ANÁLISIS DEL RETO

Tomas Diaz, 202220658, t.diazv@

Samuel Peña, 202028273, ss.pena@

Manuel Pinzón, 202125748, mma.pinzonpi@

Carga y Estructura de datos

La carga de datos y estructura de datos se basa en la estructura del laboratorio 9. Se va leyendo el csv y cada dato se vuelve un diccionario y se añade al data structs en 2 instancias: a la lista total de accidentes (array) y al árbol RBT que tiene por llave la fecha_hora en formato entero de 16 dígitos y por valor un diccionario con dos valores: por un lado la fecha y hora en formato normal y por el otro una lista (en la mayoría de los casos si no en todos, de único elemento) con los accidentes en ese instante (así se soluciona el problema mencionado en clase de que hayan 2 accidentes reportados en el mismo instante).

Pensamos que es más practico cargar únicamente el árbol por instantes en el data structs, sin mayor complejización de la estructura. Pues si bien se pudo por ejemplo dentro de cada nodo del árbol de fechas implementar un hash que clasifique los accidentes por gravedad o localidad por ejemplo, esto subiría la complejidad de la carga de datos y a la larga sería más demorado para los usuarios que en esencia somos nosotros al hacer las pruebas. Así teniendo en cuenta que en todo caso antes de solicitar algo siempre deben cargarse los datos, pensamos que esta implementación es no solo más sencilla de hacer, sino más eficiente.

Requerimiento 1

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

```python
def organizar_rango_fechas_mas_reciente(data_structs,fecha1, fecha2):
    """
    Función que soluciona el requerimiento 1
    """
    # TODO: Realizar el requerimiento 1

    valor_fecha1 = fecha_hora_inicial(fecha1)

    valor_fecha2 = fecha_hora_final(fecha2)
    "se convierte la fecha dada a tipo de llave"

    llaves = om.values(data_structs["model"]["dateIndex"],valor_fecha1, valor_fecha2)
    "se sacan los que esten en el rango dado"
    respuesta = lt.newList()
    tamanio = lt.size(llaves)
    i =1
    while i <= tamanio:
        pos = lt.getElement(llaves,i)
        "se saca el valor acorde en la lista"
        dato = pos["lista_accidentes"]
        "se deja solo la lista de accidentes "

        valor = lt.getElement(dato,1)
        "solo tendra un dato "
        lt.addFirst(respuesta,valor)
        i+=1

    return respuesta
def req1(data_structs,fecha1, fecha2):
    respuesta = organizar_rango_fechas_mas_reciente(data_structs,fecha1, fecha2)
    return respuesta
```

Descripción

Breve descripción de como abordaron la implementación del requerimiento

Este requerimiento solo consiste de una función, que devuelve una lista con los valores según el rango dado, en primera instancia convierte las fechas dadas a las que están distribuidas en el mapa, que están son de este estilo (2022090400000000). Después de saber a qué rango quiero tenerlo, invoco la función de values que me devuelve todas las fechas. Ahí como el nodo del árbol tiene un diccionario de dos llaves, una la fecha actual y la otra la lista en esa fecha. Entonces, agrego los elementos (los accidentes) a una listo y eso es lo que retorno.

Entrada	Fecha inicial (YY/MM/DD) y fecha final (YY/MM/DD)
Salidas	Lista de accidentes en el rango dado
Implementado (Sí/No)	Si, Tomas Diaz

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Paso 1: llamado a funcion organizar más recientes	O(h+k)
Paso 2: fecha inicial (se ajusta la fecha dada para poder ser comparada en el árbol)	O (1)

Paso3: fecha final (se compara la fecha dada para poder ser comparada con los rangos, al final del día dado)	O (1)
Paso 4: values(en el mapa de los datos cargos, se saca una lista según los que estén dentro del rango)	O(h+K)
Paso 5: creo lista donde será la respuesta	O (1)
Paso 6: le saco el tamaño	O (1)
Paso 7: while: en este while se va por la lista dada, pero como este tiene dos llaves, se saca dónde está localizada la de lista de accidentes, y se agrega a la lista	O(h+k)
TOTAL	***O(h+k)***

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).

Para la toma de datos se utiliza la prueba que esta puesta en el documento que es 01/11/2016 y el 08/11/2016

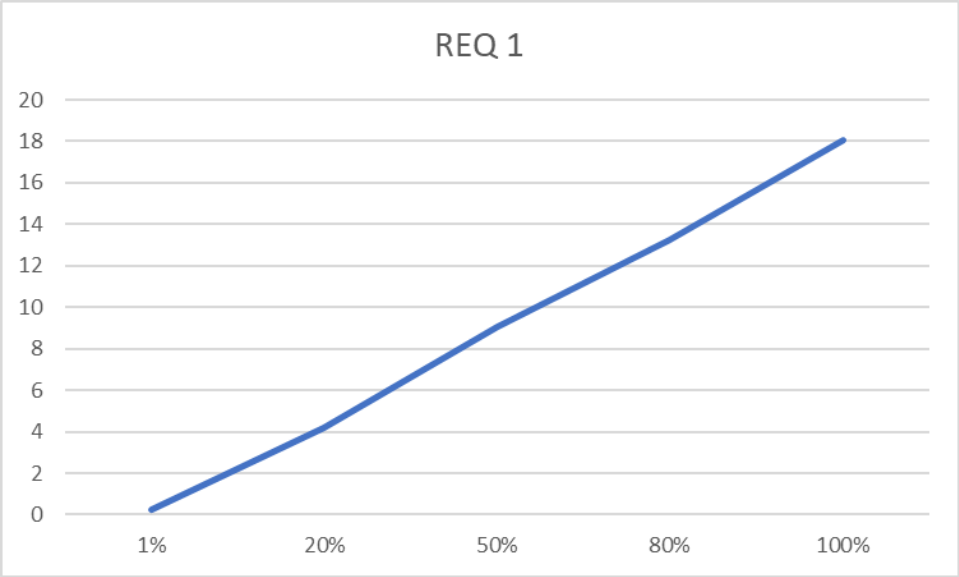
Entrada	**Tiempo (s)**
1%	0.2599
20%	4.155
50%	9.06
80%	13.23
100%	18.03

Tablas de datos

Las tablas con la recopilación de datos de las pruebas.

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.

En este requerimiento logre obtener una buena complejidad, porque a la hora de implementar árboles, se hace mucho más fácil la búsqueda entre un rango especifico de valores, solo tenía que convertir las fechas dadas a las llaves de comparación con el árbol

Requerimiento 2

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

```python
def req_2(data_structs, anio , mes , hora_i, hora_f):
    """
    Función que soluciona el requerimiento 2
    """
    # TODO: Realizar el requerimiento 2
    resp_m= aux_mes(mes)
    diasm = lt.getElement(resp_m , 1)
    mes = lt.getElement(resp_m , 2)
    i_d = 1
    respuesta = lt.newList()
    while i_d <= diasm :
        dia = str(i_d)
        dia_f = str(i_d)
        if i_d < 10:
            dia = str("0" + dia)
            dia_f = str("0" + dia_f)
        mes_d = str(mes)
        fecha_1 = str(anio + "/" + mes_d + "/" + dia + " " + hora_i)
        fecha_2 = str(anio + "/" + mes_d + "/" + dia_f + " " +hora_f)
        fechas = organizar_rango_fechas_menos_reciente(data_structs , fecha_1, fecha_2)
        fechas_s = lt.size(fechas)
        i_f = 1
        while i_f <= fechas_s:
            pos = lt.getElement(fechas ,i_f)
            lt.addFirst(respuesta ,pos)
            i_f += 1
        i_d += 1
    return respuesta
```

Descripción

Breve descripción de como abordaron la implementación del requerimiento

En este requerimiento se buscan todos los accidentes de un año y mes específicos, dentro de una franja horaria. A través del uso de un árbol binario ordenado, sacamos los valores que están en la franja horaria para cada día del mes/año estudiado. Y así podemos encontrar todos los accidentes que estan en ese mes de ese año durante la franja horaria.

Entrada	Recibe el año, mes, hora inicial y hora final
Salidas	Devuelve una lista con los accidentes dentro de esa franja de tiempo
Implementado (Sí/No)	Si, Manuel Pinzon

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Paso 1: Se cambia el número y días de un mes, y se asignan a una variable	O(1)
Paso 2: Se itera para todos los días del mes, pero en esta iteración solo hay una función que, si tiene un valor variable, y es la que usa el árbol	O(h+k)

Paso 3: Se hace una iteración para insertar todos estos valores en una lista	O(1)
TOTAL	***O(h+k)***

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).

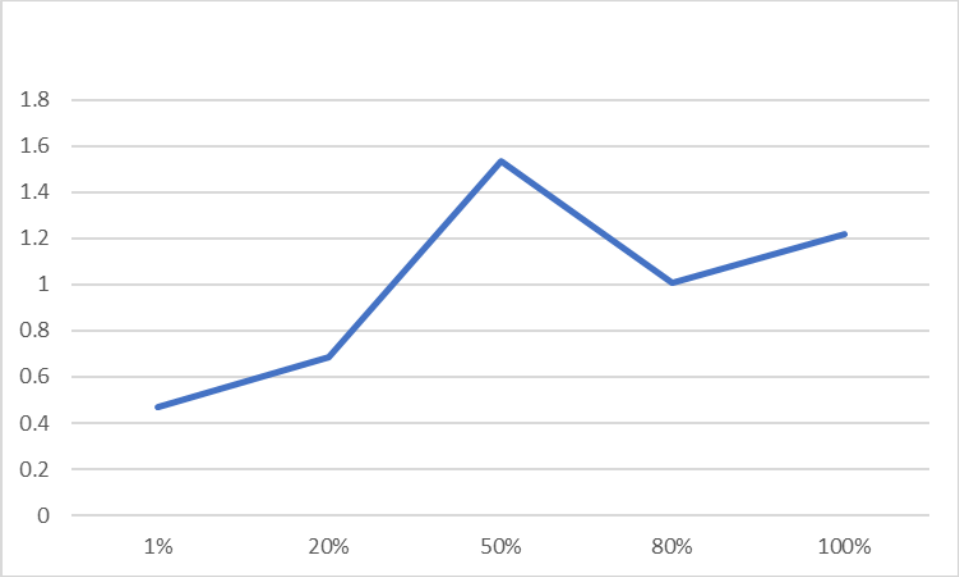
Entrada	**Tiempo (s)**
1%	0.47
20%	0.683
50%	1.534
80%	1.005
100%	1.22

Tablas de datos

Las tablas con la recopilación de datos de las pruebas.

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.

Al usar un arbol binario para la base de datos de este requerimiento podemos disminuir la complejidad significativamente, ya que se enfoca en sacar valores dentro de un rango. Y al usar este tipo de base de datos podemos facilitar la complejidad del resto de la función.

Requerimiento 3

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

```
def req_3(data_structs , clase , calle):
    """
    Función que soluciona el requerimiento 3
    """
    # TODO: Realizar el requerimiento 3
    fecha_1 = "2005"
    fecha_2 = "2025"
    fechas = organizar_rango_fechas_mas_reciente(data_structs, fecha_1 , fecha_2)
    tam = lt.size(fechas)
    num = 1
    respuesta = lt.newList("ARRAY_LIST")
    resultados = lt.newList("ARRAY_LIST")
    while num <= tam:
        accidente = lt.getElement(fechas , num)
        comparacion = aux_verdadero(accidente , clase, calle)
        if comparacion == True:
            lt.addLast(resultados , accidente)
        num += 1
    tam_resp = lt.size(resultados)
    pos = 1
    if tam_resp >= 3:
        while pos <= 3:
            resp = lt.getElement(resultados, pos)
            lt.addLast(respuesta, resp)
            pos += 1
        return respuesta
    else:
        return resultados
```

Descripción

Breve descripción de como abordaron la implementación del requerimiento

En este requerimiento se busca encontrar los accidentes que cuentan con la misma clase y calle dada. Una vez tengamos todos los accidentes en una lista, iteramos para verificar si este cumple o no con los requisitos. Luego se adicionan estos resultados a la lista de respuestas.

Entrada	La clase de choque y la calle
Salidas	Una lista con todos los accidentes que cumplen con esos criterios
Implementado (Sí/No)	Si por Manuel Pinzon

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**

Paso 1: Se usa una función para sacar todos los valores de forma ordenada y en forma de lista	O(h+k)
Paso 2: Se sacan los accidentes de esta lista, y se verifica a través de una función externa que cumpla con los criterios y se añaden a una lista	O(1)
Paso 3: Si la lista tiene más de 3 elementos, entonces se le adjuntan solo los primeros tres resultados a una lista y se da esa como respuesta	O(1)
TOTAL	***O(h+k)***

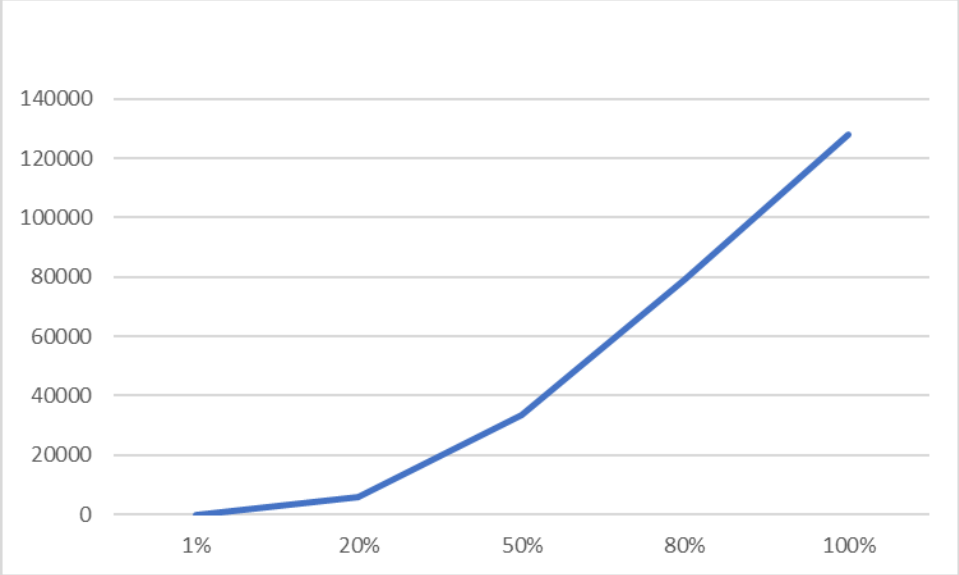
Pruebas Realizadas

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).

Entrada	**Tiempo (s)**
1%	17.028
20%	5475.55
50%	33672.1
80%	79.2587
100%	127896.9

Tablas de datos

Las tablas con la recopilación de datos de las pruebas.



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.

El uso de un árbol binario ordenado para organizar todos los datos de forma decreciente nos ayuda a poder disminuir los tiempos y la complejidad. Si no fuera por cómo están estos datos se necesitaría abordar el problema de una forma completamente distinta, lo cual perjudicaría la complejidad y los tiempos.

Requerimiento 4

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

```
def req_4(data_structs,fecha1,fecha2,gravedad):
    """
    Función que soluciona el requerimiento 4
    """
    # TODO: Realizar el requerimiento 4
    lista_fechas = organizar_rango_fechas_mas_reciente(data_structs,fecha1,fecha2)
    tamanio = lt.size(lista_fechas)
    i = 1
    respuesta = lt.newList("ARRAY_LIST")
    while i<= tamanio:
        "verificar si la gravedad que esta es la que quiero, si es asi la agrego a una lista "
        especifico = lt.getElement(lista_fechas,i)
        if especifico["GRAVEDAD"] == gravedad:
            lt.addLast(respuesta,especifico)
        i+=1

    size = lt.size(respuesta)
    "tamanio de cuantos cumplen los requisitos para imprimir"
    final = lt.newList("ARRAY_LIST")
    a=1
    if size >=5:
        "solo para imprimir los 5 "
        while a <= 5:
            valor = lt.getElement(respuesta,a)
            lt.addLast(final,valor)
            a+=1
        return final,size
    else:
        return respuesta, size
```

Descripción

Breve descripción de como abordaron la implementación del requerimiento

Este requerimiento ordene los datos según las fechas dadas, después pase elemento por elemento revisando si el valor del dato contaba con la gravedad que estaba buscando, si era verdad lo agregue a una lista. Finalmente, lo itere 5 veces para encontrar el menor en la lista (que como ya estaba organizado previamente, era simplemente sacar los primeros 5)

Entrada	Fecha inicial (YY/MM/DD), fecha final (YY/MM/DD), gravedad
Salidas	Lista de los 5 elementos más recientes según su gravedad
Implementado (Sí/No)	Si. Tomas Diaz

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Paso 1: lista_fechas: llamar función organizar rango por fechas	O(h+k)
Paso 2: tamanio :sacarle el tamaño a la lista para iterarla	O (1)
Paso 3: respuesta: lista después de iterar	O (1)
Paso 4: en este while se pasa por todas listas fechas para verificar si consta de la gravedad que se está buscando, de ser así se agrega a la lista de arriba	O(n) una n más pequeña porque es en el rango que se le muestra
Paso 5: size: es igual al tamaño de la lista que cumple con la gravedad	O (1)
Paso 6: final: se va a crear la lista con el top 5 de los más recientes	O (1)
Paso 7: se verifica si tiene más de 5 eventos, y ahí como la lista respuesta ya está ordenada en lo más reciente, solo se copia y pega los primero 5 de la lista, si no tiene más de 5 eventos se devuelve la lista respuesta, con el tamaño	O (1)
TOTAL	***O(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).

Los datos utilizaos son los de 001/01/2016 y 01/08/2016 con gravedad de CON muertos

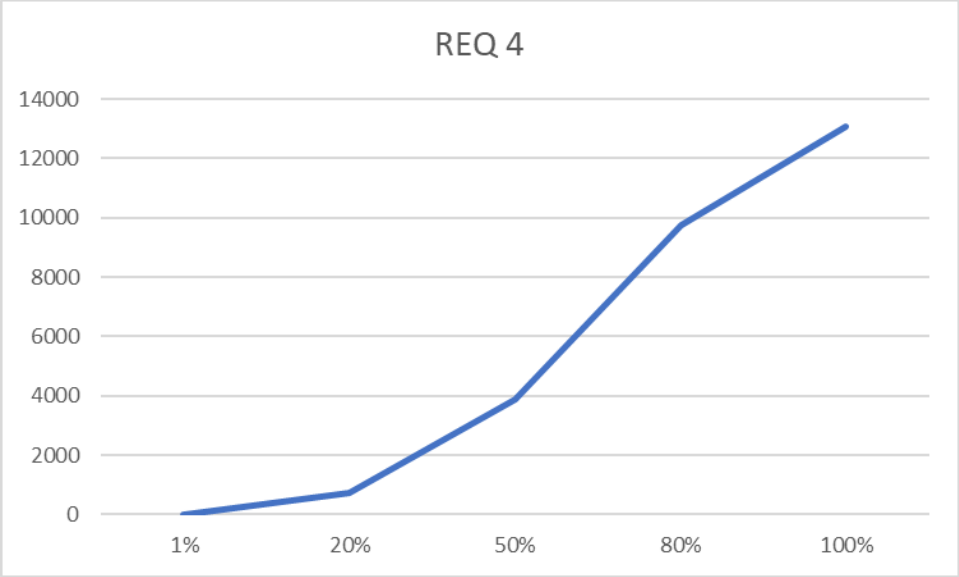
Entrada	**Tiempo (s)**
1%	5.94
20%	721.38
50%	3892.65
80%	9768.99
100%	13078.13

Tablas de datos

Las tablas con la recopilación de datos de las pruebas.

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.

En este requerimiento logre obtener una buena complejidad, porque a la hora de implementar árboles, se hace mucho más fácil la búsqueda entre un rango especifico de valores, solo tenía que convertir las fechas dadas a las llaves de comparación con el árbol. Después, solo tuve pequeñas iteraciones para saber si el accidente contaba con la gravedad que yo estaba buscando.

De igual manera, se demora harto la búsqueda de datos, porque hay un rango muy grande entre los años en los que se debe buscar, y adicionalmente mi computador no es de los más modernos entonces le cuesta un poco, si se hacen pruebas con rangos de un mes lo saca instantáneo.

Por último, este requerimiento se puede llegar a hacer en o(logn), pero yo opte por hacerlo en o(n), porque para lograr la otra velocidad, se le tendría que mandar a la carga de datos del comienzo un árbol que los clasifique por gravedad, y esto haría que se demorara mucho más la carga de datos, y ya no fuera tan eficiente. Otra opción, era hacerlo dentro del mismo requerimiento, pero esta podría llegar a tener una complejidad de O(nlogn), haciendo que sea más eficiente la implementación que utilice, porque llega a ser un o(N) que no es mala complejidad, pero al haber tantos datos si se puede llegar a demorar cuando se usan lapsos de 5 años.

Requerimiento 5

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

```python
def req_5(data_structs,anio,mes,localidad):
    """
    Función que soluciona el requerimiento 5
    """
    arbol_fechas = data_structs['dateIndex']
    lim_inf = int(anio+mes)*10**10
    lim_sup = (int(anio+mes)+1)*10**10
    lista_10 = []
    ##rango en single_linked
    rango_siniestros = om.values_array(arbol_fechas,lim_inf,lim_sup)
    ###iterar rango hasta obtener los 10 más recientes de la localidad
    len_rango = lt.size(rango_siniestros)
    i=0
    while i<=len_rango:
            len_10 =len(lista_10)
            if len_10 ==10:
                break
            lista_nodo = lt.getElement(rango_siniestros,len_rango-i)['lista_accidentes']
            len_lista_nodo = lt.size(lista_nodo)
            j=1
            while j<=len_lista_nodo:
                accidente = lt.getElement(lista_nodo,j)
                if accidente['LOCALIDAD']==localidad:
                    lista_10.append(accidente)
                len_10 = len(lista_10)
                if len_10==10:
                    break
                j+=1
            i+=1
    return lista_10
```

Descripción

Breve descripción de como abordaron la implementación del requerimiento

Este requerimiento retorna una lista de los 10 accidentes en el mes y año dados que hayan ocurrido en la localidad dada. Lo primero que hace es obtener, a partir del rango de enteros de las llaves, un array con los valores en ese rango. Después va iterando el dicho array de atrás para adelante y dentro de él las listas de accidentes con fecha para comparar accidente por accidente con la localidad y de ser igual, ingresarla a la lista de respuesta. El ciclo dura hasta que el tamaño de la lista de respuesta sea 10 o se abarque todo el rango, lo que pase primero. Por último se retorna la dicha lista.

Entrada	Data Structs, año, mes, localidad
Salidas	Lista de los 10 (o menos) accidentes mas recientes del mes en la localidad
Implementado (Sí/No)	Si, por Samuel Peña

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**

Creación árbol, cálculo de limites superior e inferior del mes y creación de lista vacía de respuesta	O(1)
Devolver array de los valores en el rango (está ordenado)	O(h+k)
Iterar array de atrás hacia adelante comparando la localidad e ir agregando a la lista respuesta si es la misma localidad hasta que el tamaño de esta sea 10. Devolver lista.	O(n)
TOTAL	***O(n)***

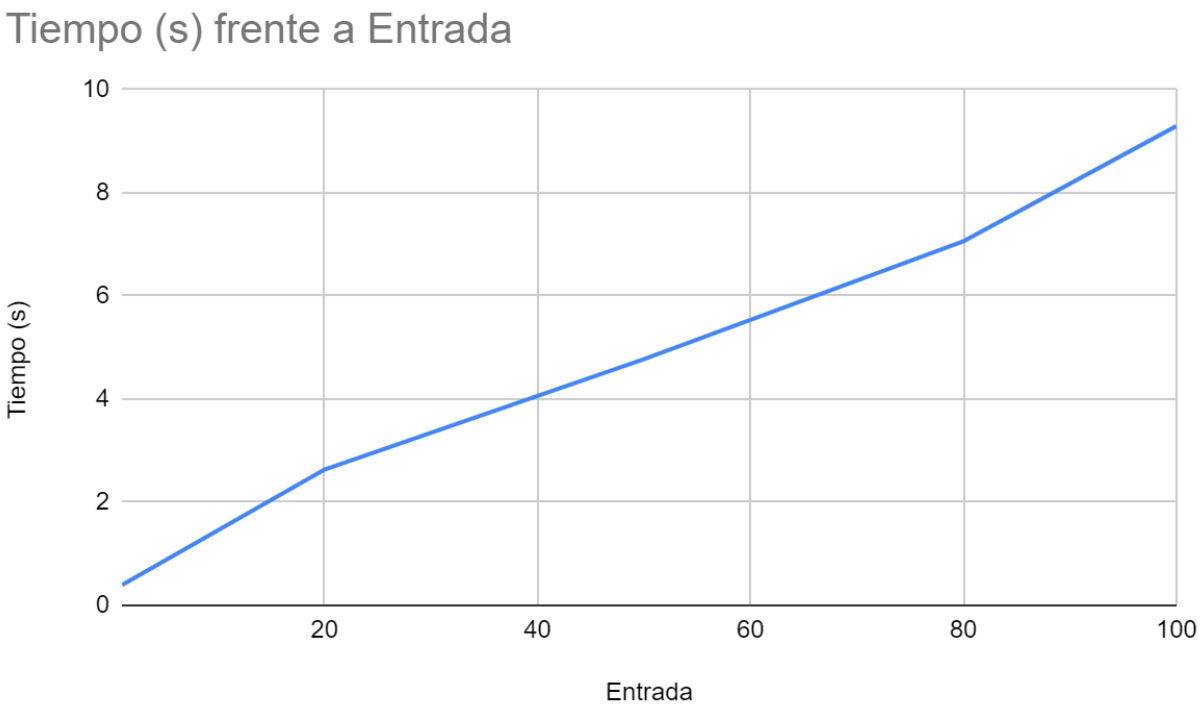
Pruebas Realizadas

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

Se dieron por parámetros el año 2019, el mes 12 y la localidad FONTIBON

Entrada	**Tiempo (s)**
1%	0,3965999996289611
20%	2,632399999536574
50%	4,770799999125302
80%	7,058299999684095
100%	9,292200000025332

Graficas



Análisis

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

Resultó funcionar a la perfección, pues como muestra la gráfica, se logró una complejidad del orden de O(n), pues la operación más compleja en el peor caso era iterar el rango dado, que en cualquier caso es muy inferior al n total de los datos.

Requerimiento 6

```python
def req_6(data_structs,anio,mes,latitud,longitud,radio,n_actividades):
    """
    Función que soluciona el requerimiento 6
    """
    # TODO: Realizar el requerimiento 6
    codificacion = anio + mes + "0"*10
    if mes == "12":
        anio = int(anio) + 1
        mes = int(mes) - 11
        codificacion_siguiente_mes = str(anio) +"0" + str(mes) + "0"*10
        "calculo el siguiente mes dependiendo si es diciembre "

    else:
        mes = int(mes) + 1
        codificacion_siguiente_mes = str(anio) + "0" + str(mes) + "0"*10
        "el valor que se le dara al mes indicado "

    lista_intervalo_valores = om.values(data_structs["model"]["dateIndex"],codificacion,codificacion_siguiente_mes)
    "lista del intervalo "

    tamanio = lt.size(lista_intervalo_valores)
    i = 1
    heap = mpq.newMinPQ(cmpfunction=compareradio)
    mapa = mp.newMap()
```

```python
    while i <=tamanio:
        exacto = lt.getElement(lista_intervalo_valores,i)
        pos = lt.getElement(exacto["lista_accidentes"],1)
        distancia = funcion_distancias_lat_long(float(latitud), float(longitud), float(pos["LATITUD"]), float(pos["LONGITUD"]))
        "se saca la distancia en la que esta "
        if distancia <= int(radio):
            mpq.insert(heap,distancia)
            mp.put(mapa,distancia,pos)
            "si es verdad se agrega a un heap "
            "Mapa con llave la distancia y respuesta lo que quiero dar "
        i+=1



    respuesta = lt.newList()
    a = 1
    while a <= int(n_actividades):
        "voy agregando poco a poco a una nueva lista para que al final solo muestre las que deseo "
        min = mpq.min(heap)
        dic = devolver_value(mapa,min)
        lt.addLast(respuesta,dic)

        mpq.delMin(heap)
        a +=1

    return respuesta
```

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

Descripción

Breve descripción de como abordaron la implementación del requerimiento

Para comenzar delimito la búsqueda al mes y año que me están pidiendo, de esa manera saca el primer día de mes a las primeras horas en forma que se pueda entender en el mi árbol, y después saco el siguiente mes, entonces al valor de mes le agrego uno (así se convierte el siguiente mes) a primera hora, de esa manera tengo un rango para poder utilizar el values. teniendo esto lo itero para ir en posición en posición calculándole la distancia con la fórmula de harvensine, y después de eso comparo si están en el rango que quiero buscar la información, si es así agrego el valor a un heap, y en un mapa le implemento como llave el valor (según la formula) y como valor toda la información del caso. Por último, hago una iteración de que se repita el número de actividades que quiero encontrar, ahí saco el min del heap, busco en el mapa según el valor que nos da min, agrego esa información a la lista y por último elimino el min para que se pueda repetir.

Entrada	Año y mes (donde se desea buscar, latitud y longitud (posición donde se quiere buscar), radio (el radio de cercanía) y actividades (cuantas actividades quiero imprimir).
Salidas	Lista de las actividades según el número que se busca en el radio que se está buscando
Implementado (Sí/No)	Si. Tomas Diaz

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Paso 1: convertir el año y mes dado en un rango, para poder seleccionar el año y mes deseado y ahí poder buscar	O (1)
Paso 2: lista_intervalo_valores = busca en la estructura de datos el rango del mes que le di	O(h+k)
Paso 3: tamanio = el tamaño de la lista dada	O (1)
Paso 4: heap = donde se van a poner los valores de distancia	O (1)
Paso 5: mapa = mapa que la llave va a ser la distancia y el valor va a ser los datos del accidente	O (1)
Paso 6: iterar toda la lista del rango seleccionado, para sacarle la distancia a cada uno del accidente y si está a la distancia que se está buscando se agrega al heap y al mapa	O(n), es una n más pequeña porque son solos los datos de un un año y mes especifico
Paso 7: insertar al heap los valores que cumplen con la distancia dada	O(logn)
Paso 8: respuesta = donde se van a acumular los datos que quiero dar	O (1)
Paso 9: itero el heap acorde cuantos datos se desea devolver	O (1)
Paso 10: saco el mínimo del heap	O (1)
Paso 11: busco ese valor en el mapa para obtener los datos del accidente	O (1)
Paso 12: lo agrego al final de la lista (para que al final quede en orden	O (1)

Paso 13: elimino el valor mínimo para que en la siguiente iteración allá otro mínimo	O(logn)
TOTAL	***O(n)***

Pruebas Realizadas

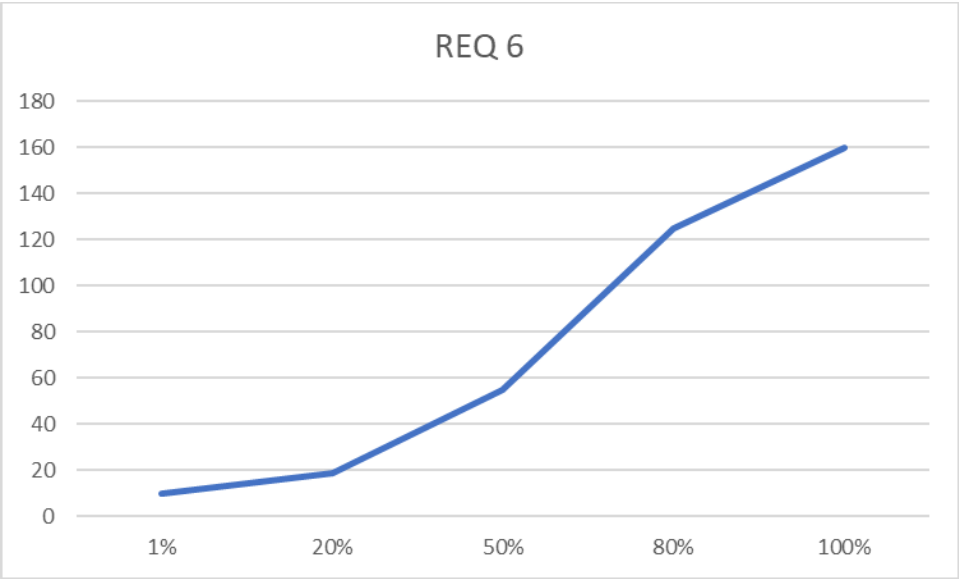
Descripción de las pruebas de tiempos de ejecución y memoria utilizada. Incluir descripción del procedimiento, las condiciones, las herramientas y recursos utilizados (librerías, computadores donde se ejecutan las pruebas, entre otros).

Los datos utilizados son lo de verificar en el reto, 3 accidentes más cercanos al punto con latitud 4.674, longitud -74.068 dentro de un radio de 5.0 km para el mes de enero de 2022.

Entrada	**Tiempo (s)**
1%	10.05
20%	18.7
50%	54.66
80%	124.77
100%	159.66

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.

En este requerimiento logre obtener una buena complejidad, porque a la hora de implementar árboles, se hace mucho más fácil la búsqueda entre un rango especifico de valores, esto me ayudo para encontrar el mes y año especifico donde quiero buscar la información. Adicionalmente, con los mapas y los ha, se hace rápido la búsqueda de valores mínimos, entonces no se tiene que iterar todos los datos y esto lo hace en log, esto me ayudo a tener organizada la información y lograr sacarle las distancias más cercanas a la longitud y latitud reportada.

Requerimiento 7_1

NOTA: El req 7 se dividió en 2 para que su proceso de programación fuera más eficiente. Nótese que la complejidad O(n) no se supera en ningún caso.

```
def req_7(data_structs, mes , anio ):
    """
    Función que soluciona el requerimiento 7
    """
    # TODO: Realizar el requerimiento 7
    respuesta = lt.newList()
    res_m = aux_mes(mes)
    dias = lt.getElement(res_m, 1)
    mes_res = lt.getElement(res_m , 2)
    i = 1
    while i <= dias:
        dia = str(i)
        dia_f = str(i)
        if i < 10:
            dia = str("0" + dia)
            dia_f = str("0" + dia_f)
        mes_d = str(mes_res)
        hora_i = str(anio + "/" + mes_d + "/" + dia)
        hora_f = str(anio + "/" + mes_d + "/" + dia_f)
        min_max = aux_mas_menos(data_structs , hora_i , hora_f)
        if min_max != None:
            menos = min_max[0]
            lt.addLast(respuesta , menos)
            mas = min_max[1]
            lt.addLast(respuesta , mas)
        i += 1
    return respuesta
```

Descripción

Breve descripción de como abordaron la implementación del requerimiento

En esta función recibimos un mes y un año, y buscamos el primer y último accidente de ese año/mes. Para ello se usa un árbol binario ordenado para conseguir los valores para cada dia, y se adjuntan a una lista de respuestas

Entrada	Año y mes
Salidas	Lista con el primer y último accidente de cada día de ese mes y ese año
Implementado (Sí/No)	Si, Manuel Pinzon

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Paso 1 : Usamos una función externa para conseguir la cantidad de días y el número del mes, para luego asignarlos a variables	O(1)
Paso 2: Se itera para cada día del mes, y se busca el primer y último accidente dentro de una franja horaria (solamente ese día). Con ayuda de una función externa hacemos esto, y damos como valor a esta variable una lista con el primer y último accidente de ese día	O(h+k)
Paso: 3 Se verifica si la respuesta de la lista del día tiene valores, para así no poner días sin accidentes en los resultados	O(1)
TOTAL	***O(h+k)***

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).

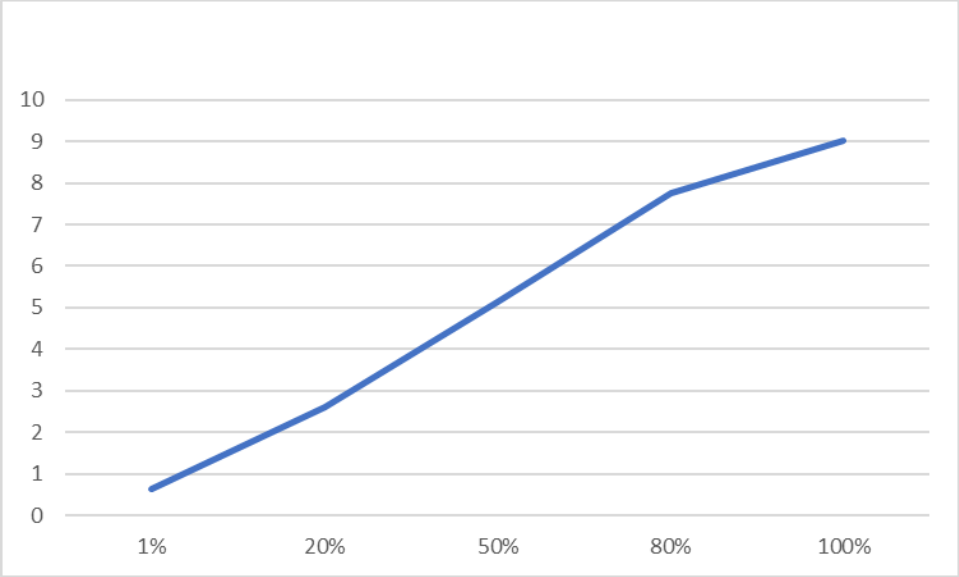
Entrada	**Tiempo (s)**
1%	0.639
20%	2.614
50%	5.14
80%	7.77
100%	90.2

Tablas de datos

Las tablas con la recopilación de datos de las pruebas.

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.

El uso de un árbol binario ordenado nos ayuda primero a ver los accidentes por día (con una franja de resultados seleccionados) y luego como están de forma ordenada podemos sacar el primer y último accidente de ese mismo día. El árbol hizo con que este requerimiento fuese más eficaz tanto en tiempo como complejidad

Requerimiento 7_2

```python
def data_frame_accidentes_por_hora(data_structs,anio,mes):

    arbol_fechas = data_structs['dateIndex']
    lim_inf = int(anio+mes)*10**10
    lim_sup = (int(anio+mes)+1)*10**10
    ##rango en single_linked
    rango_siniestros = om.values_array(arbol_fechas,lim_inf,lim_sup)

    #print(rango_siniestros)
    #crear dic
    dic_horas={}
    i =0
    while i<24:
        dic_horas[i]=0
        i+=1
    rango_siniestros_iterable = lt.iterator(rango_siniestros)
    for entry in rango_siniestros_iterable:
        lista_accidentes = entry['lista_accidentes']

        lista_acc_iterable = lt.iterator(lista_accidentes)

        for accidente in lista_acc_iterable:
            #print(accidente)
            hora_acc = int(accidente['HORA_OCURRENCIA_ACC'].split(':')[0])
            #print(hora_acc)

            dic_horas[hora_acc]+=1

    return dic_horas
```

Descripción

Esta parte del req 7 a partir de un año y mes dado, devuelve un diccionario cuyas llaves son las horas en enteros y su valor son las frecuencias de accidentes del mes en cada una de las dichas horas. A partir de este diccionario, en la vista se imprime el gráfico de barras correspondiente. Lo primero hace es obtener el rango de values de tipo array de forma exactamente igual a como se hizo en el req 5. Después se crea el diccionario de respuesta con llaves del 0 al 23 (las horas) y se le asigna a cada una 0 como valor. Luego, de manera similar al req 5, se itera accidente por accidente y se va tabulando el numero de accidentes por hora. Por último se retorna el dicho diccionario.

Entrada	Data Structs, año, mes
Salidas	Respuesta esperada del algoritmo.
Implementado (Sí/No)	Si, por Samuel Peña.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Obtener rango array	O(k+h)

Obtener diccionario vacío	O(1)
Iterar el rango (accidente por accidente) y tabular los accidentes por hora	O(n)
TOTAL	***O(n)***

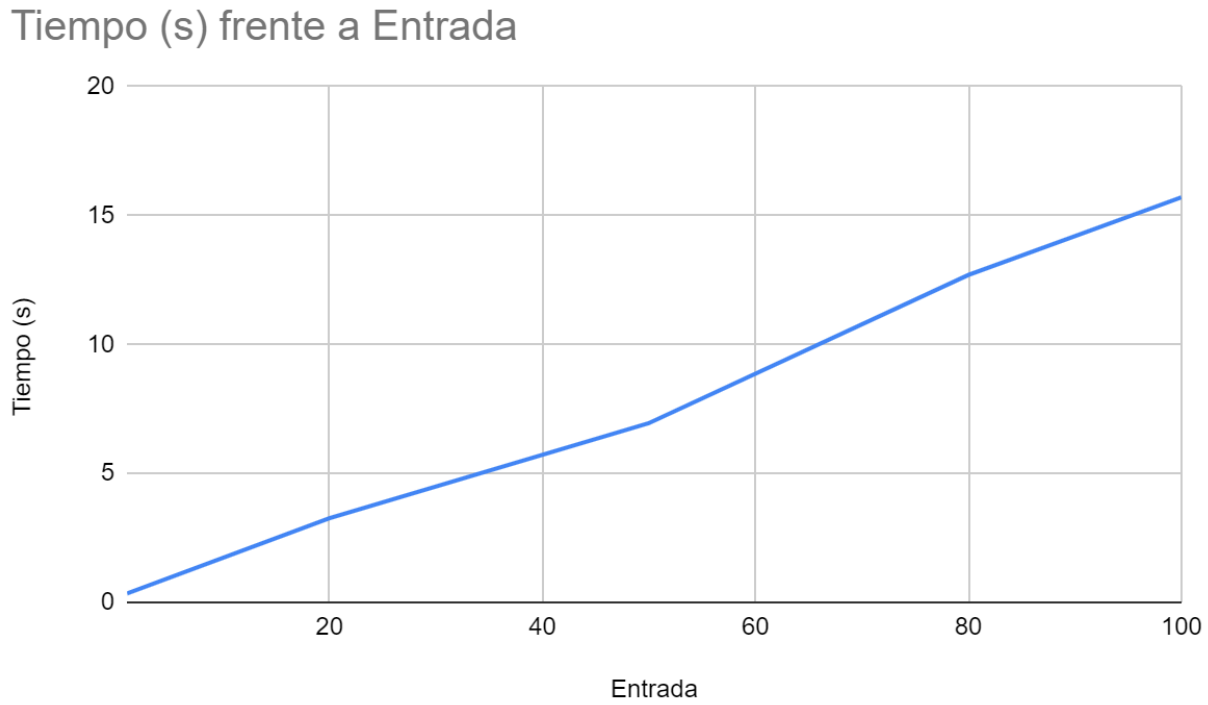
Pruebas Realizadas

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).

Entrada	**Tiempo (s)**
1%	0,34480000101029873
20%	3,263100001960993
50%	6,945200001820922
80%	12,693300001323223
100%	15,703999999910593

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.

Fue una buena implementación, los datos sustentan que fue del orden O(n). Al igual que en el req 5, la operación más compleja es iterar los accidentes en el rango.