Algoritmos y Programación II (75.41)
Trabajo Práctico Nº2

26 de octubre de 2012

1. Introducción

¡Los dueños de la famosa pizzería de Gerli necesitan nuestra ayuda de nuevo! Impulsada por un éxito sin precedentes, la pizzería ha decidido abrir numerosas sucursales dispersos por la ciudad, de forma tal de ofrecer un tiempo de entrega razonable a los clientes ubicados en todos los barrios.

La pizzería ofrecerá un número telefónico 0-800 único, en donde se recibirá el pedido y se transferirá automáticamente a la sucursal de la pizzería más cercana al destino, en caso de que exista alguna (cada sucursal tiene un radio de cobertura de 1 km).

Además, se ofrecerá un servicio urgente para aquellos clientes que deseen un tiempo de entrega menor (y estén dispuestos a pagar un recargo). Los pedidos se procesarán y entregarán en orden de prioridad. Los pedidos *urgentes* tendrán prioridad sobre los pedidos *normales*; y cuantas más pizzas pida el cliente *urgente*, mayor será su prioridad. Para los clientes *normales*, todos los pedidos tienen la misma prioridad, sin importar el tamaño del pedido.

2. Consigna

Implementar el programa central de gestión de clientes y pedidos, que debe proveer las siguientes operaciones:

1. Alta/Baja/Modificación/Consulta de clientes
2. Ingresar un pedido nuevo al sistema (el cual debe ser priorizado y asignado a la sucursal más cercana).
3. Obtener los pedidos asignados a cada sucursal, en orden de prioridad.

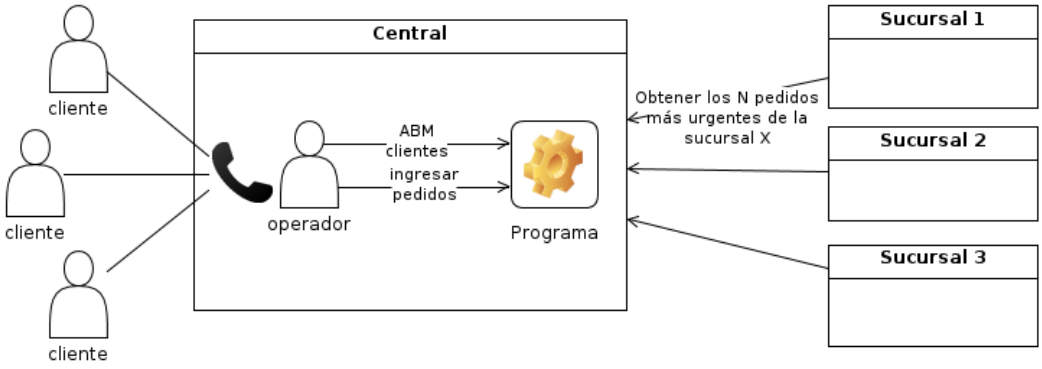

Figura 1: El programa a desarrollar es parte de un sistema automatizado, que maneja la base de datos central de clientes y pedidos

2.1. Protocolo

Este programa formará parte de un **sistema automatizado**. En lugar de presentar un menú con opciones en la pantalla, se esperará que el programa respete un **protocolo de comunicación**, en el cual se reciben **comandos** por la **entrada estándar** (`stdin`). Cada comando recibido debe ser procesado y su **resultado** debe ser escrito en la **salida estándar** (`stdout`).

2.2. Sucursales

La lista de sucursales existentes estará disponible en un archivo llamado `sucursales.csv`. Cada línea del archivo contiene la siguiente estructura:

```
Nombre de la sucursal;coordenada X;coordenada Y
```

El nombre de la sucursal es una cadena de texto (máximo 30 caracteres), y las coordenadas x e y son números enteros entre $-32{,}768$ y 32767[1]. Por ejemplo:

```
San telmo;0;49
Flores 1;-2;57
Flores 2;86;-405
```

El programa debe procesar el archivo en la etapa de inicialización, y mantener la lista de sucursales en memoria. En el Apéndice A se explica la estructura de datos que debe ser utilizada.

[1]Las coordenadas están expresadas en metros; de esta manera el sistema permite trabajar en un área cuadrada de 64 km de lado.

2.3. Clientes

El sistema debe contar con una base de datos de clientes que se guarda y se actualiza dinámicamente en memoria (es decir, cuando inicia el programa, la base de datos de clientes arranca vacía).

La idea es que el sistema identifique a un cliente por su número de teléfono, y recuerde las coordenadas x e y de su ubicación. Esto permitirá que el operario que atiende el teléfono sepa inmediatamente la dirección del cliente sin necesidad de preguntarle.

Deben proveerse los siguientes comandos para operar con la base de datos de clientes:

2.3.1. Comando: `cliente_guardar`

Formato: `cliente_guardar "telefono" x y`

Descripción: Guarda las coordenadas x e y del cliente designado con el número de teléfono proporcionado. Si ya existe una entrada en la base de datos con el mismo teléfono, la misma se reemplaza con las nuevas coordenadas.

El número de teléfono siempre se recibe entre comillas dobles, y puede contener cualquier caracter menos el caracter `"`. Longitud máxima: 30 caracteres.

Salida: `OK` en caso de no producirse error, o `ERROR` en caso contrario.

Ejemplo:

```
comando: cliente_guardar "(11) 555-5687" -3 60
salida: OK
```

2.3.2. Comando: `cliente_obtener`

Formato: `cliente_obtener "telefono"`

Descripción: Devuelve las coordenadas del cliente designado con el número de teléfono proporcionado.

Salida: Las coordenadas x e y separadas con un espacio, en caso de no producirse error, o `ERROR` en caso contrario (por ejemplo, si no se encuentra el teléfono en la base de datos).

Ejemplo:

```
comando: cliente_obtener "(11) 555-5687"
salida: -3 60
```

2.3.3. Comando: `cliente_borrar`

Formato: `cliente_borrar "telefono"`

Descripción: Elimina de la base de datos el cliente designado con el número de teléfono proporcionado.

Salida: `OK` en caso de no producirse error, o `ERROR` en caso contrario.

Ejemplo:

```
comando: cliente_borrar "(11) 555-5687"
salida: OK
```

2.4. Recepción de un pedido

Los siguientes comandos serán utilizados por el operador telefónico para efectuar la recepción de un pedido:

2.4.1. Comando: `buscar_sucursal`

Formato: `buscar_sucursal "telefono"`

Descripción: Dado el número de teléfono de un cliente, este comando debe buscar y devolver el nombre de la sucursal más cercana al cliente.

Salida: El nombre de la sucursal más cercana, o ERROR en caso de que no exista ninguna sucrusal dentro del radio de cobertura (ver sección 1), o que el teléfono no se encuentre en la base de datos.

Ejemplo:

```
comando: buscar_sucursal "(11) 555-5687"
salida: San Telmo
```

2.4.2. Comando: `ingresar_pedido`

Formato: `ingresar_pedido "telefono" pizzas`

Descripción: Guardar el pedido del cliente, con la cantidad de pizzas proporcionada, que será asignado a la sucursal más cercana. Los pedidos guardados utilizando este comando son no urgentes.

Salida: `OK` en caso de no producirse error, o `ERROR` en caso contrario.

Ejemplo:

```
comando: ingresar_pedido "(11) 555-5687" 4
salida: OK
```

2.4.3. Comando: `ingresar_pedido_urgente`

Formato: `ingresar_pedido_urgente "telefono" pizzas`

Descripción: Idem `ingresar_pedido`, con la diferencia de que el pedido es guardado como urgente (ver sección 1).

Salida: `OK` en caso de no producirse error, o `ERROR` en caso contrario.

Ejemplo:

```
comando: ingresar_pedido_urgente "(11) 555-5687" 4
salida: OK
```

2.5. Obtención de los pedidos de una sucursal

2.5.1. Comando: `obtener_pedidos`

Formato: `obterer_pedidos "sucursal" N`

Descripción: Este comando debe listar los N pedidos más urgentes asignados a la sucursal, **en orden de prioridad**. Estos pedidos se considerarán procesados y no deben volver a figurar la siguente vez que se ejecute el mismo comando.

Salida: La lista de pedidos, uno por línea o `ERROR` en caso contrario.

El formato de cada línea debe ser:
`"telefono" pizzas [normal|urgente]`

En caso de haber menos de N pedidos para la sucursal, se devuelven estos pedidos y no se produce error.

En caso de no haber pedidos para la sucursal, la salida debe ser una línea vacía.

Ejemplo:

```
comando: obtener_pedidos "San Telmo" 3
salida:
"(11) 555-5687" 4 urgente
"(11) 555-5412" 2 urgente
"(11) 555-1111" 5 normal
```

2.6. Utilización de estructuras de datos

Para la realización de este trabajo es necesario utilizar las estructuras de datos vistas en clase (tablas de hash, árboles y heaps), además de crear las estructuras adicionales que se consideren necesarias.

Todas las estructuras deben estar implementadas de la forma más genérica posible, correctamente documentadas, y con sus correspondientes pruebas unitarias.

Para implementar la búsqueda de la sucursal más cercana en forma eficiente, será necesario implementar una estructura de datos nueva, llamada **QuadTree**. En el Apéndice A se explica esta estructura de datos y las primitivas que deben ser implementadas.

3. Pruebas

Se espera que las estructuras de datos creadas específicamente para este trabajo estén acompañadas de sus correspondientes pruebas, como ya es costumbre.

Adicionalmente se provee de **pruebas automáticas para el programa completo**. Estas pruebas ya han sido escritas, y serán de utilidad para revisar que el programa cumpla con la especificación del protocolo de entrada/salida.

Una vez descomprimido el archivo zip con las pruebas[2], se debe efectuar los siguientes pasos para correr las pruebas:

1. Compilar el programa (supongamos que se llama `tp2`)
2. Ejecutar[3]:

```
$ bash pruebas/correr-pruebas.sh ./tp2
```

Cada una de las pruebas está especificada en una subcarpeta dentro de la carpeta de pruebas. Por ejemplo:

```
pruebas/
 \- correr-pruebas.sh
 \- prueba1/
     \- sucursales.csv
     \- entrada
     \- salida
 \- prueba2/
     \- sucursales.csv
     \- entrada
     \- salida
```

El script `correr-pruebas` ejecutará el programa una vez por cada prueba, con el archivo `sucursales.csv` y la entrada estándar determinados conectada al archivo `entrada`, y verificará que la salida estándar del programa sea exactamente igual al archivo `salida` (que es la salida esperada según el protocolo).

[2]Puede descomprimirse en cualquier lugar; para el ejemplo suponemos que se guardó en la misma carpeta que el TP. Es decir, el ejecutable `tp2` quedaría al mismo nivel que la carpeta `pruebas` (que contiene el archivo `correr-pruebas.sh`).

[3]Para correr las pruebas es necesario disponer de un entorno con línea de comandos Bash. En Ubuntu y Mac OSX lo más probable es que ya esté instalado. Existen varias implementaciones para Windows; por ejemplo MSYS o Cygwin.

4. Consignas opcionales

Para la aprobación del Trabajo Práctico es suficiente con implementar un programa que pase todas las pruebas (ver sección 3).

Aquí se proponen algunas consignas adicionales, que serán tenidas en cuenta en la calificación final. Notar que en alguno de estos casos será necesario modificar o expandir el protocolo de comunicación; y por lo tanto se tendrán que agregar nuevas pruebas y/o modificar las existentes.

1. Mantener la base de datos de clientes en un archivo, de forma tal que persista esta información cuando se cierre el programa. Además, mantener el historial de cantidad de pizzas pedidas por cada cliente, y agregar una opción que permita generar el listado de los N clientes más "fieles" (en orden decreciente).

2. Mantener el historial de pizzas pedidas a cada sucursal, y agregar una opción que permita generar el listado de las N sucursales más "populares" (en orden decreciente).

3. Algunos clientes vuelven a llamar luego de un rato para preguntar el estado de su pedido. Implementar esta opción, que permita ingresar el número de teléfono del cliente e imprima los últimos N pedidos recibidos y el estado de cada uno (donde el estado puede ser "esperando para ser derivado a la sucursal X", o "ya fue derivado a la sucursal X").

4. Agregar una opción que permita generar una visualización de las sucursales y clientes ubicados en el espacio bidimensional, y la partición del espacio realizada por el QuadTree, similar a como se ve en la Figura 2. La imagen puede ser generada en cualquier formato a elección[4]. Ejemplos de formatos con implementación simple: Netpbm[5], PCX[6], SVG[7].

[4] `http://en.wikipedia.org/wiki/Image_file_formats`
[5] `http://en.wikipedia.org/wiki/Netpbm_format`
[6] `http://en.wikipedia.org/wiki/PCX`
[7] `http://www.w3schools.com/svg/svg_examples.asp`

5. Criterios de aprobación

A continuación describimos criterios y lineamientos que deben respetarse en el desarrollo del trabajo.

5.1. Programa

El programa debe cumplir los siguientes requerimientos:

- Debe estar adecuadamente estructurado y modularizado, utilizando funciones definidas de la forma más genérica posible, sin caer en lo trivial.
- El código debe ser claro y legible.
- El código debe estar comentado y las funciones definidas, adecuadamente documentadas.
- El programa debe compilar sin advertencias ni mensajes de error[8], debe correr sin pérdidas de memoria, uso de valores sin inicializar, o errores en general. Es decir que, el programa debe correr en valgrind sin errores.
- Además, claro, debe satisfacer la especificación de la consigna.

5.2. Informe

El informe deberá consistir de las siguientes partes:

- Carátula con la información de los integrantes del grupo y el ayudante asignado.
- **Análisis, especificación y diseño**: describir el problema, cuál es la solución y cómo se lleva a cabo. Incluir las convenciones de código utilizadas (criterios para nombres de variables y funciones, criterios de indentación, etc).
- **Implementación**: Incluir aquí *todo* el código fuente utilizado en formato `monoespaciado`, para facilitar su lectura.
- **Pruebas unitarias**: probar las funcionalidades de cada estructura o porción del programa.

[8] `-Wall -pedantic -std=c99`

- Mantenimiento *(Opcional)*: cambios importantes realizados a lo largo del trabajo.
- También *opcionalmente,* toda explicación adicional que consideren necesaria, referencias utilizadas, dificultades encontradas y conclusiones.

El informe debe estar lo más completo posible, con presentación y formato adecuados. Por ejemplo, este enunciado cumple con los requerimientos de un informe bien presentado.

6. Entrega

El trabajo consiste en:

- El informe impreso.
- El informe digital, en formato `.pdf`
- Una versión digital de **todos** los archivos de código fuente, separados del informe, en un archivo comprimido (`.zip` o `.tar.gz`).

Los dos últimos deben enviarse a la dirección `tps.7541rw@gmail.com`, colocando como asunto: **TP2 - Padrón1 - Apellido1 - Padrón2 - Apellido2**.

Se aclara que por código fuente se entiende todos los archivos `.h` y `.c`, el archivo `Makefile` para poder compilar, y todos los archivos adicionales que sean necesarios para ejecutar el programa y las pruebas. No deben entregarse nunca archivos `.o` u otros archivos compilados.

El informe impreso debe entregarse en clase. El plazo de entrega vence el **lunes 12 de noviembre de 2012**.

A. QuadTree

A continuación presentamos una descripción del QuadTree[9] y una **sugerencia** de implementación, **en pseudocódigo**.

Un QuadTree es un árbol en el que cada nodo tiene exactamente 0 ó 4 hijos. Se utiliza comúnmente para particionar espacios bidimensionales, dividiendo el mismo recursivamente en cuatro cuadrantes. Esto permite hacer búsquedas en forma eficiente, de la misma manera que un árbol binario permite hacer búsquedas en espacios de una dimensión.

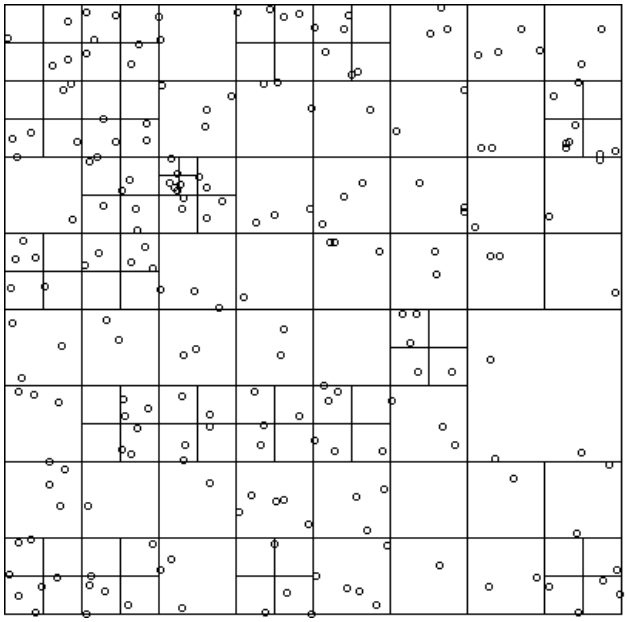

Figura 2: Representación gráfica de un QuadTree y los puntos almacenados en el mismo.

[9]Más información: `http://en.wikipedia.org/wiki/Quadtree`

A.1. Estructuras

Cada punto almacenado en el árbol consta de un par de coordenadas x e y:

```
struct Punto:
    int x
    int y
```

Además vamos a necesitar una estructura que represente un rectángulo en el espacio bidimensional:

```
struct Rect:
    Punto arr_izq
    Punto ab_der
```

Cada nodo representa a un rectángulo en el espacio, almacena una lista de puntos, y tiene 4 hijos, que representan los 4 subcuadrantes:

```
struct QuadTree:
    Rect area
    ListaDePuntos puntos
    QuadTree norOeste
    QuadTree norEste
    QuadTree surOeste
    QuadTree surEste
```

Cada nodo del árbol puede almacenar en la lista una cantidad máxima de puntos, que definimos como una constante:

```
CAPACIDAD_NODO = 4
```

A.2. Primitivas

Además de las primitivas para crear y destruir un QuadTree, vamos a necesitar dos primitivas para trabajar con el mismo: **insertar un punto** y **consultar los puntos que están dentro de un rectángulo**.

Listado 1: Primitiva para insertar un punto

```
def insertar(QuadTree qt, Punto p):
    // Ignorar puntos que no pertenecen a este QuadTree
    if not contiene(qt.area, p):
        return false // el punto no puede ser insertado

    // Si hay espacio en este QuadTree, lo agregamos
    if cantidad(qt.puntos) < CAPACIDAD_NODO:
        insertar_al_final(qt.puntos, p)
        return true

    // En caso contrario, tenemos que subdividir este
       QuadTree y agregar el punto en el subnodo que lo
       acepte
    if qt.norOeste == null:
        subdividir(qt)

    if insertar(qt.norOeste, p): return true
    if insertar(qt.norEste, p): return true
    if insertar(qt.surOeste, p): return true
    if insertar(qt.surEste, p): return true

    // El punto no pudo ser agregado por alguna razón
       desconocida (no debería suceder)
    return false
```

Listado 2: Primitiva para consultar todos los puntos que están dentro de un rectángulo

```
def consultarRect(QuadTree qt, Rect rect):
    // Preparamos una lista de puntos para devolver el
       resultado:
    ListaDePuntos resultado

    // Abortamos si el rectángulo no tiene intersección con
       este QuadTree
    if not intersectan(qt.area, rect):
        return resultado; // lista vacía

    // Revisamos todos los puntos en este nodo:
    por cada Punto p en qt.puntos:
        if contiene(rect, p):
            insertar_al_final(resultado, p)

    // Terminamos aquí, si no hay más hijos:
    if qt.norOeste == null:
        return resultado

    // En caso contrario, consultar los hijos y agregar los
       puntos encontrados a la lista
    agregar(resultado, consultarRect(qt.norOeste, rect))
    agregar(resultado, consultarRect(qt.norEste, rect))
    agregar(resultado, consultarRect(qt.surOeste, rect))
    agregar(resultado, consultarRect(qt.surEste, rect))

    return resultado
```