Curso 2019-2020
Estructura de Datos y Algoritmia
Práctica 2

Departamento de Lenguajes y Sistemas Informáticos

Objetivo

Vamos a utilizar tipos de datos lineales de las STL y definiremos también nuestros propios tipos de datos lineales (listas). Seguiremos trabajando con información relacionada con gestión de paquetes turísticos.

Fechas importantes: plazo de entrega desde el lunes 25 de noviembre hasta el viernes 29 de noviembre de 2019.

A tener en cuenta (IMPORTANTE):

- En las clases que se declaran a continuación, podéis añadir los métodos que consideréis necesarios para la realización de la práctica.
- Además en todas las clases tenéis que implementar las **operaciones canónicas** (constructor o constructores, constructor de copia, operación asignación, destructor y operador <<).
- Añadir el constructor por defecto en la clase `Localidad`.

1. Clases

1.1 Clase NodoL

Esta clase servirá para construir una lista, y se definirá por tanto en la parte privada de la clase `LNear` (de manera que sólo la lista puede acceder a un objeto de tipo `NodoL`). Contendrá

- las variables de instancia siguientes(se podrán añadir las que se consideren necesarias justificando su inclusión):
  - `Localidad localidad;`
  - `int distancia;`
  - `NodoL *next;`
  - `NodoL *prev;`

- y los siguientes métodos de instancia (se podrán añadir los que se consideren necesarios justificando su inclusión):

  - `NodoL( Localidad n )`: inicializa la localidad a la pasada por parámetro, y el resto de variables de instancia a valores por defecto (la distancia a valor -1);
  - sobrecarga del `operator <<`: muestra por pantalla la información almacenada en el nodo en el siguiente formato: nombre de la localidad, un espacion en blanco, '(', la distancia almacenada, y ')'; por ejemplo
    Sira (35)

1.2 Clase LNear

La clase **LNear** que hay que implementar contendrá información referente a las localidades más cercanas a una dada, y dicha lista estará **ordenada por el valor de la distancia en primera instancia, y por orden lexicográfico en segunda**. En concreto estará formada por:

- las variables de instancia siguientes:

  - `NodoL *pr;`
  - `NodoL *ul;` *//al estar ordenada, tenemos acceso directo a la distancia de la más alejada para saber el rango actual*
  - `Localidad error;`

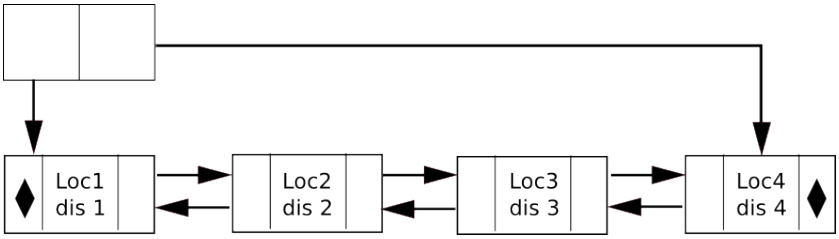
- y los siguientes métodos de instancia (se podrán añadir los que se consideren necesarios justificando su inclusión):

  - todas las operaciones canónicas de la clase;
  - `LNear()`: inicializa las variables de instancia a sus valores por defecto (null);
  - `bool esVacia()`: nos indica si la lista está vacía;
  - `int rango()`: devuelve la distancia a la que se encuentra la localidad más alejada actualmente almacenada; si no hay ninguna devuelve -1;
  - `void insertaLocalidad(Localidad p, int d)`: crea un nodo con la información pasada por parámetro y lo inserta de forma ordenada según los criterios comentados anteriormente; en caso de que ya haya una localidad con el mismo nombre y distancia no se realizará la inserción;

- `int borraLocalidad(string s)`: elimina de la lista la localidad cuyo nombre se pasa como parámetro, devolviendo la distancia a la que se encuentra. Si hay más de una borra y devuelve la distancia de la primera encontrada empezando a buscar por la cabeza de la lista. Si no la encuentra devuelve -1.
- `void borraLocalidades(int k)`: elimina de la lista todas las localidades cuya distancia sea superior a `k`.
- `Localidad & getLocalidad( int i )`: devuelve la referencia a la localidad que ocupa la posición indicada por el parámetro `i` en la lista, siendo 0 la primera posición de la lista. Si es vacía la lista devuelve el objeto `error`.
- sobrecarga del `operator` <<: escribe la lista empezando por la cabeza por la salida estándar, con el siguiente formato:

  localidad1 (distancia1)
  localidad2 (distancia2)
  localidad3 (distancia3)

  Gráficamente:



1.3 Clase Provincia

La clase `Provincia` que hay que implementar contiene

- las siguientes variables de instancia:
  - `Localidad lc;`
  - `LNear locprox;` *//objeto de tipo lista*
- y los siguientes métodos de instancia:
  - `public Provincia( Localidad l)`: asigna la localidad pasada por parámetro a la variable de instancia, e inicializa el resto de variables a su valor por defecto;
  - `public void calculaCercanas(Coleccion &, int )`: añade a su lista las localidades del mapa que están dentro del rango definido por el segundo

parámetro según la distancia; para obtener las localidades más cercanas aplicaremos sobre sus coordenadas una *distancia de Manhattan* [1]. Si la lista ya tenía almacenadas localidades, habrá que eliminarlas previamente.

- `public int borraLocalidad(string)`: elimina de su lista de localidades cercanas la localidad cuyo nombre se pasa por parámetro, de manera que devuelve la distancia a la que está dicha localidad eliminada si la encuentra, y -1 en cualquier otro caso.
- `public LNear & getCercanas()`: devuelve la lista de las localidades más cercanas; si no hay ninguna devuelve una lista vacía;
- `public string getCostera(Coleccion &)`: devuelve el nombre de la localidad costera más cercana almacenada en la lista; si no hay ninguna devuelve la cadena "no hay ninguna localidad costera";
- `public LNear getCosteras(Coleccion &)`: devuelve la lista de todas las localidades costeras almacenadas en la lista empezando en la más cercana; si no hay ninguna devuelve una lista vacía;
- `public string getConAeropuerto( )`: si la localidad no tiene aeropuerto devuelve el nombre de la localidad más cercana que tenga aeropuerto almacenada en la lista; si ya tiene aeropuerto devuelve la cadena "propio" y si no tiene y no encuentra ninguna devuelve la cadena "sin aeropuerto";
- sobrecarga del `operator <<`: muestra por pantalla la localidad capital de provincia y en la siguientes líneas la lista de las localidades más cercanas en orden de cercanía según el formato definido en `LNear`:

  Playa Blanca (2)
  Puerto Del Carmen (3)
  Teguise (3)
  Orzola (5)

2. Aplicación: crucero

Utilizando la clase `Coleccion` como base de almacenamiento de la información que se leerá desde fichero, en esta aplicación queremos realizar un crucero alrededor de una isla.

Durante este crucero se realizarán paradas que coincidirán con todas las localidades costeras que hay en la misma a partir de la primera localidad que habremos almacenado en la `Coleccion` tras leer el fichero. Si esta localidad no es costera,

[1] la *distancia de Manhattan* es un caso especial de distancia de Minkowsky definida como $d(l1, l2) = |x_1 - x_2| + |y_1 - y_2|$

tendremos que coger un autobús hasta la localidad costera más cercana, desde la que empezaremos la travesía. El crucero terminará en la misma localidad costera donde lo hemos empezado.

Implementa un fichero denominado `Crucero.cc` que contendrá un `main` que recibirá como parámetro de entrada el fichero con los datos del mapa y localidades del mismo.

El ejecutable de este fichero, tras pasarle como parámetro un fichero de texto, mostrará en el orden de visita, el nombre de las localidades costeras visitadas, además de la *distancia de crucero* [2] entre cada una de ellas y la anteriormente visitada. Más detalles sobre la salida en la sección 2.4.

2.1 Algoritmo de recorrido por la costa: partida

El **puerto de partida** será el de la localidad costera más cercana a la de referencia. Si la de referencia ya es costera, empezaremos el crucero en ella. Si hay más de una localidad costera a la misma distancia, se cogerá como referencia la primera encontrada en el vector de localidades de la `Coleccion`.

Para realizar el recorrido de la costa, necesitamos una ventana `V` con información de las 8 coordenadas vecinas de un objeto de tipo `Coordenadas P`. Esta información estará relacionada con la dirección en que nos moveremos.

5	6	7
4	P	0
3	2	1

direcciones

(-1,-1)	(-1,0)	(-1,+1)
(0,-1)	(x,y)	(0,+1)
(+1,-1)	(+1,0)	(+1,+1)

posiciones relativas respecto a P

Para saber a partir de qué coordenadas de las 8 conectadas empezamos a buscar las siguientes coordenadas de contorno, tendremos información en cada momento de:

- las coordenadas en las que nos encontramos (P),
- la dirección de donde viene de las coordenadas anteriores (un valor entre 0 y 7).

El recorrido se realizará **en el sentido de las agujas del reloj** de los 8 pares de coordenadas conectadas. Una vez que tenemos las coordenadas del puerto de salida,

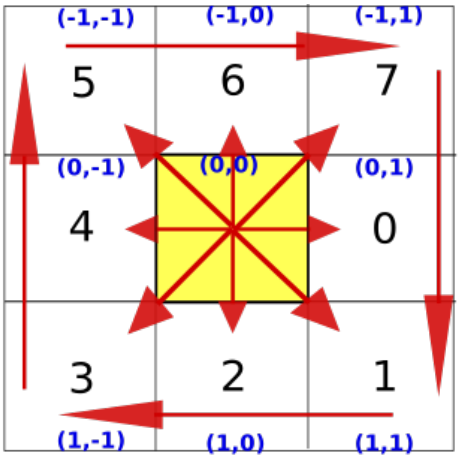
[2] número de coordenadas clasificadas como T por los que pasamos para llegar a la siguiente localidad

el algoritmo decidirá cuál es el siguiente par de coordenadas del contorno. En cada paso, se realizará una búsqueda de las siguientes coordenadas vecinas utilizando la tabla anterior, para lo que tendremos en cuenta la información indicada, y de forma iterativa se recorrerá toda la costa. Antes de llegar a esta parte iterativa, hay que encontrar el segundo par de coordenadas de una forma diferente (ya que no tenemos todavía una dirección definida) que se explicará a continuación.

2.2 Algoritmo de recorrido por la costa: encontrar el segundo par de coordenadas y primera dirección

Una vez que tenemos localizado el primer par de coordenadas (p_1) en el mapa (correspondiente a la primera localidad costera), para encontrar el segundo par de coordenadas (p_2) es suficiente con empezar recorriendo la ventana alrededor de p_1 desde la dirección 0 a la 1 y sucesivamente en sentido horario, deteniendo el recorrido cuando la transición de una posición a otra en el vector nos dé la primera transición M-T o M-L en el mapa.

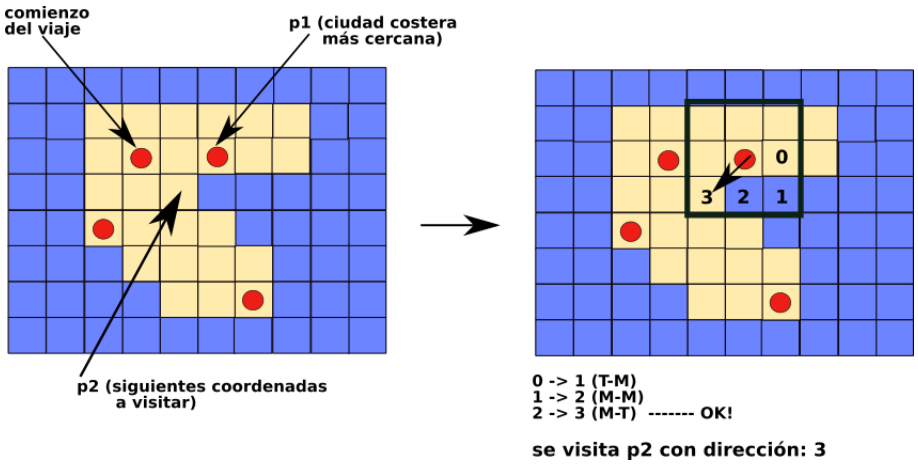
Las coordenadas donde se encuentre la 'T' (o la 'L') (p_2), serán entonces el segundo par de coordenadas a visitar. Y el índice de la ventana será la dirección en que nos moveremos.



V	(0,1)	(1,1)	(1,0)	(1,-1)	(0,-1)	(-1,-1)	(-1,0)	(-1,1)
	0	1	2	3	4	5	6	7

Es decir, recorreremos en este orden las direcciones: 0,1,2,3,4,5,6,7. La posición en la ventana en la que se encuentre el siguiente par de coordenadas a visitar nos dará la dirección en la que nos vamos a mover.

Ejemplo de cómo encontrar el siguiente par de coordenadas a partir de las iniciales:



2.3 Algoritmo de recorrido por la costa: encontrar el resto de coordenadas del contorno

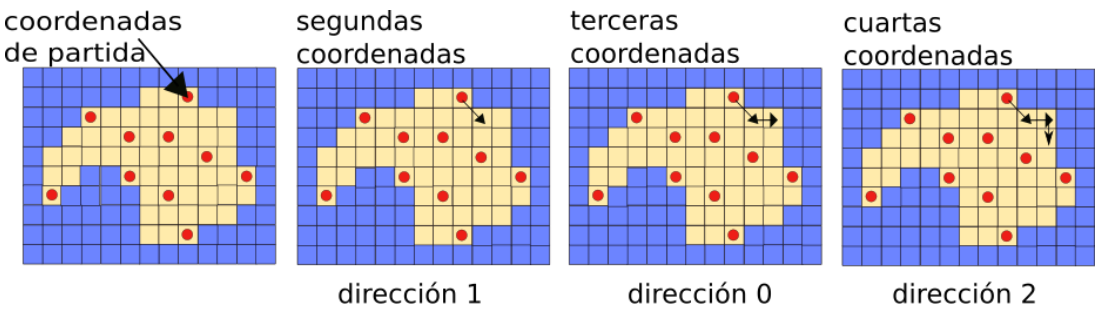
Una vez que tenemos ya el par de coordenadas de referencia (p_2) y una dirección de donde venimos (d_1), representada por un número entero entre 0 y 7, la búsqueda del siguiente par de coordenadas del contorno es sistemática. La búsqueda de p_3 a partir de p_2 empezará en la ventana V colocada sobre p_2 en $\boxed{\text{la dirección } d_2{=}(d_1+5)\%8}$, de forma que si en esa posición de la ventana alrededor de p_2 hay una `'T'` o `'L'` ya lo hemos encontrado, y si no, incrementaremos $d_2 = (d_2+1)\%8$ recorriendo toda la ventana alrededor de p_2 en sentido horario.

En el ejemplo siguiente, el primer par es (1,8), el segundo par se obtiene partiendo de la dirección 0 (como se ha explicado en la sección anterior) y es (2,9). Para encontrar el tercer par coordenadas (y siguientes), como ya partimos de toda la información (coordenadas y dirección) utilizamos la fórmula del recuadro. En concreto, partimos de (2,9) y dirección 1. Si calculamos i=(1+5)%8=6, este valor nos indica que empezaremos a buscar el siguiente par de contorno a partir de la posición de la dirección 6. Siguiendo el sentido horario, el siguiente par de coordenadas distintas de `'M'` en el mapa serán las que corresponden a la dirección 0 (coordenadas (2,10)). Este proceso lo iremos repitiendo hasta que lleguemos a las coordenadas de la localidad de partida (punto de inicio).

Las direcciones por las que se va pasando (y las coordenadas) se almacerán en estructuras de datos adecuadas, de forma que cuando llegamos a una nueva localidad,

podamos conocer la distancia desde la última localidad.

Ejemplo de cómo ir encontrando el contorno:



2.4 Implementación de Crucero.cc y salida

El fichero Crucero.cc contendrá un método `main` que:

- detectará el parámetro de la aplicación (fichero de texto con la información del mapa y localidades);
- abrirá el fichero de texto lo almacenará todo en un objeto de tipo `Coleccion`;
- mostrará por pantalla la información que se detalla a continuación.

Se mostrará por pantalla la siguiente información:

- Localidad de origen si no es costera, localidad costera más cercana y su distancia (utilizando un espacio en blanco como separador);
- se mostrará en cada línea el nombre de la localidad costera de inicio, la siguiente visitada y la distancia (utilizando un espacio en blanco como separador);
- la siguiente línea contendrá la distancia total recorrida (corresponderá al contorno) con el formato indicado en el ejemplo que viene a continuación;
- la siguiente línea contendrá la cadena de direcciones del contorno (sin espacios en blanco).

Ejemplo de FICHERO:

```
MMMMMMMMMMMM
MMMMMMTTTMMMM
MMMTTTTTTTTMM
MMTTTTTTTTTMM
MTTTTTTTTTTMM
MTTMMTTTTTTTM
MMMMMMMMMMMM
<LOCALIDAD>
Asti
3 7
<INFO>
hotel 2
restaurante 1
<LOCALIDAD>
Calari
3 5
<INFO>
museo 1
monumento 3
* la playa de la caleta
<LOCALIDAD>
Bhosa
1 8
<INFO>
museo 3
hotel 3
restaurante 4
<LOCALIDAD>
Sinrola
2 3
<INFO>
** la fontana di trevi
hotel 10
restaurante 10
```

Para este fichero de entrada la salida será:

```
Asti Bhosa 3
Bhosa Sinrola 18
Sinrola Bhosa 5
total=23
10221444444543467700700
```

3. Normas generales

Entrega de la práctica:

- Lugar de entrega: servidor de prácticas del DLSI, dirección `http://pracdlsi.dlsi.ua.es`;
- **Plazo de entrega:** desde el lunes 25 de noviembre hasta el **viernes 29 de noviembre a las 23:59 horas**;
- Se deben entregar las prácticas en **un fichero comprimido**, con todos los ficheros creados y ningún directorio de la siguiente manera:

  ```
  tar cvfz practica2.tgz Coordenadas.h Coordenadas.cc Localidad.h
  Localidad.cc Coleccion.h Coleccion.cc LNear.h LNear.cc Crucero.cc
  ```

- No se admitirán entregas de prácticas por otros medios que no sean a través del servidor de prácticas;
- El usuario y contraseña para entregar prácticas es el mismo que se utiliza en UACloud;
- La práctica se puede entregar varias veces, pero sólo se corregirá la última entregada;
- Los programas deben poder ser compilados sin errores ni warnings con el compilador de C++ existente en la distribución de Linux de los laboratorios de prácticas; si la práctica no se puede compilar su calificación será 0. Se recomienda compilar y probar la práctica con el autocorrector inmediatamente antes de entregarla;
- Los ficheros fuente deber estar adecuadamente documentados usando comentarios en el propio código, sin utilizar en ningún caso acentos ni caracteres especiales;
- Es imprescindible que se respeten estrictamente los formatos de salida indicados, ya que la corrección principal se realizará de forma automática;

- Al principio de cada fichero entregado (primera línea) debe aparecer el DNI y nombre del autor de la práctica (dentro de un comentario) tal como aparece en UACloud (pero sin acentos ni caracteres especiales);

  Ejemplo:
  `DNI 23433224 MUÑOZ PICÓ, ANDRÉS` ⇒ NO
  `DNI 23433224 MUNOZ PICO, ANDRES` ⇒ SI

Sobre la evaluación en general:

- La práctica debe ser un trabajo original de la persona que entrega; en caso de detectarse indicios de copia de una o más entregas se suspenderá la práctica con un 0 a todos los implicados;

- La influencia de la nota de esta práctica sobre la nota final de la asignatura está publicada en la ficha oficial de la asignatura (apartado evaluación).

4. Probar la práctica

- En UACloud se publicará un corrector de la práctica con algunas pruebas (se recomienda realizar pruebas más exhaustivas).

- El corrector viene en un archivo comprimido llamado `correctorP2.tgz`. Para descomprimirlo se debe copiar este archivo donde queramos realizar las pruebas de nuestra práctica y ejecutar: `tar xfvz correctorP2.tgz`.

  De esta manera se extraerán de este archivo:

  - El fichero `corrige.sh`: *shell-script* que tenéis que ejecutar.
  - El directorio `practica2-prueba`: dentro de este directorio están los ficheros
    - `p01.cc`: programa fuente en C++ con un método main que realiza una serie de pruebas sobre la práctica.
    - `p01.txt`: fichero de texto con la salida correcta a las pruebas realizadas en `p01.*`.

- Una vez que tenemos esto, debemos copiar nuestros ficheros fuente (TODOS) al mismo directorio donde está el fichero `corrige.sh`.

- Sólo nos queda ejecutar el *shell-script*. Primero debemos darle permisos de ejecución. Para ello ejecutar:

```
chmod +x corrige.sh
corrige.sh
```