CS2023 - Aula de Ejercicios Nº 3

Angel Napa
ACL: Joaquín Jordán
Semestre 2024-1

Se sugiere que cada estudiante trate de resolver los ejercicios de forma **individual** y luego los discuta en grupo.

IMPORTANTE:

- Enviar en canvas un **único archivo .cpp** (colocar las 3 soluciones en el mismo archivo .cpp).
- **No se revisaran archivos .zip** subidos a canvas.
- **Colocar nombres y apellidos** de los integrantes **al inicio** del archivo .cpp.
- La evaluación continua es grupal (mínimo 2 alumnos y máximo 3 alumnos).

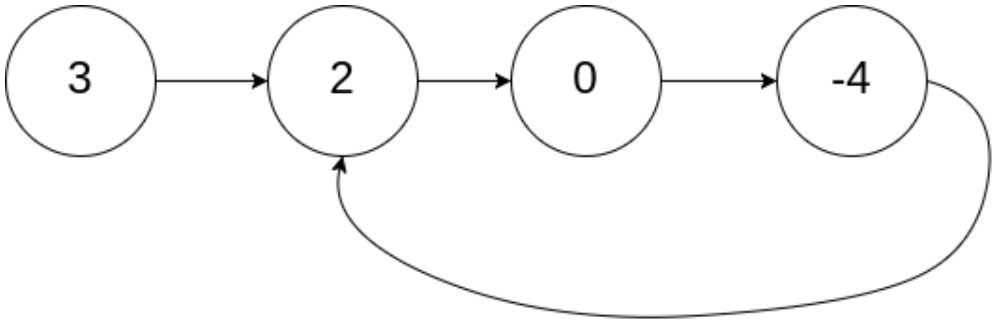
Ejercicios

1. (7 pts) Dada el `head` de una lista enlazada, devuelve el nodo donde comienza el ciclo. Si no hay ciclo, devuelve nulo.

   Hay un ciclo en una lista enlazada si hay algún nodo en la lista al que se puede acceder nuevamente siguiendo continuamente el siguiente puntero. Internamente, `pos` se usa para indicar el índice del nodo al que está conectado el siguiente puntero de la cola (indexado 0). Es -1 si no hay ciclo. Tenga en cuenta que `pos` no se pasa como parámetro.

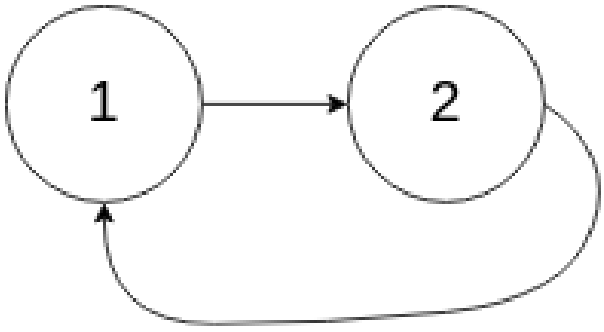
   No modifique la lista enlazada.

   - **Ejemplo 1:**



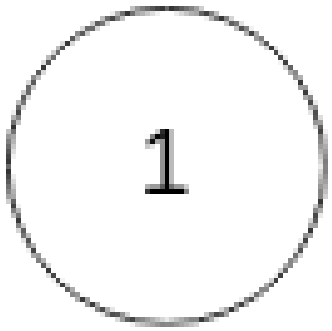
   Input: `head = [3,2,0,-4]`
   Output: `tail connects to node index 1`

   - **Ejemplo 2:**



Input: `head = [1,2]`
Output: `tail connects to node index 0`

- **Ejemplo 3:**



Input: `head = [1]`
Output: `no cycle`

2. (10 pts) Diseñar e implementar una estructura de datos para una caché de uso menos frecuente (Least Frequently Used Cache).

   Implemente la clase LFUCache:

   - LFUCache(int capacity) Inicializa el objeto con la `capacity` de la estructura de datos.
   - int get(int key) Obtiene el valor de `key` si la `key` existe en el caché. De lo contrario, devuelve -1.
   - void put(int key, int value) Actualiza el valor de la `key` si está presente, o inserta la `key` si aún no está presente. Cuando el caché alcanza su `capacity`, debe invalidar y eliminar la `key` utilizada con menos frecuencia antes de insertar un nuevo elemento. Para este problema, cuando hay un empate (es decir, dos o más `key` con la misma frecuencia), se invalidará la `key` utilizada menos recientemente.

   Para determinar la `key` utilizada con menos frecuencia, se mantiene un contador de uso para cada `key` en la caché. La `key` con el contador de uso más pequeño es la `key` que se usa con menos frecuencia.

   Cuando se inserta una `key` por primera vez en el caché, su contador de uso se establece en 1 (debido a la operación de colocación). El contador de uso de una `key` en el caché se incrementa ya sea que se llame a una operación get o put.

   Las funciones get y put deben ejecutarse cada una con una complejidad de tiempo promedio O(1).

   Se evaluará la solución en alto nivel, use texto o diseños gráficos para explicar su solución. El código de abajo es apenas referencial.

```
class LFUCache {
public:
    LFUCache(int capacity) {

    }

    int get(int key) {

    }

    void put(int key, int value) {

    }
```

```
};

/**
 * Your LFUCache object will be instantiated and called as such:
 * LFUCache* obj = new LFUCache(capacity);
 * int param_1 = obj->get(key);
 * obj->put(key,value);
 */
```

- **Ejemplo 1:**
  Input (lista de operaciones de ejemplo, no input real): `["LFUCache(capacity: 2)", "put(key: 1, value: 1)", "put(key: 2, value: 2)", "get(key: 1)", "put(key: 3, value: 3)", "get(key: 2)", "get(key: 3)", "put(key: 4, value: 4)", "get(key: 1)", "get(key: 3)", "get(key: 4)"]`
  Output (no se debe imprimir, solo desarrollar la estructura): `[1, -1, 3, -1, 3, 4]`
  Explicación:

```
// cnt(x) = the use counter for key x
// cache=[] will show the last used order for tiebreakers (leftmost element is  most recent)
LFUCache lfu = new LFUCache(2);
lfu.put(1, 1);   // cache=[1,\_], cnt(1)=1
lfu.put(2, 2);   // cache=[2,1], cnt(2)=1, cnt(1)=1
lfu.get(1);      // return 1
                 // cache=[1,2], cnt(2)=1, cnt(1)=2
lfu.put(3, 3);   // 2 is the LFU key because cnt(2)=1 is the smallest, invalidate 2.
                 // cache=[3,1], cnt(3)=1, cnt(1)=2
lfu.get(2);      // return -1 (not found)
lfu.get(3);      // return 3
                 // cache=[3,1], cnt(3)=2, cnt(1)=2
lfu.put(4, 4);   // Both 1 and 3 have the same cnt, but 1 is LRU, invalidate 1.
                 // cache=[4,3], cnt(4)=1, cnt(3)=2
lfu.get(1);      // return -1 (not found)
lfu.get(3);      // return 3
                 // cache=[3,4], cnt(4)=1, cnt(3)=3
lfu.get(4);      // return 4
                 // cache=[4,3], cnt(4)=2, cnt(3)=3
```

3. (9 pts) Se le proporciona una serie de k listas enlazadas, cada lista enlazada está ordenada en orden ascendente. Fusione todas las listas vinculadas en una lista vinculada ordenada y devuélvala.

   - **Ejemplo 1:**
     Input: `lists = [[1,4,5],[1,3,4],[2,6]]`
     Output: `[1,1,2,3,4,4,5,6]`
     Explicación: Las listas son:

```
[
  1->4->5,
  1->3->4,
  2->6
]
```

     fusionandolas en una sola: 1->1->2->3->4->4->5->6
   - **Ejemplo 2:**
     Input: `lists = []`
     Output: `[]`

- **Ejemplo 3:**
  Input: `lists = [[]]`
  Output: `[]`