**UT3 – PD1**

**Ejercicios #1-5 Lista encadenada**

Los nodos de una lista simplemente encadenada tienen dos atributos:

• DATOS, de tipo “dato”.

• SIGUIENTE, de tipo “nodo de lista”, que hace referencia al nodo siguiente en la lista.

**Ejercicio #1**

Sean nodo1, nodo2 y nodo3 tres nodos consecutivos de una lista (nodo2 es el siguiente a nodo1 y nodo3 es el siguiente a nodo2). Analice el siguiente fragmento de código (utilice dibujos o diagramas para clarificar qué es lo que sucede):

nodo1 -> nodo2 -> nodo3 -> null

**Nuevo nodo otroNodo**

**otroNodo.siguiente <- nodo1**

**nodo2.siguiente <- nodo3**

***a) Inserta “otroNodo” en la lista, quedando como anterior a nodo1.***

**R: En este fragmento estamos estableciendo a nodo1 como el nodo siguiente a otroNodo, entonces se colocaría al inicio de la lista, para que pueda tener a nodo1 como siguiente.**

**Visualmente quedaría:**

**otroNodo -> nodo1 -> nodo2 -> nodo3 -> null**

**Esto es lo que hace el código propuesto. Las siguientes opciones proponen otras cosas que se escriben debajo.**

*b) Inserta “otroNodo” en la lista, quedando entre nodo1 y nodo2.*

**R: otroNodo.siguiente <- nodo2**

**nodo1.siguiente <- otroNodo**

**Este fragmento sería lo que debería estar escrito para que otroNodo quede entre nodo1 y nodo2, es decir, debe ser el siguiente a nodo1 y nodo2 debe tenerlo como siguiente (el siguiente a nodo2 es otroNodo)**

**nodo1 -> otroNodo -> nodo2 -> nodo3 -> null**

*c) Elimina nodo2 de la lista.*

**R: Esto se debe hacer quitando las referencias a nodo2.**

**nodo1.siguiente <- nodo3**

**Alli hacemos que el siguiente a nodo1 sea nodo3 y ya no referenciamos a nodo2 y lo eliminamos de la lista.**

**nodo1 -> nodo3 -> null**

**Ejercicio #2**

Sean nodo1, nodo2 y nodo3 tres nodos consecutivos de una lista (nodo2 es el siguiente a nodo1 y nodo3 es el siguiente a nodo2). Analice el siguiente fragmento de código (utilice dibujos o diagramas para clarificar qué es lo que sucede):

nodo1 -> nodo2 -> nodo3 -> null

Nuevo nodo otroNodo

otroNodo <- nodo1.siguiente

nodo1.siguiente <- nodo3

a*) Inserta “otroNodo” en la lista, quedando como anterior a nodo1.*

*b) Inserta “otroNodo” en la lista, quedando entre nodo1 y nodo2.*

**c) Elimina nodo2 de la lista.**

**R: Esta sería la opción que trata el código brindado. En este caso, otroNodo es el siguiente a nodo1, y luego dice que nodo3 es el siguiente a nodo1. Entonces se eliminaría a nodo2 de la lista y otroNodo no se inserta, porque se termina eliminando al establecer que nodo3 es siguiente a nodo1.**

*d) No tiene ningún efecto sobre la lista.*

**Ejercicio #3**

Sean nodo1, nodo2 y nodo3 tres nodos consecutivos de una lista (nodo2 es el siguiente a nodo1 y nodo3 es el siguiente a nodo2).

nodo1 -> nodo2 -> nodo3 -> null

Analice el siguiente fragmento de código (utilice dibujos o diagramas para clarificar qué es lo que sucede) y responda las preguntas proyectadas en pantalla:

Nuevo nodo otroNodo

otroNodo.siguiente <- nodo1.siguiente

nodo1.siguiente <- otroNodo

nodo1 -> otroNodo -> nodo2 -> nodo3 -> null

a) Inserta “otroNodo” en la lista, quedando como anterior a nodo1.

**b) Inserta “otroNodo” en la lista, quedando entre nodo1 y nodo2.**

**R: La opción adecuada sería esta ya que se define al siguiente a otroNodo como el siguiente a nodo1, y luego se establece que el siguiente a nodo1 es otroNodo. Entonces otroNodo queda apuntando al nodo2, y nodo1 apunta a otroNodo. Se inserta en el medio de ambos.**

c) Elimina nodo2 de la lista.

d) Dará error en tiempo de ejecución si nodo1 es el primero o nodo3 es el último.

**Ejercicio #4**

Analice el siguiente fragmento de código (utilice dibujos o diagramas para clarificar qué es loque sucede) y responda las preguntas proyectadas en pantalla:

Nuevo nodo otroNodo

Nuevo nodo nodoActual

nodoActual <- primero

mientras nodoActual <> nulo hacer

nodoActual <- nodoActual.siguiente

fin mientras

nodoActual.siguiente <- otroNodo

a) Inserta correctamente “otroNodo” en la lista, quedando como último nodo.

b) Inserta correctamente “otroNodo” en la lista, quedando como primer nodo.

**c) El algoritmo está mal hecho, ya que dará error en tiempo de ejecución si la lista está vacía.**

**R: Esta es la respuesta correcta por varias razones. Si primero no existe (la lista esta vacía) no entraría al while. Además, este while se termina cuando nodoActual es nulo. Luego de que nodoActual sea nulo, hace que el siguiente al mismo sea otroNodo, y al ser nodoActual un null, no es correcto tener un null.siguiente, porque null no tiene siguiente.**

d) El algoritmo está mal hecho, ya que dará siempre error en tiempo de ejecución.

**Ejercicio #5**

Analice el siguiente fragmento de código (utilice dibujos o diagramas para clarificar qué es loque sucede) y responda las preguntas proyectadas en pantalla:

Nuevo nodo otroNodo

Nuevo nodo nodoActual

nodoActual <- primero

mientras nodoActual.siguiente <> nulo hacer

nodoActual <- nodoActual.siguiente

fin mientras

nodoActual.siguiente <- otroNodo

**a) Inserta correctamente “otroNodo” en la lista, quedando como último nodo.**

**R: Esta es la opción correcta (teniendo en cuenta que primero existe) ya que el while se va a ejecutar hasta que se llegue al último nodo de la lista. Cuando esto ocurra, se asigna al siguiente nodo de nodoActual, que sería el último en la lista actualmente, a otroNodo.**

**Ejercicio #6**

Escenario:

Se desea llevar un registro de asistencia de un cierto curso universitario, el cual contará con una cantidad no determinada inicialmente de alumnos. Para ello, se ha decidido utilizar una lista para representar los alumnos en este curso. Cada elemento de la lista entonces tendrá un identificador del alumno y un campo que se ha de incrementar cada vez que el alumno concurra a una clase.

También se desea registrar el total de clases impartidas en el curso, y con este dato luego para cada alumno obtener el porcentaje de asistencia a las clases. Las listas pueden implementarse físicamente de dos formas básicas: utilizando un array, o armando una lista encadenada. Se desea la opinión experta de tu Equipo para determinar qué utilizar para resolver eficientemente el problema planteado.

1. ¿Cuál es el costo de memoria en cada caso?

**Usando array el costo de memoria usado sería de 16 bytes + (cantidad\_de\_elementos \* memoria\_que\_ocupa\_un\_producto), la memoria que ocupa un producto es de 4 u 8 bytes, dependiendo del procesador utilizado, en nuestro caso al tenes procesadores de 64bits sería 8, pero por una implementación de Java se usan 4. Usando listas encadenadas el costo de memoria usado sería de 4+(4\*nodos)+4 Iniciar la referencia a la lista cuesta 4bytes, cada nodo ocupa 4 byes y los ultimos 4 bytes que se suman es porque después del último nodo tiene que haber una conexión al siguiente que vale "null".**

**Costo de memoria de arrays:**

**Cabezal del objeto + largo + cantidad elementos \* memoria producto**

**12 + 4 + (n \* 4)**

**16+(n4)**

**Costo de memoria (listas encadenadas)**

**Cabezal de lista + Cabezal del primer nodo + ((Cabezal del nodo + Nodo + referencia al nodo siguiente) cantidad de nodos)**

**12 + 4 + (( 12 + 4 + 4 )n**

**16+(20n)**

b) ¿Cuáles son las consideraciones que tu Equipo haría referentes a la cantidad de

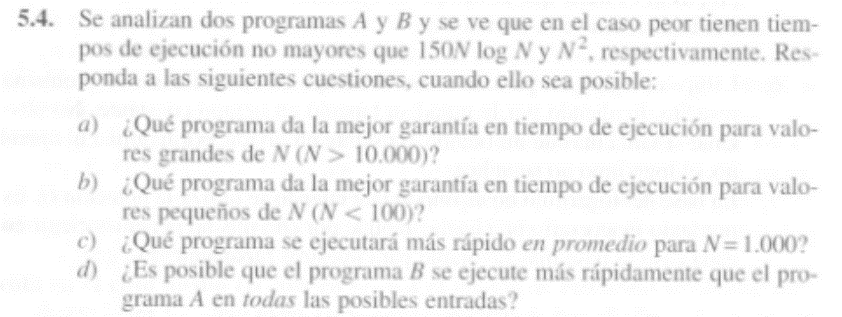
alumnos del curso que soporta cada tipo de estructura? (puedes considerar que, como

en la UCU, las inscripciones al curso suelen estar habilitadas desde varias semanas

antes de empezar el curso hasta dos semanas después de haber comenzado)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**UT2 – PD3**

****

1. En este caso el programa A garantiza un mejor tiempo de ejecución ya que según la gráfica, dentro de los peores casos, O(N^2) es peor que O(N log N). Por eso, cuando N sea mayor a 10.000, el exponencial será menos eficiente.

Gráfico, Gráfico circular

Descripción generada automáticamente

1. Ocurre lo mismo que en el caso A.
2. Es mucho mejor el algoritmo A en este caso, ya que si hacemos cuentas, O(N log N) con N = 1000 nos da 3000, y el otro caso daría 10.000.000.
3. No, ya que un algoritmo cuya complejidad es exponencial va a ser peor que el de una de orden N log N.

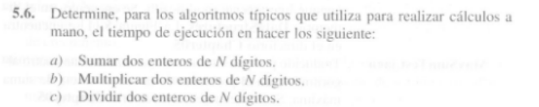
Texto

Descripción generada automáticamente

Texto, Carta

Descripción generada automáticamente

1. **Si se pone inicio < fin, entonces no llegará hasta el último elemento del array.**
2. **Si se pone inicio+fin/2 entonces en lugar de sumar primero inicio más fin, dividirá fin entre 2 y luego le sumará inicio, esto afectará en el resultado y el mismo no será correcto.**
3. **Si no se le suma 1 a medio en inicio = medio, entonces afectará en la búsqueda, incluyendo uno de los elementos que no irían en la otra mitad.**
4. **Sería igual a la parte c, ya que al no restar 1 al final estaríamos tomando un elemento de la mitad descartada que no necesitamos.**



1. **Sería de O(N) porque sería una suma por cada dígito, y como los dígitos son N, esto se repite la cantidad de dígitos que tengan los dos números.**
2. **Es de O(N^2) porque cada dígito del primer número se multiplica por todos los dígitos del segundo número.**
3. **En este caso también sería O(N^2) porque para dividir de forma tradicional primero separamos en subdivisiones el número a dividir y se multiplica, y luego se resta para obtener el resto.**

Patrón de fondo

Descripción generada automáticamente



a) **Lineal: Si el algoritmo es lineal O(N), entonces tardará 5 x 0.5 milisegundos = 2.5 milisegundos.**

**b) O(N log N): Si el algoritmo es O(N log N), entonces tardará 10 x 0.5 milisegundos = 5 milisegundos.**

**c) Cuadrático: Si el algoritmo es O(N^2), tardará 25 veces más para una entrada de tamaño 500, es decir, 25 x 0.5 milisegundos = 12.5 milisegundos.**

**d) Cúbico: Si el algoritmo es O(N^3), entonces tardará 125 veces más para una entrada de tamaño 500, 125 x 0.5 milisegundos = 62.5 milisegundos.**

Patrón de fondo

Descripción generada automáticamente

**Minuto: 60.000ms**

1. **Regla de 3:**

**Tamaño de entrada = (tiempoEjecucion / tiempoEntrada) \* 100**

**Tamaño de entrada = (60.000ms / 0.5ms) \* 100**

**El tamaño de entrada que puede procesar es 12.000.000.**

1. **Regla de 3:**

**Tamaño de entrada = (tiempoEjecucion / tiempoEntrada^2) \* 100**

**Tamaño de entrada = (60.000ms / 0.5ms^2) \* 100**

**El tamaño de entrada que puede procesar es 24.000.000.**

1. **Regla de 3:**

**Tamaño de entrada = (tiempoEjecucion / tiempoEntrada) \* 100**

**Tamaño de entrada = (60.000ms / 0.5^3) \* 100**

**El tamaño de entrada que puede procesar es 48.000.000**



* **37, 2/N, RaizN, N log log N, N, N log N, N log N^2, N log^2 N, N^1,5, N^2 log N, N^2, N^3, 2^N/2, 2^N**
* **N log N y N log N^2 crecen a la misma velocidad porque n log n^2 solo cambia por el factor constante de 2.**
* **2^N/2 y 2^N crecen a la misma velocidad porque solo tienen distinto un factor constante de 2.**

**Imagen que contiene Gráfico

Descripción generada automáticamente**



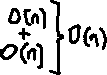
Imagen de la pantalla de un celular de un mensaje en letras negras

Descripción generada automáticamente con confianza bajaTexto

Descripción generada automáticamente



|



Texto

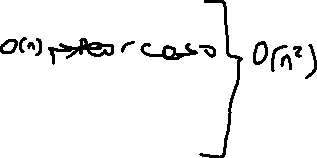
Descripción generada automáticamente

1. **La multa el día N sigue la fórmula 2^((N-1)\*2)**
2. **O(n log n)**

**UT2 PD1**

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente



**Cuando el arreglo ya está ordenado, el algoritmo es de O(N), ya que no es necesario hacer intercambios en el arreglo. Pero en el peor caso, como hace (N-1) \* (N-1) comparaciones, es de complejidad O(N^2).**

**UT3 PD3**

1. Escribe lenguaje natural, precondiciones y postcondiciones y seudocódigo para cada operación

* **Crear los elementos y la lista**

**Precondiciones: no hay.**

**Postcondiciones: Se crea una nueva lista.**

**ALGORITMO crearLista()**

**Comienzo**

**Lista lista = nueva Lista()**

**Fin**

• **Insertar un elemento en una lista (al final, la lista no está ordenada)**

**PRECONDICIONES: La lista puede estar vacía o puede contener elementos.**

**POSTCONDICIONES: Se inserta el nodo al final de la lista. Si la lista estaba vacía, el primer nodo de la lista ahora es el nodo que insertamos. Sino, el último nodo de la lista ahora apunta al nodo insertado.**

**ALGORITMO insertar(nodo)**

**SI (primero = nulo) HACER**

**primero <- nodo**

**SINO**

**temporal <- primero**

**MIENTRAS (temporal.getSiguiente() != nulo) HACER**

**temporal <- temporal.getSiguiente()**

**FIN MIENTRAS**

**temporal.setSiguiente(nodo)**

**FIN SI**

**FIN ALGORITMO**

• **Buscar un elemento en una lista**

**PRECONDICIONES: “Primero” apunta al primer nodo de la lista o a nulo si la lista está vacía. “Clave” es un objeto Comparable que se usa para buscar el nodo que tenga esa clave.**

**POSTCONDICIONES: Si existe un nodo con la etiqueta clave en la lista, se devuelve ese nodo, sino se devuelve null.**

**ALGORITMO buscar(clave)**

**temporal <- primero**

**MIENTRAS (temporal != nulo) HACER**

**SI (clave.comparar(temporal.getEtiqueta())=0) HACER**

**DEVOLVER temporal**

**FIN SI**

**temporal <- temporal.getSiguiente()**

**FIN MIENTRAS**

**DEVOLVER nulo**

**FIN ALGORITMO**

• **Listar (imprimir) todos los elementos de una lista**

**PRECONDICIONES: La lista enlazada debe existir y estar formada por nodos. “Primero” debe apuntar al primer nodo de la lista o a nulo si la lista está vacía.**

**POSTCONDICIONES: Se recorre la lista el valor de cada nodo.**

**ALGORITMO imprimir()**

**lista <- ""**

**temporal <- primero**

**MIENTRAS (temporal != nulo) HACER**

**MOSTRAR temporal.getDato()**

**temporal <- temporal.getSiguiente()**

**FIN MIENTRAS**

**RETORNAR lista.pasarAString()**

**FIN ALGORITMO**

• **Eliminar un elemento de una lista, dada su clave**

**PRECONDICIONES: “primero” debe apuntar al primer nodo de la lista o a nulo si la lista está vacía.**

**POSTCONDICIONES: Si el nodo eliminado no era el primer nodo de la lista, el nodo anterior a él se actualiza para apuntar al siguiente nodo después del eliminado.**

**ALGORITMO eliminar(clave)**

**actual <- primero**

**anterior <- nulo**

**MIENTRAS(actual!=nulo Y actual.Etiqueta().comparar(clave)!=0) HACER**

**anterior <- actual**

**actual <- actual.getSiguiente()**

**FIN MIENTRAS**

**SI (actual = nulo) HACER**

**RETORNAR falso**

**SINO SI (anterior == nulo) HACER**

**primero <- actual.getSiguiente()**

**SINO**

**anterior.setSiguiente(actual.getSiguiente())**

**FIN SI**

**DEVOLVER verdadero**

**UT3 PD7**

**Ejercicio 1:**

**Las operaciones de intersección y unión de dos conjuntos se realizan de forma bastante similar. La op. Unión debe tomar dos listas (dos conjuntos) y las recorre agregando los elementos que no estén repetidos a una tercera lista (conjuntoResultado) con todos los elementos de las dos listas sin elementos repetidos. La op. Intersección hará lo mismo, recorriendo lo conjuntos. Se recorre el primer conjunto y si los elementos también están en el segundo conjunto, se agrega al conjunto resultado.**

**Precondiciones:**

**Las listas (conjuntos) no pueden tener elementos repetidos, no pueden estar vacíos y deben tener números para poder hacer las operaciones de unión e intersección.**

**Postcondiciones:**

**El conjuntoResultado será el resultado de la unión o intersección de las dos listas(conjuntos). Contendrá el conjunto final. En caso de unión, los elementos de ambos conjuntos sin repetir. En intersección, solo los repetidos.**

**UNION:**

**COMIENZO**

**ALGORITMO union(conjunto1, conjunto2)**

**conjuntoResultado <- nuevo Conjunto**

**temporal <- conjunto1.primero**

**MIENTRAS temporal != nulo HACER**

**nuevo <- nuevo Nodo**

**conjuntoResultado.insertar(nuevo)**

**temporal <- temporal.getSiguiente()**

**FIN MIENTRAS**

**temporal2 <- conjunto2.primero**

**MIENTRAS temporal2 != nulo HACER**

**SI conjuntoResultado.buscar(temporal2) = falso HACER**

**nuevo <- nuevo**

**conjuntoResultado.insertar(nuevo)**

**FIN SI**

**temporal2 <- temporal2.getSiguiente()**

**FIN MIENTRAS**

**DEVOLVER conjuntoResultado**

**FIN**

**TIEMPO DE EJECUCIÓN:**

**Esta función es de orden O(n \* m). En este caso "n" es la longitud del conjunto1 y "m" es la longitud del conjunto2. Para cada elemento en el conjunto2, se busca si ese elemento ya está en el conjunto1 usando el método "buscar" y se agrega al conjuntoResultado solo si no está. Como el método "buscar" también tiene es O(n), se ejecuta m veces, dando a O(n \* m) como resultado.**

**--**

**INTERSECCIÓN:**

**ALGORITMO opInterseccion(conjunto1, conjunto2)**

**conjuntoResultado2 <- nuevo Conjunto**

**temporal <- conjunto1.primero**

**MIENTRAS temporal != nulo HACER**

**SI conjunto2.buscar(etiqueta temporal) = verdadero HACER**

**nuevo <- nuevo Nodo**

**conjuntoResultado2.insertar(nuevo)**

**FIN SI**

**temporal <- temporal.getSiguiente()**

**FIN MIENTRAS**

**DEVOLVER conjuntoResultado2**

**FIN**

**En el peor de los casos, cuando todos los elementos de conjunto1 están en conjunto2, el tiempo de ejecución de la función sería O(n \* m), donde "n" es el tamaño de conjunto1 y "m" es el tamaño de conjunto2**

**UT3 PD14**

Tenemos una aplicación que recibe información desde el exterior y la va procesando a medida que llega. Lamentablemente, las velocidades de arribo y proceso de la información son muy diferentes, por lo que resulta necesario poner “buffers” entre medio para adaptar los sistemas. Hemos entonces decidido que lo mejor seráhacerlo usando colas circulares, implementadas con vectores.

**Ejercicio 1:**

Desarrolla un algoritmo en seudocódigo que implemente la operación PoneEnCola (unElemento), para un TDA Cola implementado con un “vector circular”.

**Lenguaje natural: implementa la operación de agregar un elemento a una cola circular que usa un vector. Primero se verifica si hay suficiente espacio en la cola para agregar un nuevo elemento (TAMAÑO). Si el tamaño de la cola es igual al tamaño del vector menos uno, se indica que la cola está llena. Sino, se agrega el nuevo elemento al final de la cola, indicado por el índice final, y se actualiza el índice final para que apunte a la siguiente posición circular en el vector.**

**Precondiciones: La cola circular con vector ya fue creada.**

**Postcondiciones: Un elemento fue agregado al final de la cola.** **Si la cola estaba vacía antes de la operación, el índice inicio también se ha actualizado para apuntar al primer elemento de la cola.**

**ALGORITMO ponerEnCola(unElemento)**

**SI ((final + 1) MOD tamaño) = inicio HACER**

**MOSTRAR "La cola está llena"**

**SINO HACER**

**vector[final] <- unElemento**

**final <- (final + 1) MOD tamaño**

**FIN SI**

**FIN**

**Ejercicio 2:**

Desarrolla un algoritmo en seudocódigo que implemente la operación QuitaDeCola (devuelve un elemento),para un TDA Cola implementado con un “vector circular”.

**ALGORITMO quitarDeCola()**

**SI ((cola.esVacia() = verdadero) HACER**

**MOSTRAR “La cola está vacía”**

**SINO HACER**

**elemento <- vector[inicio]**

**vector[inicio] <- nulo**

**inicio <- (inicio + 1) MOD tamaño**

**DEVOLVER elemento**

**FIN SI**

**FIN**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**UT3 - PD6**

**Prueba del programa:**

Dado un nuevo archivo de entrada, al ejecutar el programa desarrollado se leerá el archivo de entrada y se

mostrarán en consola las ciudades cargadas y el total de elementos – ciudades – contenidas en la estructura.

**1.** Descargar el archivo “suc1.txt”, guardarlo en el directorio raíz de su implementación como “sucursales.txt” y ejecutar el programa, emitiendo por consola la cantidad de elementos y la lista de ciudades contenida en la estructura. La cantidad de elementos es:

a) 104

b) 105

c) 106

**d) 107 🡪 La cantidad de elementos es 107.**

**2.** Eliminar la ciudad “Chicago”- listar nuevamente el conjunto de sucursales. Dada la ciudad ·“Hong Kong”, la que le sigue en la lista es la ciudad ….

a) Buenos Aires

b) Tokio

**c) Shenzhen**

d) Cleveland

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

**3.** Levantar un segundo archivo, “suc2.txt”. Eliminar las ciudades “Shenzen” y “Tokio”. Se cumple lo siguiente:

a) Quedan 5 ciudades

b) Queda 1 ciudad y da error de ejecución

c) Queda vacía y da error de ejecución

**d) Ninguna de las anteriores 🡪 Queda vacía y no da error de ejecución**

**Texto

Descripción generada automáticamente**

**4.** Levantar un tercer archivo, “suc3.txt”. Invocar al método “Imprimir(“;\_”)”. El resultado de esta operación es:

Caracas; \_Tulsa;\_Mobile;\_Vancouver;\_ Montreal;\_

Montreal;\_Caracas;\_Tulsa;\_Mobile;\_Vancouver

Montreal;\_ Tulsa;\_ Caracas;\_Mobile;\_Vancouver;\_

**Montreal;\_Caracas;\_Tulsa;\_Mobile;\_Vancouver;**

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente