INF-134 Estructuras de Datos, 2022-1

Tarea 3

Profesores: Juan Pablo Castillo, Roberto Díaz, Javier Robledo
juan.castillog@sansano.usm.cl, roberto.diazu@usm.cl, javier.robledo@usm.cl

Ayudantes:
Domingo Benoit (domingo.benoit@sansano.usm.cl)
Joaquín Gatica (joaquin.gatica@sansano.usm.cl)
Diana Gil (diana.gil@sansano.usm.cl)
Martín Ruiz (martin.ruiz@usm.cl)
Martín Salinas (martin.salinass@sansano.usm.cl)
Betrice Valdes (beatrice.valdes@sansano.usm.cl)

Fecha de entrega: 10 de julio, 2022.
Plazo máximo de entrega: ~~5 horas.~~ 3 días.

1. Reglas del Juego

La presente tarea debe hacerse en grupos de 3 personas. Toda excepción a esta regla debe ser conversada con los ayudantes (usando los correos indicados en el encabezado de esta tarea). No se permiten de ninguna manera grupos de más de 3 personas. Debe usarse el lenguaje de programación C++. Al evaluarlas, las tareas serán compiladas usando el compilador `g++`, usando la línea de comando `g++ archivo.cpp -o output -Wall`. Alternativamente, se aceptan variantes o implementaciones particulares de `g++`, como el usado por MinGW (que está asociado a la IDE code::blocks). Se deben seguir los tutoriales que están en Aula USM, cualquier alternativa explicada allí es válida. Recordar que una única tarea en el semestre puede tener nota menor a 30. El no cumplimiento de esta regla implica reprobar el curso. No se permite usar la biblioteca *stl*, así como ninguno de los *containers* y algoritmos definidos en ella (e.g. `vector`, `list`, etc.). Está permitido usar otras funciones de utilidad definidas en bibliotecas estándar, como por ejemplo `math.h`, `string`, `fstream`, `iostream`, etc.

2. Objetivos

Entender y familiarizarse con la implementación y utilización de estructuras de datos tipo hashing y grafos.

Además se fomentará el uso de las buenas prácticas y el orden en la programación de los problemas correspondientes.

3. Problemas a Resolver

En esta sección se describen los problemas a resolver implementando programas, funciones o clases en C++. Se debe entregar el código para cada uno de los problemas en archivos `.cpp` separados (y correspondientes `.hpp` de ser necesario).

3.1. Problema 1: Caching

Se debe implementar el TDA `CacheDiccionario` que permite guardar definiciones.

Cada definición, corresponde a un par término-significado. Un término corresponderá a una palabra o frase corta con un largo inferior a los 32 caracteres. Cada significado corresponderá a un texto de largo indefinido y podrá ser un string vacío.

Para resolver consultas sobre el TDA rápidamente deberá ser implementado usando las técnicas de hashing vistas en el curso. Por otro lado, habrán fuertes restricciones de la cantidad de definiciones que podrá almacenar como máximo el TDA en memoria y la tabla deberá "limpiarse" cada cierto tiempo. Por último, deberá ser posible evaluar la eficiencia del TDA obteniendo estadísticas de rendimiento.

3.1.1. Definición

El TDA `CacheDiccionario` debe soportar cuatro operaciones como mínimo:

- `bool query(string termino, string& significado);`

  Permite consultar por una definición, buscando su `termino`. De encontrarse, la operación deberá retornar verdadero y entregar el significado asociado al término en el parámetro por referencia `significado`. En cualquier otro caso, retornará falso y un string vacío en el parámetro por referencia `significado`.

- `void insert(string termino, string significado);`

  Permite insertar una nueva definición, recibiendo como parámetros su `termino` y `significado`. El TDA nunca deberá negar u omitir una inserción, sino que al alcanzarse la máxima capacidad del almacenamiento interno deberá realizarse una limpieza. Esta limpieza deberá eliminar la mitad de los elementos con menor cantidad de consultas. Posterior a esta limpieza, deberá quedar almacenada la definición que se pretendía insertar. En caso de insertarse un término repetido, deberá reemplazarse el significado anterior, manteniendo la cantidad de consultas realizadas previamente.

- `void querystats(int& total, int& conocidos, int& desconocidos);`

  Entrega estadísticas de las consultas realizadas sobre el TDA desde su creación, entregando los siguientes valores en los parámetros por referencia:

  - `total`: cantidad total de consultas realizadas.
  - `conocidos`: cantidad total de consultas realizadas a términos con significado no vacío.
  - `desconocidos`: cantidad total de consultas realizadas a términos con significado vacío.

- `void perfstats(int& accesses, int& hits, int& misses, int& cleanups);`

  Entrega estadísticas de rendimiento del TDA desde su creación, entregando los siguientes valores en los parámetros por referencia:

  - `accesses`: la cantidad de accesos realizadas a todas las ranuras de la tabla de hashing base del TDA.
  - `hits`: la cantidad de veces que se encontró un término consultado.
  - `misses`: la cantidad de veces que no se encontró un término consultado.
  - `cleanups`: la cantidad de veces que se realizó una limpieza.

3.1.2. Detalles de implementación

El TDA deberá implementarse usando una tabla de hashing tomando en cuenta lo siguiente:

- La tabla de hashing podrá guardar hasta un máximo de 128 definiciones.
- La política de resolución de colisiones será por medio de hashing cerrado.

- Como principal estrategia se deberá utilizar, a elección, hashing cuadrático o doble hashing. Se pueden mezclar entre sí o con hashing lineal para obtener un mejor rendimiento.
- La función de hashing a utilizar será a libre elección, pero estará sujeta a las restricciones vistas en el curso.
- El significado de cada término en la caché puede guardarse en la tabla de hashing misma o en una estructura de datos auxiliar en memoria principal.
- Al momento de realizar una limpieza, de haber empates en la cantidad de consultas, puede elegirse arbitrariamente que término eliminar y cual no.

El TDA deberá llamarse `CacheDiccionario` teniendo como mínimo las operaciones descritas y un constructor sin parámetros que lo inicializa como vacío. Puede extender el TDA e incluir otras operaciones o funciones que estime conveniente.

Notar que para este problema como mínimo será necesario entregar un archivo de implementación `cache-diccionario.cpp` del TDA. Evite definir una función `main` en este archivo.

Opcionalmente, puede entregar otro archivo `main.cpp` que contenga una función `main` que haga uso del TDA y que demuestre su buen funcionamiento. Sin embargo, debe considerar que durante la revisión es posible que se verifique su solución haciendo uso de otros potenciales `main.cpp`.

Adicionalmente, se puede entregar un archivo de cabecera `cache-diccionario.hpp` y un archivo `makefile` si estima que facilitaran la revisión de la tarea.

3.2. Problema 2: Grafo

La novela "Los miserables" de Victor Hugo es considerada una de las más importantes del siglo XIX. En el presente problema se busca representar como un grafo no dirigido con pesos cuyos vértices/nodos representan personajes y los arcos/aristas entre ellos representan la cantidad de veces en que aparecen juntos en la novela.

3.2.1. Entrada de datos

La entrada de datos del programa consistirá de dos archivos de texto plano ASCII, uno conteniendo el grafo como matriz de adyacencia y el otro los nombres de los personajes.

El archivo con la matriz de adyacencia `miserables.csv` tiene formato CSV cuyos campos están separados por punto y coma (`;`). Un ejemplo reducido de este archivo es [1]:

```
;11;48;55;27;25;...
11;0;1;19;17;12;...
48;1;0;4;1;1;...
55;19;4;0;0;2;...
...
```

De este archivo podemos decir, por ejemplo, que del vértice con ID 11 al vértice con ID 55 se tiene un arco con peso 19, es decir, aparecen 19 veces juntos en la novela. Esto se puede observar de mejor manera si se abre el archivo con una aplicación de manejo de hojas de cálculo como Excel:

	A	B	C	D	E	F
1		11	48	55	27	25
2	11	0	1	19	17	12
3	48	1	0	4	1	1
4	55	19	4	0	0	2

El archivo `miserables_id_to_names.csv`, también de tipo CSV con separador `,`, traduce un `ID` al nombre del personaje que le corresponde. A continuación un extracto de dicho archivo [2]:

[1] En este ejemplo, los tres puntos seguidos (`...`) implican que hay más información pero fue omitida, no que en el archivo existen puntos

[2] *Ver nota al pie de página* 1

```
Id,Label
11,Valjean
48,Gavroche
55,Marius
27,Javert
...
```

Con este archivo sabemos que el vértice con ID 11 es Valjean y el vértice con ID 55 es Marius, por lo que Valjean y Marius aparecen 19 veces juntos en la novela.

3.2.2. Salida de datos

El programa deberá determinar las siguientes métricas a partir de los dos archivos de entrada:

- **Personaje Principal**: será aquél vértice con mayor grado del grafo.
- **Personaje secundario más relevante**: será aquel personaje que tenga la distancia mayor a la del personaje principal, entendiéndose como distancia entre vértices el camino más corto considerando los pesos (la suma de los pesos de los arcos del camino de menor magnitud).
- **Longitud de camino promedio**: para un grafo $G = (V, E)$ la longitud de camino promedio viene dada por la distancia promedio, la cual se define como:

$$averageDistance(G) = \frac{\sum_{v,p \in V} shortestPath(v,p)}{\binom{n}{2}}$$

  En donde $shortestPath(v,p)$ es el largo del camino más corto que NO toma en cuenta el peso del arco entre v y p (asume 0 si su peso 0 y 1 si es distinto de 0). La suma de estos largos es dividida por la cantidad total de combinaciones de pares de vértices posibles que se pueden obtener.

- **Longitud de camino promedio (con pesos)**: para un grafo con pesos $G = (V, E, w)$ la longitud de camino promedio (con pesos) está dada por la distancia promedio con pesos, la cual se define como:

$$averageWeightedDistance(G) = \frac{\sum_{v,p \in V} weightedShortestPath(v,p)}{\sum_{e \in E} w(e)}$$

  En donde $weightedshortestPath(v,p)$ es el largo del camino más corto que SI toma en cuenta el peso del arco entre v y p. La suma de estos largos se divide por la suma total del peso de los arcos (obtenida por $w(e)$ donde $e \in E$).

Un ejemplo de salida estándar del programa sería el siguiente [3]:

```
Personaje Principal: Valjean
Personaje secundario mas relevante: Jondrette
Diametro: 6.597
Diametro (con pesos): 21.299
```

4. Entrega de la Tarea

Entregue la tarea enviando un archivo comprimido `tarea1-apellido1-apellido2-apellido3.zip` o `tarea1-apellido1-apellido2-apellido3.tar.gz` (reemplazando sus apellidos según corresponda) a la página aula.usm del curso, a más tardar el día 10 de julio, 2022, a las 23:59:00 hs (Chile Continental), el cual contenga como mínimo:

- Los archivos con los códigos fuentes necesarios para el funcionamiento de la tarea. ¡Los archivos deben compilar!

[3] Los números/personajes en este ejemplo no necesariamente son los correctos

- **nombres.txt**: Nombre, ROL, Paralelo y qué programó cada integrante del grupo.
- **README.txt**: Instrucciones de compilación en caso de ser necesarias, y la forma de compilación que usó para cada problema de la tarea (debe ser alguna de las indicadas en los tutoriales entregados en Aula USM).

5. Restricciones y Consideraciones

- ~~Por cada hora de atraso en la entrega de la tarea se descontarán 10 puntos en la nota.~~ Por cada día de atraso en la entrega de la tarea se descontarán 5 puntos en la nota.
- El plazo máximo de entrega es ~~5 horas~~ 3 días después de la fecha original de entrega.
- **Las tareas que no compilen no serán revisadas y serán calificadas con nota 0**.
- Debe usar **obligatoriamente** alguna de las formas de compilación indicadas en los tutoriales entregados en Aula USM.
- Por cada *Warning* en la compilación se descontarán 5 puntos.
- Si se detecta **COPIA** la nota automáticamente sera 0 (CERO), para todos los grupos involucrados. El incidente será reportado al jefe de carrera.
- La prolijidad, orden y legibilidad del código fuente es obligatoria. Habrá descuentos si alguno de estos items no se cumple.

6. Consejos de Programación

El código fuente del programa debe estar estructurado adecuadamente en archivos (separados de ser necesario). Si el código fuente está desordenado, se pueden descontar hasta 20 puntos de la nota.

Cada función programada debe tener comentarios de la siguiente forma:

```
/*****
*   TipoFunción NombreFunción
******
*   Resumen Función
******
*   Input:
*       tipoParámetro NombreParámetro : Descripción Parámetro
*        .......
******
*   Returns:
*       TipoRetorno, Descripción retorno
*****/
```

Por cada comentario faltante, se restarán 5 puntos.

Por último, la indentación (1 TAB o 4 espacios), es muy importante. Por **cada bloque mal indentado, se quitarán 10 puntos**.