Algoritmos y Estructuras de Datos - DC - UBA

Segundo cuatrimestre de 2024

Trabajo práctico 2: Diseño e implementación de estructuras

Normativa

Revisión de mitad de TP: viernes 08 de noviembre, en el horario de la clase.
Límite de entrega: domingo 17 de noviembre a las 23:59.
Límite de re-entrega: domingo 08 de diciembre a las 23:59.
Forma de entrega: Subir el archivo `BestEffort.java`, y todos los archivos de otras clases que hagan, dentro de un solo ZIP al campus.

Modificaciones

- 8/11: Consideramos que eliminar el último elemento de un `ArrayList` es $O(1)$.

Enunciado

La empresa de logística y distribución BestEffort[1] S.A. hace lo mejor posible para entregar todos los pedidos que recibe en tiempo y forma. Esta empresa opera en diversas ciudades de la provincia de Buenos Aires. Cada pedido define un traslado de productos de una ciudad a otra.

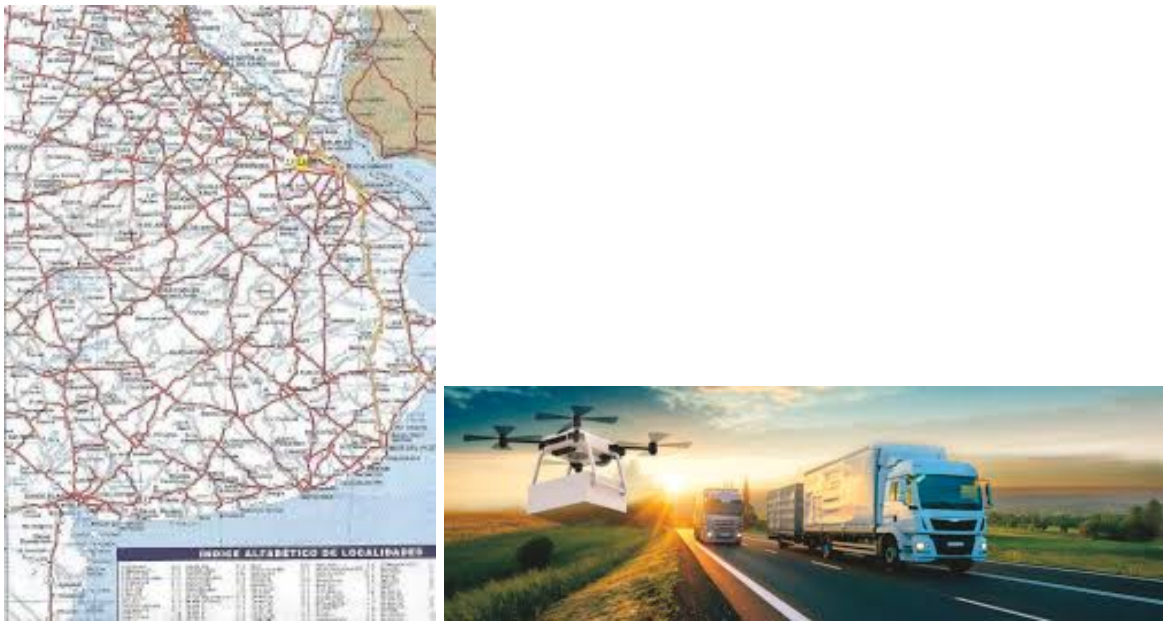

Figura 1: Rutas de Buenos Aires (izq.) (Fuente: https://www.unlu.edu.ar/mapas-lujan.html). Camiones y drones para la distribución de productos (der.) (Fuente: https://visionindustrial.com.mx/industria/operacion-industrial/logistica-y-distribucion-nacional-grandes-retos-ante-la-globalizacion)

La red de ciudades en las que opera BestEffort define un grafo completo[2], es decir: hay un camino directo entre cada par de ciudades. Además, esos caminos directos resultan ser los de menor costo y por lo tanto la elección natural para los traslados.

Se requiere diseñar e implementar un sistema que permita:

- Registrar traslados.

[1] https://en.wikipedia.org/wiki/Best-effort_delivery
[2] https://en.wikipedia.org/wiki/Complete_graph

- Despachar traslados: BestEffort adopta una política que permite regular dinámicamente el despacho de los traslados más redituables y aquellos que están pendientes de entrega hace más tiempo (esto impide lo que se conoce como *inanición* [3]).
- Recolectar estadísticas de las ciudades.

Aclaraciones

- Cada ciudad se identifica con un entero no negativos, son consecutivos y van de 0 a cantidadDeCiudades-1.
- Cada traslado:
  - Tiene un identificador único (entero no negativo)
  - Involucra exactamente a dos ciudades distintas: origen y destino.
  - Tiene una ganancia neta asociada (entero positivo).
  - Tiene un timestamp[4] (entero no negativo), que corresponde al momento en el que el cliente hizo el pedido. Son valores únicos.
- Dados dos traslados t_1 y t_2, decimos que t_1 es más redituable que t_2 si se cumple alguna de las siguientes condiciones:
  - $t_1.ganancia > t_2.ganancia$
  - $t_1.ganancia == t_2.ganancia \wedge t_1.id < t_2.id$
- Dados dos traslados t_1 y t_2, decimos que t_1 es más antiguo que t_2 si se cumple que $t_1.timestamp < t_2.timestamp$
- Las estadísticas aplican sobre los traslados despachados. Es decir, un traslado empieza a aportar información estadística a partir del momento en el que es seleccionado por el sistema para ser despachado.
- Las estadísticas deben devolver:
  - La/s ciudad/es con mayor ganancia. La *ganancia* de una ciudad es la suma de las ganancias de todos los traslados que se originaron en esa ciudad.
  - La/s ciudad/es con mayor pérdida. La *pérdida* de una ciudad es la suma de las ganancias de todos los traslados destinados a dicha ciudad.
  - La ciudad con mayor superávit: $ganancia - perdida$. En caso de empate, devuelve la que tiene menor identificador.
  - La ganancia promedio por traslado a nivel global.
- En caso de requerir un arreglo redimensionable, consideraremos que la complejidad agregar un elemento al final es considerada $O(1)$.
  - Más precisamente, es $O(1)$ amortizado.
  - Como la redimensión es poco frecuente, sólo sucede cuando se excede el tamaño actual...
  - Esto quiere decir que -si bien el momento de la copia induce un costo proporcional a la cantidad de elementos-, ese costo puede ser pensado como si estuviera promediado en el costo de cada inserción, cuya complejidad $O(1)$.
  - Considerar que es $O(1)$ en peor caso **es incorrecto**, pero en el contexto de este TP ignoraremos este problema y aceptaremos contarlo como O(1).
  - **No pueden asumir que esta simplificación valga en otras instancias de evaluación**.
  - Ver el Anexo para una explicación de la clase `ArrayList`, que implementa una secuencia usando un arreglo redimensionable.

La solución propuesta debe cumplir las restricciones que se imponen sobre la complejidad temporal de las operaciones, detalladas a continuación. Usamos las siguientes variables:

- C: conjunto de ciudades.
- T: conjunto de traslados.

La entrega debe incluir *al menos* una clase `Java` que implemente la solución al problema. Recomendamos (y valoraremos) modularizar la solución en varias clases.

IMPORTANTE: No se manden a programar sin pensar antes una solución al problema. La idea es que piensen "en papel" una estructura de representación que permita cumplir las complejidades y la consulten con su corrector durante la clase. Una vez que su corrector les de el OK, pueden comenzar a programar la solución.

[3] https://en.wikipedia.org/wiki/Starvation_(computer_science)
[4] https://en.wikipedia.org/wiki/Timestamp

Operaciones a implementar

Sea `InfoTraslado = <id: ℕ, origen: ℕ, destino: ℕ, gananciaNeta: ℕ, timestamp: ℕ>`

1. `nuevoSistema(in cantCiudades: ℕ, in traslados: seq<InfoTraslado>)`: BestEffort $O(|C| + |T|)$
   inicializa el sistema de la empresa BestEffort.
   Requiere $cantCiudades > 0$. Los traslados deben tener ids distintos entre si, los origenes y destinos están entre 0 y cantCiudades-1, los gananciaNeta y timestamp son positivos.
2. `registrarTraslados(inout sistema: BestEffort, in traslados: seq<InfoTraslado>)` $O(|traslados|\ log(|T|))$
   incorpora todos los traslados al sistema.
   Requiere que los traslados tengan id distinto entre si, y con respecto con los traslados ya registrados anteriormente.
3. `despacharMasRedituables(inout sistema: BestEffort, in n: ℕ): seq<ℕ>` $O(n\ (log(|T|) + log(|C|)))$
   devuelve los identificadores de los n traslados más redituables (y los elimina del sistema), ordenados de forma decreciente por ganancia. En caso de que n sea mayor a la cantidad de traslados por despachar, se despachan todos.
4. `despacharMasAntiguos(inout sistema: BestEffort, in n: ℕ): seq<ℕ>` $O(n\ (log(|T|) + log(|C|)))$
   devuelve los identificadores de los n traslados más antiguos (y los elimina del sistema), ordenados de forma creciente por timestamp. En caso de que n sea mayor a la cantidad de traslados por despachar, se despachan todos.
5. `ciudadConMayorSuperavit(in sistema: BestEffort): ℕ` $O(1)$
   devuelve el identificador de la ciudad con mayor superávit, en caso de empate devuelve la que tiene menor identificador.
6. `ciudadesConMayorGanancia(in sistema: BestEffort): seq<ℕ>` $O(1)$
   devuelve una secuencia con el identificador de la ciudad (o los identificadores de las ciudades, en caso de empate) con mayor ganancia.
7. `ciudadesConMayorPerdida(in sistema: BestEffort): seq<ℕ>` $O(1)$
   devuelve una secuencia con el identificador de la ciudad (o los identificadores de las ciudades, en caso de empate) con mayor pérdida.
8. `gananciaPromedioPorTraslado(in sistema: BestEffort): ℕ` $O(1)$
   devuelve la parte entera de la ganancia promedio por traslado a nivel global.
   Requiere que se haya despachado al menos un traslado.

Condiciones de entrega y aprobación

Durante la clase del día viernes 08 de noviembre deberán hablar con su corrector asignado para la **revisión obligatoria** de mitad de TP. En esta, deben mostrarle la estructura de representación propuesta para resolver el problema. Al finalizar esta clase, deben salir con una estructura que funcione para cumplir las complejidades pedidas.

La entrega deberá incluir el archivo `BestEffort.java`, el cual implementará la solución al problema. Las demás clases diseñadas para la solución se deben incluir en otros archivos `.java`. También los archivos de testeo que desarrollen. Todos estos archivos deben ser subidos al campus antes de la fecha y hora límite de entrega.

Para la aprobación del trabajo práctico, la implementación debe superar todos los tests provistos y, además, debe cumplir con las complejidades temporales especificadas en la sección anterior. Se debe dejar comentado (de forma breve y concisa) la justificación de la complejidad obtenida. Solamente pueden utilizar las estructuras vistas en la teórica hasta ahora (arreglo, vector, lista enlazada, ABB, AVL, heap, trie), implementadas por ustedes mismos. Pueden usar el código de las estructuras que hicieron para los talleres. **No se puede usar ninguna clase predefinida en la biblioteca estándar de Java, con la excepción de ArrayList, String y StringBuffer**. Si bien les proveemos una base de tests, **esperamos que agreguen sus propios tests**. También evaluaremos:

- la claridad del código y de las justificaciones de las complejidades.
- que se respeten los principios de abstracción y encapsulamiento.
- la modularidad del código: crear nuevas clases y funciones auxiliares cuando sea necesario.
- el testeo correcto de sus implementaciones.

Anexo: ArrayList de Java

`ArrayList` es la implementación que tiene Java del TAD secuencia sobre arreglo redimensionable/vector. Es necesario importarlo utilizando `import java.util.ArrayList;`. Dicho `import` debe estar al principio del archivo Java, luego del `package aed;`.

`ArrayList<T>` cuenta con las siguientes operaciones, que pueden resultar de utilidad:

- Constructor `ArrayList()`: construye una secuencia vacía. $O(1)$
- `boolean boolean add(T v)`: agrega el valor `v` al final de la secuencia. Devuelve siempre `true`. $O(1)$ [5]
- `T get(int index)`: obtiene el valor de la posición `index`. $O(1)$
- `T set(int index, T v)`: modifica el valor de la posición `index`. Devuelve el valor almacenado anteriormente. NO realiza una copia de `v`, por lo que genera aliasing. $O(1)$
- `T remove(int index)`: remueve el valor en la posición `index` y lo devuelve. $O(n)$. En caso de eliminar el último elemento, consideramos que la operación es $O(1)$
- `void clear()`: elimina todos los elementos de la secuencia. $O(1)$

[5] Si bien en peor caso es O(n), como explicamos anteriormente, en el contexto de este TP vamos a considerar que la operación es O(1).