DESCRIPCIÓN Y ESTIMACIÓN DE COMPLEJIDAD DE REQUERIMIENTOS  
  
  
  
  
  
  
  
  
  
  
  
Camilo Andrés Salinas Martínez.

201714930.  
  
Laura Isabella Forero Camacho.  
  
201716278

UNIVERSIDAD DE LOS ANDES  
FACULTAD DE INGENIERÍA  
CARRERA DE INGENIERÍA DE SISTEMAS  
ABRIL, BOGOTÁ D.C.  
2018

**Análisis de Requerimientos Funcionales**

|  |  |
| --- | --- |
| NOMBRE | 1A- Obtener el taxi que más servicios ha realizado iniciando en una zona dada, para una determinada compañía. |
| RESUMEN | Usando el Árbol Balanceado de compañías y la Tabla de Hash de taxis, obtener el taxi (o taxis) que más servicios ha realizado en una zona y para un compañía dada. En orden cronológico por su fecha de inicio. |
| ENTRADAS | |
| Area (Pickup\_community\_area)  Compañía | |
| RESULTADOS | |
| Linked List de Taxi (con el(los) ID del taxi resultante y la fecha/hora de inicio de cada uno de sus servicios (en orden cronológico)). | |
| COMPLEJIDAD (Método) | |
| O(N^2)  O(N)--------> Búsqueda en Hash Table.  (LOG2 N)--> Búsqueda en árbol rojo-negro.  O(N^2)-----> Operaciones con las listas que retornan las dos consultas. Selección para buscar el mayor.  --------------------------  O(LOG2 N + N+ N^2) ---> Se elije N^2 ya que es el mayor término en la operación.  Se estima una complejidad de O(N^2), ya que se suman, la búsqueda en una Hash Table O(N), en un árbol rojo negro O(log2 N) y las operaciones con la lista que retorna las dos consultas que son O(N^2). Luego de obtener del árbol la lista de taxis de la compañía, y la lista de servicios correspondientes al área de la Tabla. Por selección, se busca cada taxi en la lista de servicios del área, guardando en una variable el que mayor servicios presto en el área, es decir, por cada taxi de la compañía se pretende recorrer la lista del área comparando con los otros, en otras palabras, se utilizan dos ciclos para avanzar en las listas. | |

|  |  |
| --- | --- |
| NOMBRE | **2A**- Buscar conjunto de servicios a partir de un rango de consulta (duración). |
| RESUMEN | Agrupar los servicios por su duración (grupos de 60 segundos), crear una tabla hash con cada rango de duración y sus servicios. Obtener el conjunto de Servicios a partir de una duración de consulta (en segundos) usando la Tabla de Hash. |
| ENTRADAS | |
| Duración de consulta. | |
| RESULTADOS | |
| Linked List de servicios.  Por cada servicio se muestra:   * ID del taxi * ID del servicio * Duración en segundos. | |

|  |
| --- |
| COMPLEJIDAD (Método) |
| O(N)  Se estima esa complejidad ya que es la complejidad de búsqueda en un hash table. Como la tabla se crea al cargar en este método solo se consulta. |

|  |  |
| --- | --- |
| NOMBRE | **1B**- Buscar el conjunto de servicios de una distancia recorrida dada, definida en un rango. |
| RESUMEN | Agrupar los servicios por su distancia recorrida, y definir un árbol balanceado con los grupos de servicios. Obtener el conjunto de servicios por su rango (Distancia Mínima, Distancia Máxima) usando el Árbol Binario Ordenado Balanceado. |
| ENTRADAS | |
| Distancia (rango). | |
| RESULTADOS | |
| Linked List de servicios. | |

|  |
| --- |
| COMPLEJIDAD (Método) |
| O(log N)  Se estima una complejidad de O(log N) ya que el método consiste en buscar o consultar elementos en un árbol balanceado en este caso la implementación es un árbol balanceado Rojo Negro. Cuya complejidad en el peor caso es 2 log N. |

|  |  |
| --- | --- |
| NOMBRE | 2B- Buscar los servicios que iniciaron en una zona de recogida y terminan en una zona de terminación cuya hora de recogida esta en un rango de fechas dadas. |
| RESUMEN | Agrupar los servicios por su zona de recogida y de terminación, en una Tabla de Hash. Agrupar los servicios en una zona X y Y (inicio y terminación) dada estos se deben ordenar en un Árbol Balanceado cronológicamente por su día/hora de recogida. (Se realiza al momento de cargar). Buscar los servicios que iniciaron en una zona de recogida y terminan en una zona de terminación cuya hora de recogida está en un rango de fechas dadas. |
| ENTRADAS | |
| Zona de inicio.  Zona de terminación.  Fecha/hora inicial de consulta  Fecha/hora final de consulta | |
| RESULTADOS | |
| Lista de servicios. | |

|  |
| --- |
| COMPLEJIDAD (Método) |
| O(N)  Se estima esa complejidad ya que es la complejidad de búsqueda en un hash table. Como la tabla se crea al cargar en este método solo se consulta. |

|  |  |
| --- | --- |
| NOMBRE | **1C-** Ordenar los taxis registrados en el sistema, utilizando un sistema de puntos. |
| RESUMEN | Utilizar un heap sort para ordenar cada taxi, por el sistema de puntos. Los puntos de cada taxi se calculan así: se toma cada servicio prestado por el taxi, se suma el total de millas recorridas y el total de dinero recibido por dicho taxi en todos sus servicios y se divide el total de dinero recibido entre el total de millas recorridas, multiplicado por el total de servicios prestados. |
| ENTRADAS | |
| Ninguna. | |
| RESULTADOS | |
| Lista de Taxis ordenados por el sistema de puntos. | |

|  |
| --- |
| COMPLEJIDAD (Método) |
| O(N log N)  Se estima esta complejidad ya que es la complejidad para el algoritmo HeapSort. Este método obtiene los taxis los ordena y retorna la lista ordenada. |

|  |  |
| --- | --- |
| NOMBRE | **2C**- Consultar las localizaciones geográficas (latitud, longitud) de los servicios iniciados por un taxi (a partir de su Id) que están dentro de la zona a X millas de una localización geográfica dada. |
| RESUMEN | Se guarda en una Tabla Hash (Separate Chaining) todos los servicios registrados en el sistema dadas sus coordenadas (latitud, longitud). Estos servicios se representan en un Árbol Balanceado ordenados por el ID del taxi. Se desea consultar la información las localizaciones geográficas (latitud, longitud) de los servicios iniciados por un taxi (a partir de su Id) que están dentro de la zona a X millas de una localización geográfica dada. |
| ENTRADAS | |
| Localizaciones geográficas (latitud, longitud)  ID de taxi  Millas para calcular la distancia | |
| RESULTADOS | |
| Lista de Servicios. | |

|  |
| --- |
| COMPLEJIDAD (Método) |
| O(N)  O(N)--------> Búsqueda en Hash Table.  (LOG2 N)--> Búsqueda en árbol rojo-negro.  --------------------------  O(LOG2 N + N) ---> Se elije N ya que es el mayor término en la operación.  Se estima esa complejidad ya que es la complejidad de búsqueda en un árbol rojo-negro y en una tabla Hash Table. Como el árbol y la tabla se crea al cargar en este método solo se consulta. |

|  |  |
| --- | --- |
| NOMBRE | **3C-** Consultar todos los servicios registrados en el rango de tiempo, que adicionalmente hayan salido de una zona y hayan terminado en otra. |
| RESUMEN | De un árbol binario balanceado con los servicios, ordenados por fecha y hora (en rangos de 15 minutos). Dada una fecha y hora se debe retornar todos los servicios registrados en el rango, que hayan salido de una zona y hayan terminado en otra. |
| ENTRADAS | |
| Fecha y hora. | |
| RESULTADOS | |
| Lista de servicios. | |

|  |
| --- |
| COMPLEJIDAD (Método) |
| O(log2 N)  Se estima esa complejidad ya que es la complejidad de búsqueda en un árbol rojo-negro. Como el árbol se crea al cargar en este método solo se consulta. |

**Análisis de Estructuras de Datos**

Nombre: **Heap Binario**

Descripción:

Un Heap Binario es una estructura de datos que implementa las operaciones básicas de una cola de prioridad de una manera muy eficiente. Los datos son almacenados en una matriz, las claves están almacenadas en forma de árbol binario, de manera que la clave de una posición n (padre), es mayor a la clave de la posición 2n y 2n+1 (hijos). Sus dos operaciones básicas es borrar el máximo e insertar (skin y swim). En una cola de prioridad de *n* elementos, los algoritmos de pila no requieren más de 1 + Log Npara inserción y no más de 2 Log N para eliminar el máximo.

Ventajas:

Su estructura es muy eficiente en términos de espacio. Así mismo, cuenta con fácil acceso a los elementos.

Desventajas:

Es una estructura de capacidad estática, la cual trae el limitante de que cuando se llena por completo es necesario realocar el heap en un arreglo de mayor tamaño( Basado en el la implementación de Heap Binario sobre arreglo).

Casos de Uso:

* **1C-** Ordenar los taxis registrados en el sistema, utilizando un sistema de puntos.

Complejidad: Log N tanto en inserción como en eliminación

Justificación: Según los requerimientos funcionales, es necesario usar esta estructura de datos.

Nombre: **Hash Table**

Descripción: Es una estructura de datos que asocia llaves o claves con valores, funciona transformando la clave con una función hash en un hash, un número que identifica la posición donde la tabla hash localiza el valor deseado.

La función Hash, Si tenemos una matriz que puede contener pares de valores clave M, entonces necesitamos una función que puede transformar cualquier clave dada en un índice en que matriz: un número entero en el intervalo [0, M-1].

En general, existen varios tipos de implementación para el manejo de colisiones (cuando dos entradas distintas a una función de hash producen la misma salida), los más reconocidos son **Linear Probing** que cuando hay una colisión (cuando hash a un índice de tabla que ya está ocupada con una llave diferente de la clave de búsqueda), entonces solo comprobar la siguiente entrada en la tabla (por incrementando el índice). Hay tres posibles resultados, en caso de que sea la misma llave se reemplaza el valor. **Separate Chaining** que se implementa bajo un array, pero el elemento de cada posición es un nodo, en caso de colisión se añade como siguiente al nodo del índice correspondiente.

Ventajas: Mantienen la complejidad de búsqueda en O(1)

Desventajas: Las colisiones son casi inevitables con grandes cantidades de datos, se convierten ineficientes si hay muchas colisiones y un factor de carga poco adecuado para la cantidad de datos, no permiten valores en Null.

Casos de Uso:

* **2C**- Consultar las localizaciones geográficas (latitud, longitud) de los servicios iniciados por un taxi (a partir de su Id) que están dentro de la zona a X millas de una localización geográfica dada.
* **2B**- Buscar los servicios que iniciaron en una zona de recogida y terminan en una zona de terminación cuya hora de recogida esta en un rango de fechas dadas.
* **2A**- Buscar conjunto de servicios a partir de un rango de consulta (duración).
* **1A-** Obtener el taxi que más servicios ha realizado iniciando en una zona dada, para una determinada compañía.

Complejidad: O(1)

Justificación: Dados los requerimientos funcionales, se decidió usar la Hash Table.

Nombre: **Árbol Rojo Negro**

Descripción:

Un árbol rojo negro es un árbol binario balanceado donde cada nodo tiene también un atributo de color, cuyo valor puede ser o rojo o negro. Para que un árbol binario ordenado sea considerado como un árbol rojo negro debe cumplir con las siguientes propiedades:

* La raíz del árbol es negra.
* Los hijos de un nodo rojo son negros.
* Las hojas del árbol son negras.
* Todas las ramas del árbol (caminos desde la raíz hasta una hoja) tienen el mismo número de nodos negros.

Este tipo de estructuras sirven para el almacenamiento de elementos entre los que exista una relación de orden. La complejidad de la búsqueda en los árboles rojo negro es O(log2 n).

Ventajas: Conserva las ventajas de un árbol de búsqueda binario (BST), además es muy eficiente cuando se eliminan y añaden nodos constantemente, ya que se mantiene balanceado y asegura Log N en búsquedas.

Desventajas: Implementación complicada debido a todos los casos de excepción que se pueden dar

Casos de Uso:

* **3C-** Consultar todos los servicios registrados en el rango de tiempo, que adicionalmente hayan salido de una zona y hayan terminado en otra.
* **2C**- Consultar las localizaciones geográficas (latitud, longitud) de los servicios iniciados por un taxi (a partir de su Id) que están dentro de la zona a X millas de una localización geográfica dada.
* **1B**- Buscar el conjunto de servicios de una distancia recorrida dada, definida en un rango.
* **1A**- Obtener el taxi que más servicios ha realizado iniciando en una zona dada, para una determinada compañía.

Complejidad: Log N (en búsquedas)

Justificación: Según lo dicho en los requerimientos funcionales se nos pide utilizar un árbol binario, se decide usar un árbol Rojo Negro por su eficiencia en búsquedas y facilidad en su implementación.

Nombre: **Linked List**

Descripción: Estructura de datos lineal compuesta de nodos relacionados ya sea por un enlace doble o simple.

Ventajas:

* Eliminación de datos sencilla.
* Inserción de nodos sencilla.
* Estructura de fácil escalamiento, ya que la memoria usada para su implementación puede variarse mientras el programa corre.
* No es necesario definir un tamaño inicial.
* Puede ser usada en pilas y colas.

Desventajas:

* Difícil acceso aleatorio
* Ciertos algoritmos de ordenamiento y búsqueda se hacen complicados o imposibles de ejecutar (búsqueda binaria, Timsort).
* Ocupa más espacio en memoria que un arreglo, ya que tiene que guardar los apuntadores entre nodos (enlaces)

Casos de uso:

* **2C**- Consultar las localizaciones geográficas (latitud, longitud) de los servicios iniciados por un taxi (a partir de su Id) que están dentro de la zona a X millas de una localización geográfica dada.
* **2B**- Buscar los servicios que iniciaron en una zona de recogida y terminan en una zona de terminación cuya hora de recogida esta en un rango de fechas dadas.
* **2A**- Buscar conjunto de servicios a partir de un rango de consulta (duración).
* **1A-** Obtener el taxi que más servicios ha realizado iniciando en una zona dada, para una determinada compañía.

Complejidad:

Hemos elegido para búsqueda sobre esta estructura una búsqueda lineal.

* La cual debido a la estructura de datos el peor caso daría una complejidad O(N), ya que sería el caso en el que sea el último elemento en la lista, o un elemento no existente.

Para el ordenamiento elegimos MergeSort:

* Debido a que en su peor caso nos da una complejidad O(NLogN), debido al funcionamiento de este algoritmo, no existen casos de mejor o peor rendimiento.

Justificación:

Se siguió los lineamientos dados por los requerimientos entregados en el proyecto, esta fue la razón de uso de esta estructura lineal (Aun se esta en elección si se usará una lista sencilla o doble, Sin embargo se implementaron ambos casos).

Nombre: **Cola**

Descripción: Estructura de datos secuencial, conformada por nodos, en la cual los datos entran por un extremo y salen por el otro, permitiendo saber la duración relativa entre elementos (Ej: el ultimo elemento lleva menos tiempo dentro de la estructura que el primero)

Ventajas:

* Inserción de nodos sencilla.
* Estructura de fácil escalamiento, ya que la memoria usada para su implementación puede variarse mientras el programa corre.
* No es necesario definir un tamaño inicial.

Desventajas:

* Difícil acceso a elementos intermedios, imposible hacerse sin modificar la estructura.
* Es necesario rehacer la estructura para ordenarla, o ingresar los datos ordenados inicialmente.
* Ocupa más espacio en memoria que un arreglo, ya que tiene que guardar los apuntadores entre nodos (enlaces)

Casos de uso:

* **3C-** Consultar todos los servicios registrados en el rango de tiempo, que adicionalmente hayan salido de una zona y hayan terminado en otra.
* **2C**- Consultar las localizaciones geográficas (latitud, longitud) de los servicios iniciados por un taxi (a partir de su Id) que están dentro de la zona a X millas de una localización geográfica dada.
* **1B**- Buscar el conjunto de servicios de una distancia recorrida dada, definida en un rango.
* **1A**- Obtener el taxi que más servicios ha realizado iniciando en una zona dada, para una determinada compañía.

Complejidad:

* Debido a las dificultades de acceso y distintas características de la cola, usaremos una lista previa para el ordenamiento, y para la búsqueda se usara una búsqueda lineal que descarte elementos no necesitados.
  + Por lo tanto la búsqueda será de O(N).

Justificación:

Se siguió los lineamientos dados por los requerimientos entregados en el proyecto, esta fue la razón de uso de esta estructura lineal.

Nombre: **Pila**

Descripción: Estructura de datos secuencial, conformada por nodos, en la cual los datos entran y salen por un extremo, es decir, el último en entrar es el primero en salir. Sus métodos básicos son push (insertar) y pop(obtener el último insertado).

Ventajas:

* Ingreso de datos sencillo.
* estructura de almacenamiento eficiente.

Desventajas:

* Acceso aleatorio imposible sin modificar la estructura al azar.
* Ordenamientos y búsquedas con complejidad máxima.

Complejidad:

* Debido a las dificultades de acceso y distintas características de la cola, usaremos una lista previa para el ordenamiento, y para la búsqueda se usara una búsqueda lineal que descarte elementos no necesitados.
  + Por lo tanto la búsqueda será de O(N).

Justificación:

Se siguió los lineamientos dados por los requerimientos entregados en el proyecto, esta fue la razón de uso de esta estructura lineal.

**Análisis de Algoritmos de Ordenamiento**

Nombre: **Heap Sort**

Descripción: Algoritmo de ordenamiento no adaptativo y no estable que consiste en el uso de una cola de prioridad o heap, y ordena en base a esta, tiene dos fases. Primero una fase de construcción de un heap partir del conjunto de elementos de entrada, y después, una fase de extracción sucesiva de la cima del heap.

El algoritmo, después de cada extracción, recoloca en el nodo raíz o cima, la última hoja por la derecha del último nivel. Lo cual destruye la propiedad heap del árbol. Pero, a continuación realiza un proceso de "descenso" del número insertado de forma que se elige a cada movimiento el mayor de sus dos hijos, con el que se intercambia. Este intercambio, realizado sucesivamente "hunde" el nodo en el árbol restaurando la propiedad montículo del árbol y dejándo paso a la siguiente extracción del nodo raíz.

Ventajas: En el mejor de los casos es N Log N, en el peor de los casos N Log N, es muy bueno con datos desordenados.

Desventajas: Difícil implementación, no es adaptativo, no es estable.

Casos de Uso:

Requerimientos:

* **1C-** Ordenar los taxis registrados en el sistema, utilizando un sistema de puntos.

Complejidad: N Log N

Nombre: **MergeSort**

Es un algoritmo de ordenamiento externo que se basa en la combinación de dos matrices ordenadas y luego fusiona sus resultados.

Ventajas: Puede Ser usado con fuentes de datos de cualquier tamaño sin afectar su rendimiento

Desventajas: No es adaptativo.

Complejidad : O( NLogN).