UFV Universidad Francisco de Vitoria

PRÁCTICA 2

- Ordenación y Búsqueda
- Tablas Hash
- Least Recently Used cache
- Transición del lenguaje de programación C a Python

ESTRUCTURA DE DATOS Y ALGORITMOS – Curso 2024/2025
GRADO en Ingeniería Matemática

01. NORMAS GENERALES

- El código deberá mostrar modularidad, calidad, legibilidad y hacer un uso apropiado de las estructuras adecuadas, así como incluir comentarios.
- Se deberá codificar los ejercicios en lenguaje de programación **Python**.
- **La práctica se realizará en equipos de 2**, que deben quedar constituidos en Canvas para simplificar la entrega por parte de cualquiera de sus miembros.
- Un estudiante del equipo deberá subir a Canvas la práctica realizada. La subida al espacio habilitado podrá realizarse hasta el **01/12/2024 – 23:55.**
- Se deberá subir únicamente un fichero comprimido con extensión ".rar" / ".zip", nombrado del siguiente modo: **`PR2_Nombre1_Nombre2.rar`**. A modo de ejemplo se presenta este nombrado: **`PR2_Maria_Jimena.rar`**. El nombre y el apellido del alumno no deberá contener acentos al nombrar los ficheros.
- El fichero rar/zip contendrá:
  - Ficheros de código, nombrados siguiente modo: **`PR2_P1.py`**, …
  - Únicamente incluir fuentes con extensión ".py".

- **IMPORTANTE**: Aquellos ejercicios cuyos ficheros no cumplan la normativa de nombrado se calificarán con 0 puntos. También aquellos que no ejecuten.
- **IMPORTANTE**: Para cada ejercicio incluir un comentario al principio del código indicando claramente los detalles de sobre el autor/autores(equipo), así como una explicación clara sobre cómo se ha resuelto con las decisiones tomadas para realizarlo. Esta explicación debe ser completa y exhaustiva. Ejemplo:

```
'''
Estructura de Datos y Algoritmos | Ing. Matemática | Curso 24/25
PRÁCTICA 2 – Ordenación y Búsqueda. Tablas Hash. Caches LRU
AUTORES:
 - Nombre1 Apellido11 Apellido12
 - Nombre2 Apellido21 Apellido22
EJERCICIO: XX
EXPLICACIONES:
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aliquam
sed aliquam velit. Aliquam vehicula consequat porttitor. Morbi at
commodo nisl. Donec sit amet viverra lectus, id facilisis ligula.
'''
```

02. OBJETIVO

El objetivo de esta segunda práctica es ayudar a los estudiantes a comprender la utilidad y funcionamiento de los diferentes métodos de ordenación (directos e indirectos), de las posibilidades que ofrece la búsqueda binaria frente a la iterativa, así como ser capaz de implementar y aprovechar la utilidad de las tablas hash. Estas se aplicarán en un caso concreto, el de una caché LRU (Least Recently Used cache)

Adicionalmente, permitirá seguir practicando con Python, el nuevo lenguaje de Programación al que migramos desde C en la asignatura.

03. ENUNCIADO

3.1. PR2_P1.py (3.33 puntos)

En este ejercicio se trabajará con una lista de Python que contiene información de diferentes **piezas de fábrica**. Cada pieza tiene los siguientes atributos:

- **`id_pieza`**: Identificador único de la pieza (entero).
- **`precio`**: Precio de la pieza en euros (entero).
- **`unidades`**: Cantidad de unidades en inventario (entero).
- **`nombre_pieza`**: Nombre descriptivo de la pieza (cadena de texto).

Importante: No se permite utilizar el método **`list.sort()`** de las listas de Python. En su lugar, deberán implementar sus propios métodos de ordenación.

Requerimientos:

1. **Creación de Datos**:
   - Crear una clase `Pieza` que contenga los atributos mencionados (`id_pieza, precio, unidades, nombre_pieza`).
   - Crear una lista de objetos `Pieza` mediante un script. Deben generarse datos aleatorios, extensos (1000 piezas) y variados.
2. **Métodos de Ordenación**:
   - Implementar dos métodos de ordenación vistos en clase: **Selección** y **Quicksort**.
   - Utilizar estos métodos de ordenación en dos casos:
     - Ordenar (directo) la lista de piezas por **`precio`**.
     - Ordenar (indirecto) la lista nuevamente por **`id_pieza`**.
   - Medir el tiempo de ejecución de la ordenación y mostrarlo. Sugerencia: usar un decorador para esta medición **(*).**
3. **Búsqueda Binaria vs Secuencial**:
   - Solicitar al usuario que ingrese un **`id_pieza`** y aplicar la búsqueda binaria sobre la lista ordenada para encontrar la pieza que tenga exactamente esa cantidad de unidades. También realizar la búsqueda secuencial para buscar dicha pieza.
   - En caso de éxito, mostrar toda la información de la pieza; en caso contrario, hay que indicar que no se encontró el valor deseado.
   - Mostrar al usuario el número de consultas realizadas, tanto en la búsqueda binaria como la secuencial.
   - Medir el tiempo de ejecución de la búsqueda y mostrarlo. Sugerencia: usar un decorador para esta medición **(*).**

(*) Medición de Tiempos de Ejecución:

- Implementar un decorador que mida el tiempo de ejecución de cada algoritmo de ordenación y de la búsqueda binaria.
- Comparar los tiempos de los métodos de ordenación para analizar cuál es más eficiente en cada caso.
- Sugerencia para medir el tiempo de ejecución (cuidado con las funciones recursivas, requieren una función de "envoltura" o wrapper): https://ellibrodepython.com/tiempo-ejecucion-python

3.2. PR2_P2.py (3.33 puntos)

Codifica una clase **`TablaHashDNIsV1`** que pueda insertar al menos 50 DNIs siguiendo las tres estrategias de resolución de colisiones de direccionamiento abierto:

- Acordar previamente la estructura de la clase y las funciones Hash a utilizar.
- Implementar los siguientes 6 métodos sin modificar el nombre:

    1. **`insertarLineal`**
    2. **`funcionHashLineal`**
    3. **`insertarCuadratico`**
    4. **`funcionHashCuadratica`**
    5. **`insertarDobleHasing`**
    6. **`funcionHashDoble`**

Escribir una clase **`Tester1`** que prueba la inserción generando 50 DNIs de forma aleatoria utilizando el método `random()`.

A partir de la clase **`TablaHashDNIsV1`** realizar las modificaciones en una nueva clase **`TablaHashDNIsV2`**. También se requerirá crear una clase **`Celda`** que almacene el valor a guardar y el estado de la celda (`ocupada, vacía, borrada`) en lugar de almacenar el valor directamente.

- Añadir un método para buscar la posición de un elemento dado
- Añadir un método para borrar un elemento dado
- Modificar el método que inserta con DobleHashing para que primero compruebe si se va a superar el umbral y en ese caso duplicar el array (pensar manera de acotar tamaño de la lista, al no tener arrays como tal en Python).

Escribir una clase **`Tester2`** que prueba la inserción generando 100 DNIs de forma aleatoria utilizando el método **`random`**`()` y partiendo de una primera TablaHash de 25 elementos, con un umbral del 80%

3.3. PR2_P3.py (3.33 puntos)

Diseña una estructura de datos que siga las restricciones de una caché LRU (Least Recently Used). Implementa la clase LRUCache que tendrá los siguientes métodos:

- **`LRUCache(int capacidad):`** Inicializa la caché LRU con un tamaño positivo de capacidad. [Se trata del constructor de la clase]
- **`int obtener(int clave):`** Devuelve el valor de la clave si esta existe en la caché; de lo contrario, devuelve -1.
- **`void poner(int clave, int valor):`** Actualiza el valor de la clave si ya existe en la caché. De lo contrario, agrega la pareja clave-valor a la caché. Si el número de claves excede la capacidad después de esta operación, elimina la clave menos recientemente utilizada.

Funciones obtener/poner deben ejecutar con complejidad temporal promedio de O(1).

Consejos para Resolver el Problema

Una caché LRU (Least Recently Used) es un tipo de estructura de datos en la cual los elementos se organizan de tal manera que el elemento menos recientemente utilizado se elimina cuando se supera la capacidad máxima. Esto es útil en aplicaciones donde el almacenamiento es limitado y queremos retener solo los datos que se usan con frecuencia.

Para implementar la caché LRU, se pueden utilizar las siguientes estructuras:

- **Diccionario (HashMap):** Para almacenar los pares clave-valor y poder acceder a los valores en O(1).
- **Lista doblemente enlazada (Doubly Linked List):** Para organizar el orden de acceso de los elementos. La clave de esta lista es que permite mover elementos al inicio o al final de la lista en O(1), lo cual es útil para identificar y eliminar el elemento menos recientemente utilizado.

La implementación típica de una caché LRU mantiene el elemento más reciente en la cabeza de la lista y el menos reciente en la cola. Cuando se realiza una operación obtener, se mueve el elemento accedido al inicio de la lista. En una operación poner, se agrega o actualiza el elemento y, si la capacidad se excede, se elimina el elemento de la cola.

Necesitarás diseñar una clase Nodo.

Probar diferentes llamadas poner/obtener para validar el correcto funcionamiento.