """
    Función que soluciona el requerimiento 4
    """
    # TODO: Realizar el requerimiento 4
    list_rank = om.values(data_structs['dates_index'], fecha_i, fecha_f)
    list_acc = lt.newList("ARRAY_LIST")
    for value in lt.iterator(list_rank):
        for x in lt.iterator(value):
            if x["GRAVEDAD"] == g_accidente:
                lt.addLast(list_acc, x)

    param = lt.newList("ARRAY_LIST")
    lt.addLast(param, "CODIGO_ACCIDENTE")
    lt.addLast(param, "FECHA_HORA_ACC")
    lt.addLast(param, "DIA_OCURRENCIA_ACC")
    lt.addLast(param, "LOCALIDAD")
    lt.addLast(param, "DIRECCION")
    lt.addLast(param, "GRAVEDAD")
    lt.addLast(param, "CLASE_ACC")
    lt.addLast(param, "LATITUD")
    lt.addLast(param, "LONGITUD")

    grav_list = lt.newList("ARRAY_LIST")

    merg.sort(list_acc, comparereq4)

    sorted_list = lt.subList(list_acc,1,5)

    for accident in lt.iterator(sorted_list):
        aux = lt.newList("ARRAY_LIST")
        for pr in lt.iterator(param):
            lt.addLast(aux,accident[pr])
        lt.addLast(grav_list,aux["elements"])


    lt.addFirst(grav_list, param["elements"])

    return list_acc, grav_list["elements"]
```

Para el req 4 se solicita un rango de fechas y la gravedad deseada (mediante un upper() en el view siempre se recibe este argumento en mayúsculas), primero se sacan los accidentes por rango del mapa ordenado bajo la función values(), seguido a esto, se crea una lista donde se guardaran los accidentes con la gravedad solicitada, esto mediante un iterador que recorre la lista de rangos y agrega a la lista de accidentes si el parámetro ingresado coincide con el almacenado en el mapa.

Luego se crean los parámetros solicitados en la tabla del req mediante un array list siendo añadido al final mediante un addLast(). Para terminar, se crea una lista para los 5 accidentes más recientes, ordenando la lista mediante un Merge Sort y creando una sublista obtenemos dichos accidentes. Luego se recorre la sublista ya ordenada y se sacan los parámetros solicitados para al final obtener la lista de accidentes en el rango solicitado con la gravedad ingresada y la lista de los 5 más recientes ordenada.

Entrada	Esta función recibe el data structure, la fecha inicial y final del rango deseado junto con la gravedad requerida
Salidas	Retorna la cantidad total de accidentes con la gravedad solicitada dentro del rango de fechas definido junto con la lista de los 5 accidentes mas recientes dentro de esa primera lista.
Implementado (Sí/No)	Si, implementado por Jorge Solórzano

Análisis de complejidad

Pasos	**Complejidad**
Sacar la lista de accidentes por rango del mapa	O(LogN)
Recorrer la lista de accidentes creada y comparar si cuenta con la clave de gravedad ingresada	O(N)
Agregar al final a un Array List si coincide	O (1)
Crear la lista de parámetros (Array) y agregar al final	O (1)
Ordenar la lista de accidentes de menor a mayor mediante mergesort	O(NlogN)
Filtrar los datos ordenados de un sublist de 5 elementos max bajo los param creados	O (1)
Insertar los parámetros para el tabulate	O (1)
TOTAL	***O(NlogN)***

Pruebas Realizadas

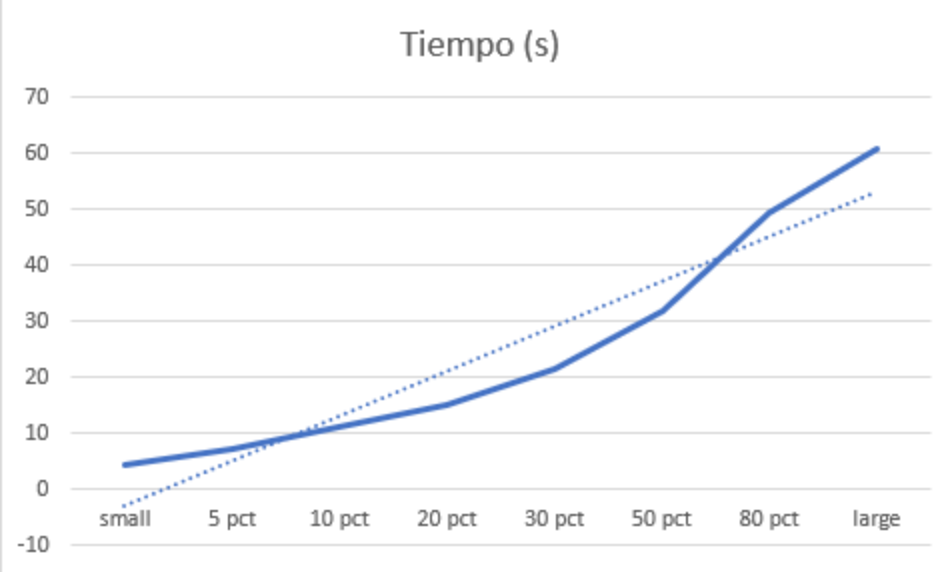
Procesadores	**AMD Ryzen 5 3400G with Radeon Vega Graphics 3.70 GHz**
Memoria RAM	16 GB
Sistema Operativo	Windows 11

Entrada	**Tiempo (s)**
small	4,462
5 pct	7,112
10 pct	11,043
20 pct	15,133
30 pct	21,427
50 pct	31,841
80 pct	49,594
large	60,750

Tablas de datos

Muestra	Salida	Tiempo (ms)
small	Accidentes con Muertos Entre 01/10/2016 y 01/10/2018	4,462
5 pct	Accidentes con Muertos Entre 01/10/2016 y 01/10/2018	7,112
10 pct	Accidentes con Muertos Entre 01/10/2016 y 01/10/2018	11,043
20 pct	Accidentes con Muertos Entre 01/10/2016 y 01/10/2018	15,133
30 pct	Accidentes con Muertos Entre 01/10/2016 y 01/10/2018	21,427
50 pct	Accidentes con Muertos Entre 01/10/2016 y 01/10/2018	31,841
80 pct	Accidentes con Muertos Entre 01/10/2016 y 01/10/2018	49,594
large	Accidentes con Muertos Entre 01/10/2016 y 01/10/2018	60,750

Graficas



Análisis

Comparado con los retos anteriores, la toma de datos fue más rápida manejando RBT. Mirando la grafica se tienen los resultados que esperaba, el tiempo varia dependiendo de los parámetros ingresados, pero la diferencia no se aleja de un comportamiento linearítmico. Si se manejara un árbol desordenado para este reto, la diferencia de tiempos seria muy notoria por la cantidad de búsquedas y comparaciones a realizar.

Requerimiento 5

Plantilla para el documentar y analizar cada uno de los requerimientos.

```
def req_5(data_structs, localidad, mes, año):
    """
    Función que soluciona el requerimiento 5
    """
    # TODO: Realizar el requerimiento 5

    # Se crea una lista con Los meses
    month_list = lt.newList("ARRAY_LIST")
    lt.addLast(month_list, "ENERO")
    lt.addLast(month_list, "FEBRERO")
    lt.addLast(month_list, "MARZO")
    lt.addLast(month_list, "ABRIL")
    lt.addLast(month_list, "MAYO")
    lt.addLast(month_list, "JUNIO")
    lt.addLast(month_list, "JULIO")
    lt.addLast(month_list, "AGOSTO")
    lt.addLast(month_list, "SEPTIEMBRE")
    lt.addLast(month_list, "OCTUBRE")
    lt.addLast(month_list, "NOVIEMBRE")
    lt.addLast(month_list, "DICIEMBRE")

    # Buscamos a que número corresponde el mes dado
    mes = lt.isPresent(month_list, mes)

    # Creamos un formato año/mes/día para el año y mes dado donde los días
comienzan en 01 y terminan en 31
    if mes != None:
        fecha_i = "{0}/{1:02d}/01".format(año, mes)
        fecha_f = "{0}/{1:02d}/31".format(año, mes)

    # Se obtienen los valores que están entre el rango tomando en cuenta los
extremos
    values = om.values(data_structs['dates_index'], fecha_i, fecha_f)

    # Lista para guardar los accidentes ocurridos en el rango de tiempo que
correspondan a la localidad dada
    local_list = lt.newList("ARRAY_LIST")

    # Se buscan en los accidentes ocurridos en el rango de tiempo que
correspondan a la localidad dada
    for dias in lt.iterator(values):
        for siniestro in lt.iterator(dias):
```

```
            if siniestro["LOCALIDAD"] == localidad:
                lt.addLast(local_list, siniestro)

    # Ordeno la lista de fecha mas reciente a menos reciente
    aux_list = merg.sort(local_list, compareDatesandhourreq5)

    # Hacer sublista de 10 si es mayor a 10 si no dejar igual
    if lt.size(aux_list) > 10:
        f_list = lt.subList(aux_list, 1, 10)
    else:
        f_list = aux_list

    # Parametros a utilizar
    param = lt.newList("ARRAY_LIST")
    lt.addLast(param, 'CODIGO_ACCIDENTE')
    lt.addLast(param, 'DIA_OCURRENCIA_ACC')
    lt.addLast(param, 'DIRECCION')
    lt.addLast(param, 'GRAVEDAD')
    lt.addLast(param, 'CLASE_ACC')
    lt.addLast(param, 'FECHA_HORA_ACC')
    lt.addLast(param, 'LATITUD')
    lt.addLast(param, 'LONGITUD')

    final_list = lt.newList("ARRAY_LIST")

    for siniestro in lt.iterator(f_list):
        aux_list = []
        for par in lt.iterator(param):
            aux_list.append(siniestro[par])
        lt.addLast(final_list, aux_list)

    return param, final_list
```

Descripción

Breve descripción de como abordaron la implementación del requerimiento

Para el requisito primero se convirtió el nombre del mes al número correspondiente. Posteriormente todo esto se convirtió al formato Año/mes/día. Se tomaron todos los valores en el rango de fechas y se buscaron solo aquellas que pertenecían a la localidad deseada. Luego, se pregunta si hay mas de diez accidentes en cuyo caso se hace una sublista con 10 elementos en caso de que no se toma lista completa. Finalmente, se toman los parámetros deseados.

Entrada	Estructura de datos, localidad, mes y año.
Salidas	Los parámetros utilizados y los 10 registros de accidentes más recientes en la localidad.

Implementado (Sí/No)	Sí, Juan Camilo Lyons Bustamante

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Creación de lista	O(1)
Insertar última posición lista	O(1)*12
Comparación	O(1)
Definición de variables	O(1)*2
Búsqueda rango de fechas	O(log(N))
Recorrer rango de fechas	O(n) donde n es menor que N
Ordenamiento	O(nlog(n)) donde n es menor que N
Comparación	O(1)
Sublista	O(10)
Creación de lista	O(1)
Insertar última posición lista	O(1)*8
Recorrer para obtener solo los parámetros	O(n*8) donde n es menor que N
TOTAL	***O(log(N))***

Pruebas Realizadas

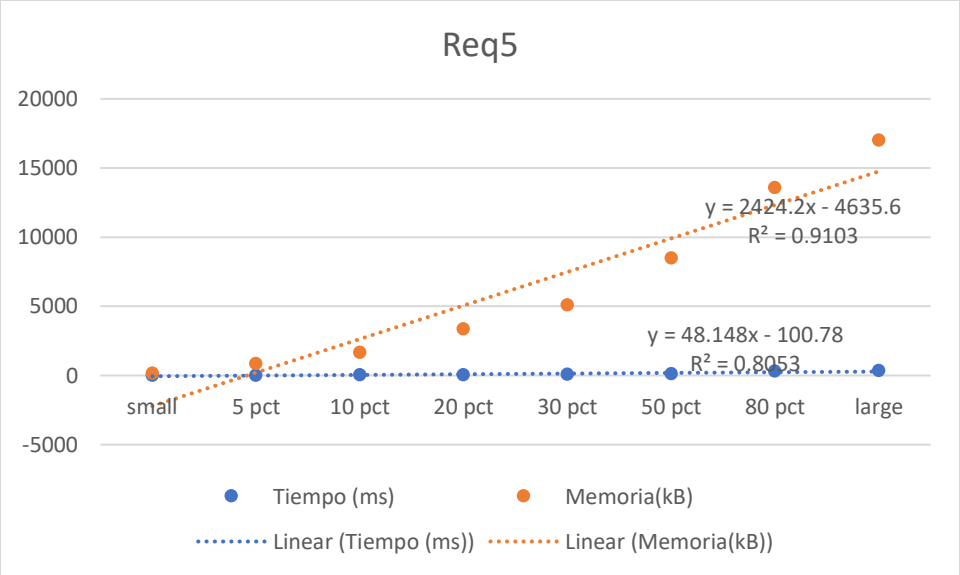
Descripción de las pruebas de tiempos de ejecución y memoria utilizada. Incluir descripción del procedimiento, las condiciones, las herramientas y recursos utilizados (librerías, computadores donde se ejecutan las pruebas, entre otros).

Procesadores	**Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz 2.20 GHz**
Memoria RAM	8.00 GB (7.85 GB usable)
Sistema Operativo	Windows 11 Home

Entrada	**Tiempo (ms)**	**Memoria(kB)**
small	0.27049998939037323	1.375
5 pct	0.5340999811887741	1.7265625
10 pct	0.8089999854564667	1.75
20 pct	1.5488000065088272	1.78125
30 pct	1.8655999898910522	1.8359375
50 pct	3.2822999954223633	1.84375
80 pct	5.850599989295006	1.84375
large	8.822200000286102	1.84375

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.

Se puede observar que la complejidad espacial permanece casi que constante con mientras que la complejidad temporal sí aumenta de una manera más significativa. Respecto al análisis de complejidad se puede observar que los valores encontrados, aunque en cierta medida son presentan el comportamiento, los valores de tiempo son muy bajos por lo que en general se puede afirmar que la complejidad temporal es de log(N).

Requerimiento 6

Descripción

```
def req_6(data_structs,topN,coordenadas,radio,mes,anio):
    """
    Función que soluciona el requerimiento 6
    """
    # TODO: Realizar el requerimiento 6
    mes = MonthNameToNumber(mes)
    lat1,lon1 = coordenadas
    lat1 = float(lat1)
    lon1 = float(lon1)

    rbt = data_structs["dates_index"]
    fecha_initial = "{0}/{1}/01".format(anio,mes)
    fecha_final = "{0}/{1}/31".format(anio,mes)

    values = om.values(rbt,fecha_initial,fecha_final)
    array = lt.newList("ARRAY_LIST")
    for element in lt.iterator(values):
        for accident in lt.iterator(element):
            lt.addLast(array,accident)

    near_accidents = mpq.newMinPQ(comparereq6)
    for accident in lt.iterator(array):
        lat2 = float(accident["LATITUD"])
        lon2 = float(accident["LONGITUD"])
        distancia = haversine(lat1,lon1,lat2,lon2)
        if distancia <= float(radio):
            lista_aux = lt.newList("ARRAY_LIST")
            lt.addLast(lista_aux,distancia)
            lt.addLast(lista_aux,accident)
            mpq.insert(near_accidents,lista_aux["elements"])
```

```
lista_min = lt.newList("ARRAY_LIST")
for i in range(0,topN):
    minimum = mpq.delMin(near_accidents)
    lt.addLast(lista_min,minimum)

param = ["CODIGO_ACCIDENTE","FECHA_HORA_ACC","DIA_OCURRENCIA_ACC",
        "LOCALIDAD","DIRECCION","GRAVEDAD","CLASE_ACC","LATITUD","LONGITUD","DISTANCIA(km)"]

lista_tabulate = lt.newList("ARRAY_LIST")
lt.addLast(lista_tabulate,param)
for accident in lt.iterator(lista_min):
    lista_aux = lt.newList("ARRAY_LIST")
    for x in param:
        if x == "DISTANCIA(km)":
            lt.addLast(lista_aux,str(round(accident[0],3)))
        else:
            lt.addLast(lista_aux,accident[1][x])
    lt.addLast(lista_tabulate,lista_aux["elements"])

return lista_tabulate["elements"]
```

```
def haversine(lat1, lon1, lat2, lon2):
    """
    Calcula la distancia entre dos puntos geográficos utilizando la ecuación de Haversine.
    Los puntos geográficos se especifican mediante su latitud y longitud en grados decimales.
    """
    R = 6371  # radio de la tierra en km

    # convertir latitud y longitud a radianes
    lat1, lon1, lat2, lon2 = map(math.radians, [lat1, lon1, lat2, lon2])

    # calcular la diferencia de latitud y longitud
    dlat = lat2 - lat1
    dlon = lon2 - lon1

    # aplicar la ecuación de Haversine
    a = math.sin(dlat / 2) ** 2 + math.cos(lat1) * math.cos(lat2) * math.sin(dlon / 2) ** 2
    c = 2 * math.asin(math.sqrt(a))
    distancia = R * c

    return distancia
```

```
def comparereq6(data1,data2):

    if data1[0] == data2[0]:
        return data1[1]["FECHA_OCURRENCIA_ACC"] < data2[1]["FECHA_OCURRENCIA_ACC"]
    else:
        return data1[0] > data2[0]
```

En este requerimiento se nos pide mostrar los N accidentes ocurridos dentro de una zona específica para un mes y un año. Se nos proporcionan las coordenadas en latitud y longitud en grados. Primero se ajustan las entradas para que concuerden con el formato utilizado en la estructura de datos. Después sacamos todos los accidentes del mes y año especificado usando el mapa de la estructura de datos. Para obtener los accidentes más cercanos necesitamos calcular la distancia en km desde el centro del área al lugar del accidente. Para esto usamos la ecuación de Haversine. Aplicamos esta ecuación usando una función aparte.

Ya teniendo la distancia, procedemos a obtener las menores distancias. Para esto se utilizó una cola de prioridad orientada a menor. De esta cola de prioridad se obtienen los N accidentes más cercanos a las coordenadas. Finalmente, se obtiene una lista de lista diseñada para tabulate con toda la información que se quiere mostrar al usuario.

Entrada	Estructura de datos, Número de accidentes, Coordenadas del centro del área (Latitud y Longitud), Radio del área en km, Mes, Año.
Salidas	Lista de listas diseñada para tabulate con los N accidentes ocurridos en el mes y año especificados más cercanos a las coordenadas ingresadas.
Implementado (Sí/No)	Si. Implementado por Juan Andrés Vargas B.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Búsqueda en el RBT	O(Log N)
Creación de arreglo	O(M) con M<N
Operaciones de Haversine	O(M)
Comparaciones	O(M)
Inserción cola de prioridad	O(Log R) con R<M
TOTAL	***O(M)***

Pruebas Realizadas

AMD Ryzen 5 4600H with Radeon Graphics
3.00GHz

8,00 GB (7,36 GB utilizable)

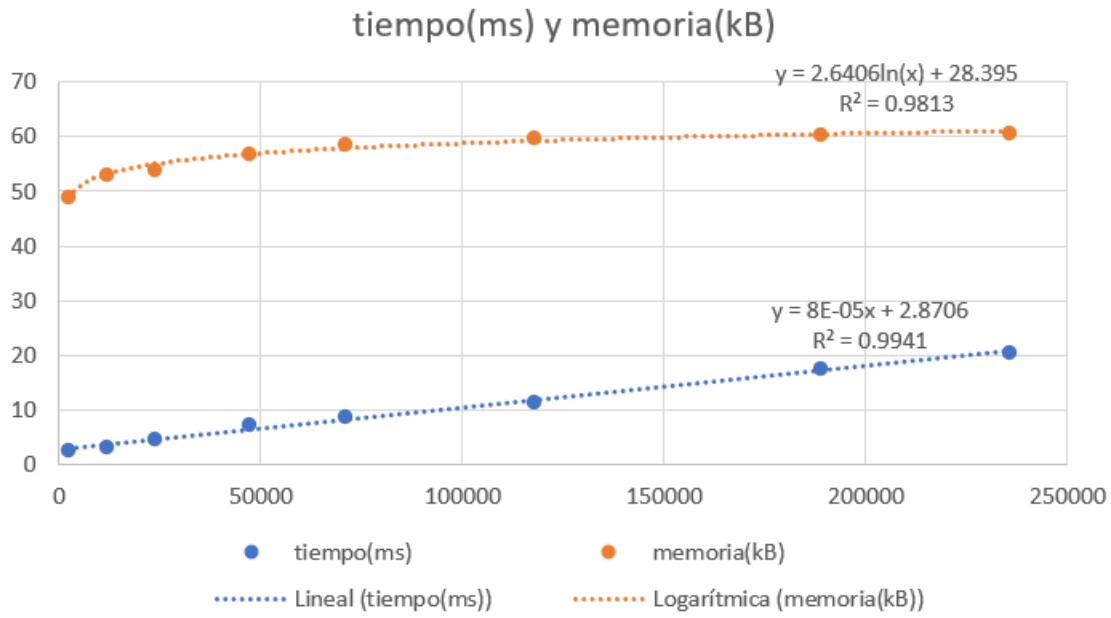
Windows 11 Sistema operativo de 64 bits, procesador x64

Entrada	**Tiempo (ms)**	**Memoria(kB)**
small	2.69	49.09
5 pct	3.25	53.16
10 pct	4.63	53.97
20 pct	7.31	56.90
30 pct	8.80	58.64

50 pct	11.52	59.79
80 pct	17.73	60.41
large	20.54	60.53

Graficas

Las gráficas con la representación de las pruebas realizadas.



Análisis

Observamos de la gráfica que el orden de complejidad de los tiempos es de orden lineal. Comportamiento acorde con la implementación ya que siempre se recorre todo el arreglo tanto para crearlo como para realizar las operaciones y comparaciones necesarias. El orden de complejidad del uso es memoria vuelve a tener un comportamiento logarítmico y creemos que se debe a la misma razón que se explicó en el requerimiento 3.

Requerimiento 7

Descripción

```python
def req_7(data_structs, mes, año):
    """
    Función que soluciona el requerimiento 7
    """
    # TODO: Realizar el requerimiento 7

    # Se crea una lista con los meses
    month_list = lt.newList("ARRAY_LIST")
    lt.addLast(month_list, "ENERO")
    lt.addLast(month_list, "FEBRERO")
    lt.addLast(month_list, "MARZO")
    lt.addLast(month_list, "ABRIL")
    lt.addLast(month_list, "MAYO")
    lt.addLast(month_list, "JUNIO")
    lt.addLast(month_list, "JULIO")
    lt.addLast(month_list, "AGOSTO")
    lt.addLast(month_list, "SEPTIEMBRE")
    lt.addLast(month_list, "OCTUBRE")
    lt.addLast(month_list, "NOVIEMBRE")
    lt.addLast(month_list, "DICIEMBRE")

    # Buscamos a que número corresponde el mes dado
    mes = lt.isPresent(month_list, mes)

    # Creamos un formato año/mes/día para el año y mes dado donde los días comienzan en 01 y terminan en 31
    if mes != None:
        fecha_i = "{0}/{1:02d}/01".format(año, mes)
        fecha_f = "{0}/{1:02d}/31".format(año, mes)

    # Se obtienen los valores y llaves que están entre el rango tomando en cuenta los extremos
    values = om.values(data_structs['dates_index'], fecha_i, fecha_f)

    # Lista de horas para todos los días del mes en el años dado
    hours_list = lt.newList("ARRAY_LIST")

    # Obtener todas las horas de accidentes
    for value in lt.iterator(values):
        for v in lt.iterator(value):
            # Se obtiene la hora solamente en int
            hour = int(v['HORA_OCURRENCIA_ACC'].split(':')[0])
            lt.addLast(hours_list, hour)

    # Ordenar la lista
    hours_list = merg.sort(hours_list, comparehourslist)

    # Parametros a utilizar
    param = lt.newList("ARRAY_LIST")
    lt.addLast(param, 'CODIGO_ACCIDENTE')
    lt.addLast(param, 'DIA_OCURRENCIA_ACC')
    lt.addLast(param, 'DIRECCION')
    lt.addLast(param, 'GRAVEDAD')
    lt.addLast(param, 'CLASE_ACC')
    lt.addLast(param, 'LOCALIDAD')
    lt.addLast(param, 'FECHA_HORA_ACC')
    lt.addLast(param, 'LATITUD')
    lt.addLast(param, 'LONGITUD')
```

```python
    # Ordenar los accidentes de cada uno de los días por horas
    for value in lt.iterator(values):
        value = merg.sort(value, compareHourreq7)

    # Nueva lista para valores
    values_list = lt.newList("ARRAY_LIST")

    # Obtengo una lista con todas las horas y parámetros
    for value in lt.iterator(values):
        first = lt.getElement(value, 1)
        last = lt.getElement(value, lt.size(value))

        # Obtener el penultimo ya que en el caso del primero se tiene en cuenta que si
        # el primero y el segundo tienen la misma hora se toma el que tenga menor código
        lastbutone = lt.getElement(value, lt.size(value))

        # Si el penultimo y el último tienen la misma hora el que tenga menor código se escogerá el de menor código
        if lastbutone['FECHA_HORA_ACC'] == last['FECHA_HORA_ACC'] and lastbutone['CODIGO_ACCIDENTE'] < last['CODIGO_ACCIDENTE']:
            last = lastbutone

        aux_list = lt.newList("ARRAY_LIST")

        dictionary = {}
        for p in lt.iterator(param):
            dictionary[p] = first[p]
        lt.addLast(aux_list, dictionary)

        dictionary = {}
        for p in lt.iterator(param):
            dictionary[p] = last[p]
        lt.addLast(aux_list, dictionary)

        lt.addLast(values_list, aux_list)

    return hours_list, values_list
```

Para este requerimiento, se comienza creando una lista que contendrá los meses como parámetro. Luego buscamos que el mes ingresado como parámetro sea valido y este en la lista creada. Creamos un formato año/mes/día para los parámetros dados donde el día comienda en 01 y termina en 31.

Luego sacamos los valores del mapa dentro del rango de dado, y se crea una lista para las horas del día, la cual se llena iterando los valores dados por el mapa. Seguido a esto, ordenamos la lista de horas de menor a mayor. Creamos la lista de parámetros para la tabla, ordenamos los accidentes de cada día por horas y luego iteramos para sacar una lista con todas las horas y parámetros dados.

Por último, sacamos el penúltimo valor de la lista para compararlo con el ultimo y ver cual tiene menor código de accidente, lo agregamos a un diccionario de día y obtenemos un diccionario con los accidentes ocurridos en ese día y otro con los accidentes ocurridos por horas, el cual se agrega a una lista para tabular.

Entrada	Este requerimiento recibe el data structure, junto con un mes y año.
Salidas	Retorna una tabla por día con el accidente mas temprano y mas tarde de un mes dado, junto con una grafica con los accidentes ocurridos por hora en el mes.
Implementado (Sí/No)	Si, implementado por Juan Camilo Lyons.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Crear la lista de meses	O (1)
Comparar si el mes ingresado es correcto	O (N)
Crear el formato de año mes día	O (2)
Sacar las fechas del árbol en el rango dado	O (LogN)
Crear la lista de horas	O (N)
Crear la lista de parámetros e insertarlos	O (1)
Ordenar los accidentes para cada día	O (NLogN)
Sacar el primer y ultimo valor	O (2)
Obtener el penúltimo valor	O (2)
Crear los diccionarios hora	O (N)
Crear los diccionarios día	O (N)
TOTAL	***O(NLogN)***

Pruebas Realizadas

Procesadores	**AMD Ryzen 5 3400G with Radeon Vega Graphics 3.70 GHz**
Memoria RAM	16 GB
Sistema Operativo	Windows 11

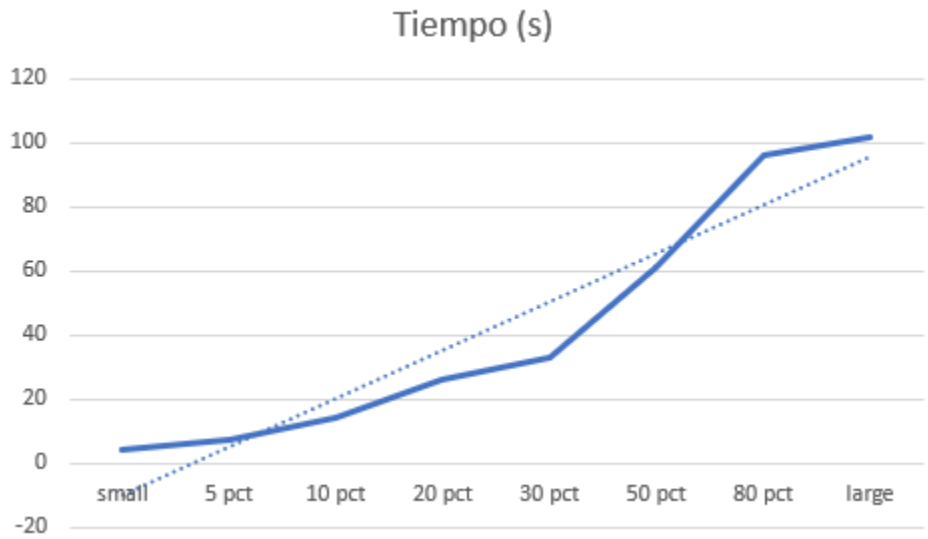
Entrada	Tiempo (s)
small	4,36
5 pct	7,446
10 pct	14,279
20 pct	25,970
30 pct	32,994
50 pct	61,331
80 pct	96,236
large	101,626

Tablas de datos

Muestra	Salida	Tiempo (ms)
small	Información de los accidentes más tempranos y más tardes para cada día del mes de diciembre del año 2019	4,36
5 pct	Información de los accidentes más tempranos y más tardes para cada día del mes de diciembre del año 2019	7,446
10 pct	Información de los accidentes más tempranos y más tardes para cada día del mes de diciembre del año 2019	14,279
20 pct	Información de los accidentes más tempranos y más tardes para cada día del mes de diciembre del año 2019	25,970
30 pct	Información de los accidentes más tempranos y más tardes para cada día del mes de diciembre del año 2019	32,994
50 pct	Información de los accidentes más tempranos y más tardes para cada día del mes de diciembre del año 2019	61,331
80 pct	Información de los accidentes más tempranos y más tardes para cada	96,236

	día del mes de diciembre del año 2019	
large	Información de los accidentes más tempranos y más tardes para cada día del mes de diciembre del año 2019	101,626

Graficas



Análisis

A pesar de la cantidad de procesos, el tiempo es optimo para lo esperado. Por lo que se ve en la mayoría de req, se mantienen NlogN, pero igual varían los tiempos según los parámetros, en la grafica se ve un salto grande que vuelve a estabilizarse al tener una mayor cantidad de datos, manteniéndose cerca de la tendencia definida. En cuanto a memoria, no lo medimos pero a simple vista ocupa menos de la que lograríamos ocupar usando otra estructura de datos.

Requerimiento 8

Plantilla para el documentar y analizar cada uno de los requerimientos.

```
def req_8(data_structs, fecha_inicial, fecha_final, clase):
    """
    Función que soluciona el requerimiento 8
    """
```

```
    # TODO: Realizar el requerimiento 8

    # Modificar formato de fecha
    fecha_inicial = "/".join(fecha_inicial.split("/")[::-1])
    fecha_final = "/".join(fecha_final.split("/")[::-1])

    # Obtener los valores correspondientes al rango de fechas
    values = om.values(data_structs['dates_index'], fecha_inicial, fecha_final)

    # Lista para guardar los siniestros
    values_list = lt.newList("ARRAY_LIST")

    #Lista de parametros
    param = lt.newList("ARRAY_LIST")
    lt.addLast(param, "FECHA_OCURRENCIA_ACC")
    lt.addLast(param, "HORA_OCURRENCIA_ACC")

    # Para guardar los siniestros que esten en la clase dada y solo los parámetros deseados
    for value in lt.iterator(values):
        for v in lt.iterator(value):
            if v["CLASE_ACC"] == clase:
                dictionary = {"GRAVEDAD": v["GRAVEDAD"], "LOCATION": [float(v["LATITUD"]), float(v["LONGITUD"])]}
                for p in lt.iterator(param):
                    dictionary[p] = v[p]
                lt.addLast(values_list, dictionary)

    return values_list
```

```
def print_req_8(control, fecha_inicial, fecha_final, clase, Memory):
    """
        Función que imprime la solución del Requerimiento 8 en consola
    """
    # TODO: Imprimir el resultado del requerimiento 8
    clase = clase.upper()
    delta, req8 = controller.req_8(control, fecha_inicial, fecha_final, clase, Memory)
    print("Hay {0} accidentes entre las fechas {1} y {2}.".format(len(req8['elements']), fecha_inicial, fecha_final))
    my_map = folium.Map(location = [4.64873653, -74.07124224], zoom_start=16)
    c = {"SOLO DANOS" : "green", "CON HERIDOS" : "blue", "CON MUERTOS" : "red"}
    mc = MarkerCluster().add_to(my_map)
    for loc in lt.iterator(req8):
        folium.Marker(
        location = loc["LOCATION"],
        popup = "Fecha: {0}\n Hora:{1}".format(loc["FECHA_OCURRENCIA_ACC"], loc["HORA_OCURRENCIA_ACC"]),
        icon = folium.Icon(color= c[loc["GRAVEDAD"]]),).add_to(mc)
```

```
my_map.save('index.html')
if len(delta) == 2:
    print("Tiempo: {0} ms \n Memoria: {1} kb".format(delta[0], delta[1]))
else:
    print("Tiempo: {0} ms".format(delta[0]))
```

Descripción

Breve descripción de como abordaron la implementación del requerimiento

Primero se busca los valores que estén entre el rango de fechas, luego se revisa cada uno para ver si concuerda con la clase deseada y se extraen los parámetros deseados. Luego utilzamos la librería folium para crear un mapa centrado en una localización. Posteriormente se hace un diccionario para asignar colores según la gravedad. Se crean clusters donde se agregarán marcadores correspondientes a los sinisestros. Finalmente, por cada marcador se toma una localización y dos datos que son la fecha y la hora además de asignar un color para luego guardar todo el mapa en un archivo html.

Entrada	Estructura de datos, fecha inicial, fecha final y clase.
Salidas	El número total de accidentes ocurridos en el rango de fecha y un mapa interactivo de clústeres que muestre todos los accidentes según su tipo en el mapa de Bogotá, en donde cada marcador tenga asignado un color de acuerdo con su gravedad.
Implementado (Sí/No)	Sí, Juan Camilo Lyons Bustamante

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Búsqueda rango de fechas	O(log(N))
Recorrer rango de fechas	O(n) donde n es menor que N
TOTAL	***O(log(N))***

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).

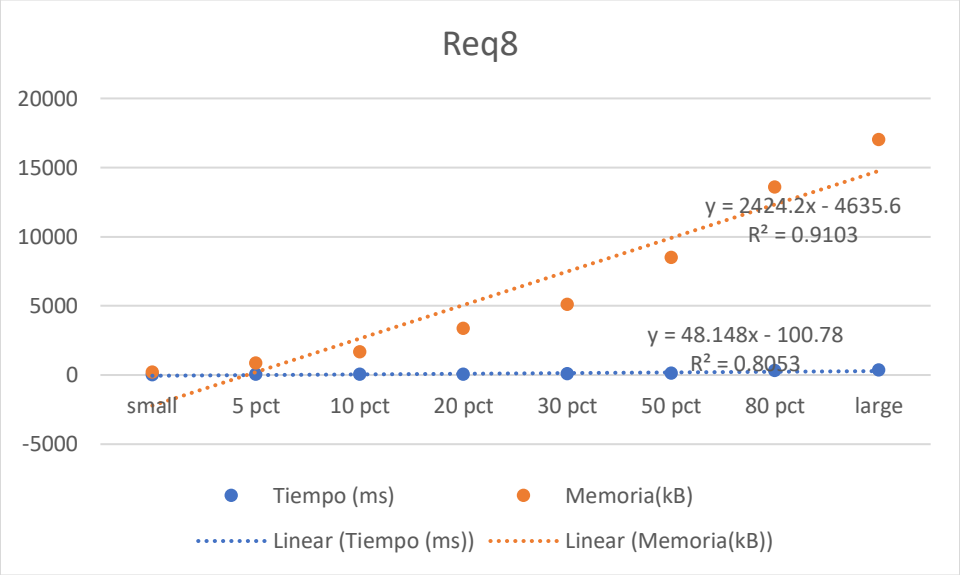
Procesadores	**Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz 2.20 GHz**
Memoria RAM	8.00 GB (7.85 GB usable)
Sistema Operativo	Windows 11 Home

Entrada	**Tiempo (ms)**	**Memoria(kB)**
small	6.001200005412102	173.875
5 pct	18.487900003790855	838.046875

10 pct	25.83220000565052	1669.875
20 pct	53.12009999155998	3368.921875
30 pct	69.35819998383522	5075.34375
50 pct	113.61049999296665	8479.390625
80 pct	292.79919999837875	13583.0625
large	347.89710000157356	16998.984375

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.

Se puede observar en la anterior gráfica que lo que más aumenta en la gráfica es la memoria. Por otro lado, los aumentos en el tiempo son comparables a un log(N) debido a que el aumento en el tamaño de N supone un menor aumento en el tiempo de respuesta cuyo comportamiento es que a medida que aumenta a valores mayores el aumento es menor.