

Algoritmos y Estructuras de Datos II

**PROYECTO INDIVIDUAL – PARTE I**

<LUIS, AYUB>

Contenido

[Introducción 3](#_Toc486861165)

[Problema planteado 3](#_Toc486861166)

[Análisis de alternativas 3](#_Toc486861167)

[Planteo de Soluciones 3](#_Toc486861168)

[Análisis de Memoria 3](#_Toc486861169)

[Diagrama de clase 5](#_Toc486861170)

[Algoritmos 6](#_Toc486861171)

[TElementoAVL 6](#_Toc486861172)

[TArbolAVL 10](#_Toc486861173)

[Ucupharma 11](#_Toc486861174)

[Selección y justificación de alternativa a implementar 14](#_Toc486861175)

[Conclusiones 16](#_Toc486861176)

[Guía del usuario 16](#_Toc486861177)

[Anexo 17](#_Toc486861178)

[Analisis de Memoria 17](#_Toc486861179)

# Introducción

En esta primera parte del proyecto se nos planteo el problema de crear un programa para la empresa AlasUcu, capaz de gestionar los vuelos que le sean utiles al cliente sabiendo el origen y su respectivo destino.

El programa se encargara de encontrar todos los vuelos, atravez de diferentes aerolíneas, que le sean utiles al cliente para llegar a su destino deseado, al mismo tiempo, se le indicara al cliente cual será el vuelo de menor costo.

## Problema planteado

La empresa AlasUcu ha solicitado un programa que sea capaz de encontrar todos los vuelos desde un origen hacia un destino, mostrando las aerolíneas disponibles y cantidad de escalas que se debe hacer para llegar al destino. Tambien se mostrara el vuelo de menor costo.

Además del programa solicitado, también se deberá considerar la cantidad de memoria que esta ocupa, y si se tiene más de una solución, se deberá elegir por la mejor solución.

# Análisis de alternativas

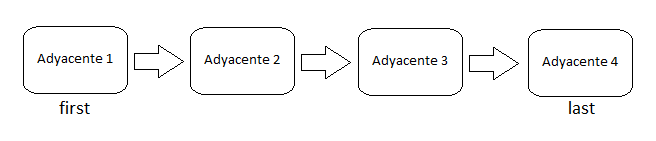
### Planteo de Soluciones

Para ambas selecciones la estructura de datos elegida es la de TArbolAVLes De Búsqueda Balanceados (AVL), habrá 2 árboles, un árbol para las categorías, donde cada categoría tendra un árbol de productos que pertenecen a dicha categoría. Teniendo en cuenta esto, pasamos a las soluciones sugeridas:

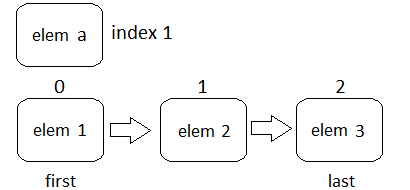
### Análisis de Memoria

(Ver Anexo)

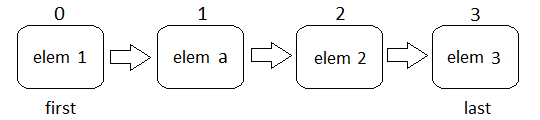
* ***Solución 1: Lista de Adyacencias (LinkedList):*** Esta solución permite manejar una lista de adyacencias de tamaño dinámico, contiene una gran cantidad de funciones a nuestra disposición, entre otros métodos útiles.



El *LinkedList* tiene una referencia del primer elemento (first) y del último elemento (last). Esto permite tanto agregar como eliminar el primer o el último elemento de la lista. Los elementos de una *LinkedList* contienen un índex, que permite realizar distintas acciones a un elemento en específico. Primero, para conseguir el índex de un elemento, se utiliza el método *indexOf(elemento)*, Una vez conseguido el índex del elemento, se pueden realizar varias acciones, entre ellas, eliminar en una posición determinada, o agregar en una posición determinada. En esta última, cuando se agrega, mueve hacia la derecha el elemento que se encontraba en esa posición, y al resto de elementos que se encontraban adelante.

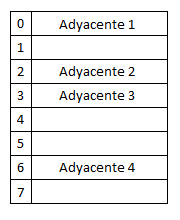


*Imagen 1: El elemento a va a ser agregado a la posición 1.*



*Imagen 2: El elemento a fue agregado, corriendo a los otros elementos hacia la derecha.*

* ***Solución 2: Matriz de Adyacencias (HashMap):*** Esta solución nos permite crear una matriz de tamaño estático, el cual a pesar de tener un tamaño definido, permite acceder a los elementos de manera mas rápida.

El HashMap es una estructura muy tentadora, ya que a la hora de obtener elementos, lo realiza de manera instantánea {O(1)}. El principal problema que conlleva utilizar esta estructura es que el tamaño debe ser estatico, ya que cada vez que se agrega un elemento a una matriz llena, esta se duplica para poder almacenar el nuevo elemento, lo cual aumenta considerablemente la memoria consumida por el programa.

### 

### 

### Diagrama de clase

### Algoritmos

### Administración

cargarDatos(String ruta1, String ruta2, String ruta3) : boolean **O(n)**

Comienzo

Devolver cargarAerolineas(ruta1) AND cargarAeropuertos(ruta2) AND cargarVuelos(ruta3) AND cargarGrafos()

Fin

boolean cargarAerolineas(String ruta) : boolean **O(n)**

Comienzo

Si (ruta <> "") Entonces

String[] datosAerolineas 🡨 ManejadorArchivosGenerico.leerArchivo(ruta, false)

Para Cada (String aero: datosAerolineas) Hacer

String[] datos 🡨 aero.split(",")

Aerolinea a 🡨 nuevo Aerolinea(datos[0],datos[1])

aerolineas.put(datos[0],a)

FinParaCada

Devolver true

FinSi

Devolver false

Fin

cargarAeropuertos(String ruta) : boolean **O(n)**

Comienzo

Si (ruta <> "") Entonces

String[] listaAeropuertos 🡨 ManejadorArchivosGenerico.leerArchivo(ruta, false)

Para Cada (String aero: datosAerolineas) Hacer

String[] datos 🡨 d.split(",")

TVertice v 🡨 nuevo TVertice(datos[0])

aeropuertos.add(v)

FinParaCada

Devolver true

FinSi

Devolver false

Fin

cargarVuelos(String ruta) : boolean **O(n)**

Comienzo

Si (ruta <> "") Entonces

String[] conexiones 🡨 ManejadorArchivosGenerico.leerArchivo(ruta, false)

Para Cada (String aero: datosAerolineas) Hacer

String[] datos 🡨 d.split(",")

Si (aerolineas contiene datos[0]) Entonces

TArista vuelo 🡨 nuevo TArista(datos[1], datos[2], datos[3])

aerolineas.get(datos[0]).vuelos.add(vuelo)

FinSi

FinParaCada

Devolver true

FinSi

Devolver false

Fin

cargarGrafos(): boolean **O(n)**

Comienzo

Si (aerolineas <> null){

Para Cada (Aerolinea a : aerolineas.valores) Hacer

a.setGrafo(aeropuertos, a.vuelos)

FinParaCada

Devolver true

FinSi

Devolver false

Fin

obtenerCaminos(Comparable origen, Comparable destino) : TVuelos **O(?)**

Comienzo

Si (origen <> "") AND (destino <> "") Entonces

TVuelos caminosTotal <-- nuevo TVuelos()

Para Cada (Aerolinea a: aerolineas.valores)) Hacer

TVuelos vuelos <-- a.grafo.todosLosCaminos(origen, destino, a.nombre)

caminosTotal.caminos.addAll(vuelos.caminos)

FinParaCada

Si (caminosTotal.caminos.tamaño > 0) Entonces

Devolver caminosTotal

Sino

MostrarMensaje "No hay vuelos disponibles entre " + origen + " y " + destino + "."

FinSi

Sino

MostrarMensaje "Uno de los atributos se encuentra vacio."

FinSi

Devolver null

Fin

imprimirVuelos(TVuelos vuelosTotal) : String **O(n)**

Comienzo

Si (vuelosTotal <> null) Entonces

StringBuilder sb <-- nuevo StringBuilder()

Para Cada (TVuelo c: vuelosTotal.caminos) Hacer

sb.append(c.imprimirEtiquetas()+"\n")

FinParaCada

sb.append(" ------------------------ "+"\n")

sb.append("El vuelo de menor costo es: "+"\n")

sb.append(vuelosTotal.obtenerMenor().imprimirEtiquetas()+"\n")

Devolver sb.toString()

FinSi

Devolver ""

Fin

### TGrafoDirigido

todosLosCaminos(Comparable etiquetaOrigen, Comparable etiquetaDestino, String aerolinea) : TVuelos **O(?)**

Comienzo

TVuelos todosLosCaminos <-- nuevo TVuelos()

TVertice v <-- buscarVertice(etiquetaOrigen)

Si (v <> nulo) Entonces

TVuelo caminoPrevio <-- nuevo TVuelo(v,aerolinea)

v.todosLosCaminos(etiquetaDestino, caminoPrevio, todosLosCaminos)

Devolver todosLosCaminos

FinSi

Devolver null

Fin

### TVertice

todosLosCaminos(Comparable etVertDest, TVuelo caminoPrevio, TVuelos todosLosCaminos) : TVuelos **O(?)**

Comienzo

visitado <-- true

Para Cada (TAdyacencia adyacencia : this.adyacentes) Hacer

TVertice destino <-- adyacencia.getDestino()

Si (!destino.visitado) Entonces

Si (destino.etiqueta = etVertDest) Entonces

TVuelo copia <-- caminoPrevio.copiar()

copia.agregarAdyacencia(adyacencia)

todosLosCaminos.caminos.add(copia)

Sino

caminoPrevio.agregarAdyacencia(adyacencia)

destino.todosLosCaminos(etVertDest, caminoPrevio, todosLosCaminos)

caminoPrevio.eliminarAdyacencia(adyacencia)

FinSi

FinSi

FinParaCada

visitado <-- false

Devolver todosLosCaminos

Fin

### TVuelo

imprimirEtiquetas() : String **O(n)**

Comienzo

int escalas <-- otrosVertices.size() - 1

StringBuilder sb = nuevo StringBuilder()

sb.append("Aerolinea: " + getAerolinea())

sb.append(" // " + getOrigen().etiqueta)

Para Cada (Comparable adyacente : otrosVertices) Hacer

sb.append(" -> " + adyacente)

FinParaCada

sb.append(" // Costo: " + costoTotal)

Si (escalas > 0) Entonces

sb.append(" // Escalas: " + escalas)

Sino

sb.append(" // Directo")

FinSi

Devolver sb.toString()

Fin

agregarAdyacencia(TAdyacencia adyacenciaActual) : boolean **O(1)**

Comienzo

Si (adyacenciaActual.destino) <> nulo) Entonces

costoTotal 🡨 (costoTotal + adyacenciaActual.costo)

Devolver otrosVertices.add(adyacenciaActual.destino.etiqueta)

FinSi

Devolver false

Fin

eliminarAdyacencia(TAdyacencia adyacenciaActual) : boolean **O(1)**

Comienzo

Si (adyacenciaActual.destino) <> nulo) Entonces

costoTotal 🡨 (costoTotal - adyacenciaActual.costo)

Devolver otrosVertices.remove(adyacenciaActual.destino.etiqueta)

FinSi

Devolver false

Fin

copiar() : TVuelo **O(1)**

Comienzo

TVertice origen 🡨 nuevo TVertice(this.origen.etiqueta)

TVuelo copia 🡨 nuevo TVuelo(origen, this.aerolinea)

copia.costoTotal 🡨 costoTotal

origen.adyacentes.addAll(this.origen.adyacentes)

copia.otrosVertices.addAll(otrosVertices)

Devolver copia

Fin

### TVuelos

obtenerMenor() : TVuelo **O(n)**

Comienzo

TVuelo aux = vuelos.iterator().next()

double min = aux.getCostoTotal()

Para Cada (TVuelo c: vuelos) Hacer

Si (min > c.costoTotal) Entonces

min = c.costoTotal

aux = c

FinSi

FinParaCada

Si (aux <> nulo) Entonces

Devolver aux

FinSi

Devolver nulo

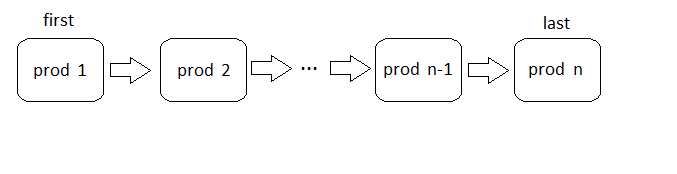
Fin

# Selección y justificación de alternativa a implementar

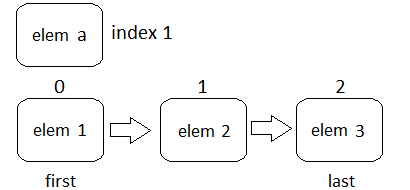
De las 2 soluciones presentadas, en mi opinión la más apropiada es la solución 1, ya que permite manejar la lista de una manera más dinámica, sin importar el tamaño, y se puede agregar e eliminar adyacencias sin mayores complicaciones.

El problema de la solución 1 es que a pesar de que sea simple agregar, quitar elementos y buscar elementos, lo limitante es el tamaño estático, a pesar de que este tamaño puede modificarse, no es recomendable, ya que cada vez que se agrega un elemento a una matriz llena, esta se duplica para poder almacenar el nuevo elemento, lo cual aumenta considerablemente la memoria consumida por el programa.

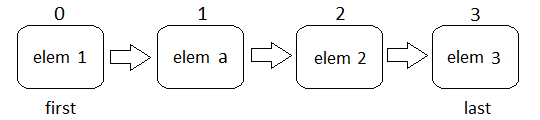
***Lista de Adyacencias (LinkedList):***



Una *LinkedList* es una clase proporcionada por java, la cual implementa otra clase llamada *List*, la cual también ya se encuentra en java mismo. El *LinkedList* tiene una referencia del primer elemento y del último elemento. Esto permite tanto agregar como eliminar el primer o el último elemento de la lista. Los elementos de una *LinkedList* contienen un índex, que permite realizar distintas acciones a un elemento en específico. Se pueden realizar varias acciones, entre ellas, eliminar en una posición determinada, o agregar en una posición determinada. En esta última, cuando se agrega, mueve hacia la derecha el elemento que se encontraba en esa posición, y al resto de elementos que se encontraban adelante.



*Imagen 1: El elemento a va a ser agregado a la posición 1.*



*Imagen 2: El elemento a fue agregado, corriendo a los otros elementos hacia la derecha.*

Pasaremos a explicar las distintas clases que contendría el programa utilizando listas encadenadas simples:

* **Clase Administracion:**

Contendrá las funcionalidades para el manejo de gestión que fueron solicitadas

* **Interfaz IGrafoDirigido:**

Contendrá las funcionalidades esenciales de un TGrafoDirigido, como eliminarArista(), eliminarVertice(), insertarVertice(), insertarArista(), y fundamentalmente todosLosCaminos().

* **Clase TGrafoDirigido:**

Implementa IGrafoDirifido, por lo que contendrá las funcionalidades esenciales de un grafo dirigido, contendrá una lista de TVertices y otra de TAristas.

* **Clase Aerolinea:**

Contendrá la información de las aerolíneas, como su nombre, la etiqueta, y una lista de TAristas, donde se guardaran los vuelos realizados por esta aerolínea.

* **Interface IVertice:**

Contendrá las funcionalidades esenciales de una vértice, como insertarAdyacencia(), buscarAdyacencia(), eliminarAdyacencia(), y fundamentalmente todosLosCaminos().

* **Clase TVertice:**

Implementa IVertice, por lo que tendrá las funcionalidades de un vértice de un grafo dirigido, contiene una lista de adyacencias con la cual se puede saber que otros vertices están relacionados con el.

* **Interface IArista:**

Contendrá las funcionalidades esenciales de una arista, como los getters y setters de las variables etiquetaOrigen, costo, y etiquetaDestino.

* **Clase TArista:**

Implementa IArista, por lo que tendrá las funcionalidades de una arista de un grafo dirigido, contiene las etiquetas del vértice de origen y del destino, y su costo.

* **Interface IAdyacencia:**

Contendrá las funcionalidades esenciales de una adyacencia, como los getters de las variables etiqueta, costo, y destino.

* **Clase TAdyacencia:**

Implementa IAdyacencia, por lo que tendrá las funcionalidades de una adyacencia de un grafo dirigido, contiene el vértice destino, su etiqueta, y su costo.

* **Clase TVuelo:**

Contendrá los vuelos necesarios para ir desde un origen hasta un destino determinado, también tendrá el costo de dicho camino.

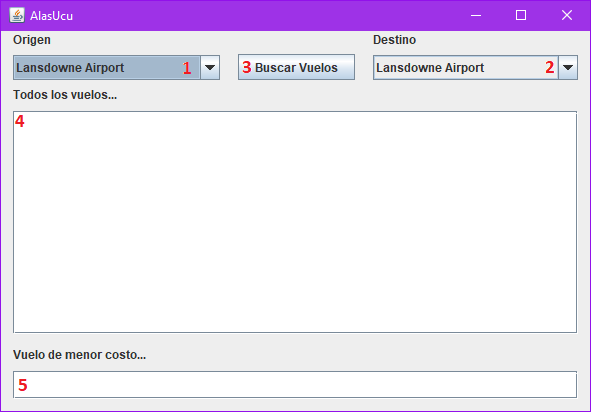
* **Clase TVuelos:**

Contendrá una lista de TVuelo, con todos los caminos diferentes que van desde un origen predeterminado hacia un destino predeterminado, también puede obtener el TVuelo de menor costo.

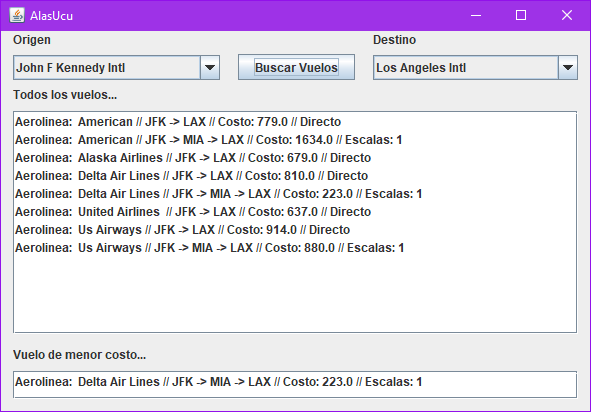
# Conclusiones

En Resumen, En mi opinión en uso de lista de adyacencias es la mejor solución para este problema, ya que permite crear una lista dinámica con menos limitaciones, y más personificación. En el caso de que los datos aumentaran en el futuro, sería más sencillo agregarlos a la lista de adyacencia de su respectivo vértice, que crear todo desde cero.

# Guía del usuario



* **1** : Lista de aeropuertos que se tomaran como origen del vuelo
* **2** : Lista de aeropuertos que se tomaran como destino del vuelo
* **3** : Presione el botón (Buscar Vuelos) para que el programa comience la búsqueda de vuelos que tengan como origen y destino los aeropuertos seleccionados en **1)** y **2)**
* **4** : Aquí aparecerán todos los vuelos que tengan como origen el aeropuerto seleccionado en **1)** , y como destino el aeropuerto seleccionado en **2)**.
* **5** : Aquí aparecerá el vuelo con menor costo de los vuelos encontrados.
* A continuación se mostrara un ejemplo del programa, utilizando el aeropuerto “John F Kennedy Intl” como origen y “Los Angeles Intl” como destino.



# Anexo

### Análisis de Memoria

**Clase Aerolínea:**

|  |  |  |
| --- | --- | --- |
| Nombre de la Variable | Tipo de variable | Costo en bytes |
| vuelos | Collection<TArista> | (Tamaño de la colección) \* (Tamaño de TArista) |
| grafo | TGrafoDirigido | 32 \* (Tamaño del HashMap + 4) \* (Capacidad del Hashmap) |
| etiqueta | Comparable | 8 |
| nombre | String | 8 |
| Total |  | {(Tamaño de la colección) \* (Tamaño de TArista)} + {32 \* (Tamaño del HashMap + 4) \* (Capacidad del Hashmap)} + 16 |

**Clase Administracion:**

|  |  |  |
| --- | --- | --- |
| Nombre de la Variable | Tipo de variable | Costo en bytes |
| aerolineas | Map<Comparable,Aerolinea> | 32 \* (Tamaño del HashMap + 4) \* (Capacidad del Hashmap) |
| aeropuertos | Map<Comparable,TVertice> | 32 \* (Tamaño del HashMap + 4) \* (Capacidad del Hashmap) |
| Total |  | {32 \* (Tamaño del HashMap + 4) \* (Capacidad del Hashmap)} \* 2 |