Trabajo Práctico Final:
Caché LRU

Estructuras de Datos
Tecnicatura en Programación Informática
Universidad Nacional de Quilmes

1. Introducción

Un *caché LRU* es un tipo especial de map con un tamaño fijo. Una vez lleno, al insertar un par (clave,valor) se "hace lugar" eliminando el par (clave,valor) menos recientemente accedido. La idea es mantener a mano los elementos utilizados frecuentemente, sacrificando aquellos accedidos de manera esporádica. En efecto, las siglas LRU significan *Least Recently Used*, esto es, "menos recientemente accedido" en inglés.

2. Caché LRU

Una manera de implementar un caché LRU es combinando un *map* y una *lista doblemente enlazada*. Para el trabajo práctico utilizaremos una variación de las listas doblemente enlazadas pedidas como entrega y maps implementados utilizando hashing, tal como fueran presentados en la práctica. En la Figura 1 podemos ver representado mediante una lista y un map un caché LRU mapeando strings a enteros con tres pares: $(a, 0)$, $(b, 1)$ y $(c, 2)$.

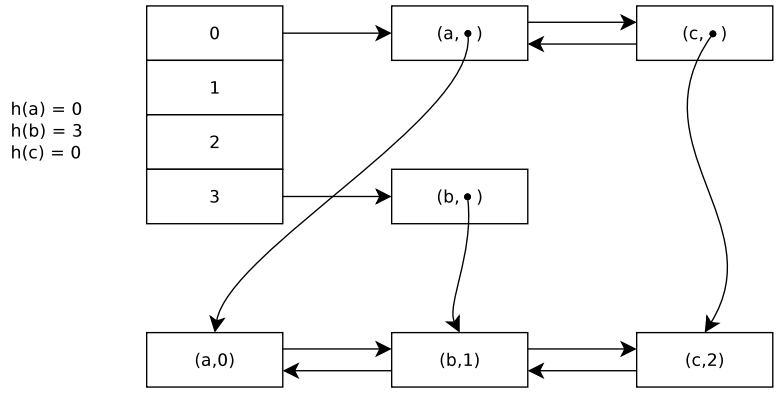

Figura 1: caché LRU modelado mediante hashing y listas doblemente enlazadas

En parte superior de la figura tenemos la tabla hash. Dicha tabla modela un map cuyo dominio es el conjunto de claves del caché LRU (en el ejemplo, $\{a, b, c\}$) y sus valores son *punteros* a los nodos de una lista doblemente enlazada que contiene los pares (clave,valor) del caché LRU. El aspecto principal de esta representación es el siguiente invariante:

> *Los pares de la lista están ordenados cronológicamente, de más recientemente accedido (el head de la lista) a menos reciementemente accedido (el último elemento de la lista).*

Con esta representación, la implementación de las operaciones principales que pediremos para el trabajo (*add* y *get*) son:

- **add:** Sea (k, v) el par a agregar. Lo primero es verificar si el caché está lleno. Si lo está, debemos eliminar el elemento menos reciemente accedido. Gracias al invariante de representación, sabemos que es el último de la de lista. Sea (k_u, v_u) dicho elemento. Luego, debemos borrar la clave k_u y su valor asociado de la tabla hash, y el par (k_u, v_u) de la lista. Finalmente, podemos insertar (k, v) agregando dicho par al principio de la lista, y agregando el par (k, p) a la tabla hash, donde p es el puntero al primer nodo de la lista.
- **get:** Sea k la clave a buscar en el caché. Buscamos el valor asociado a k en la tabla hash. Si lo encontramos, tenemos entonces el puntero al nodo correspondiente en la lista. Para mantener el invariante, debemos eliminar dicho nodo de la lista y agregarlo al principio. Finalmente, actualizamos el valor asociado a k en la tabla hash, de modo que ahora apunte el primer nodo de la lista. Terminamos retornando el valor asociado a k.

Notar que nuestra implementación es muy eficiente: todas las operaciones sobre listas requeridas por **add** y **get** son $O(1)$ si la lista es doblemente enlazada, y asumiendo una buena función de hash, las operaciones sobre la tabla hash son eficientes también.

3. Implementación

Para nuestra implementación de caché LRU utilizaremos las listas doblemente enlazadas pedidas como entrega y tablas hash, cuya implementación fue distribuida por mail a la lista de alumnos de ED. Sin embargo, introduciremos una variación importante: utilizaremos la listas doblemente enlzadas con propósito *doble*:

1. Para representar la lista de elementos del caché ordenados de más recientemente accedido a menos recientemente acceido, como describimos en la Secc. 2.
2. Para implementar hashing abierto. Notar que para la colisión de las claves a y c en la Fig. 1 tenemos una lista doblemente enlazada.

Por lo tanto, nuestras listas doblemente enlazadas deberán ser *polimórficas*: por un lado, deben almacenas pares (clave,valor). Por otro, deben poder almacenar también pares (clave, puntero a nodo de la lista). A su vez, no es deseable que la tabla hash pueda "ver" los nodos de una lista doblemente enlazada, la listas son un tipo abstracto. Por lo tanto, debemos introducir una *abstracción* para el concepto de puntero a un nodo de la lista. Lamaremos a dicha abstracción un *cursor*.

En lo que resta de esta sección profundizaremos en las variaciones requeridas para los tipos abstractos de listas doblemente enlazadas y tabla hash. Comencemos con los aspectos de implementación preliminares.

3.1. Aspectos Preliminares

Los pares clave valor almacenados por nestro caché serán de tipo (string,entero). Será conveniente tener un tipo que modele dichos pares.

Ejercicio 1. Definir un módulo **types** (o sea, los archivos **types.h** y **types.cpp**) definiendo pares clave valor donde la clave es un string y los valores son enteros.

Ayuda: Sugerimos la siguiente interfaz para el módulo:

```
typedef std::string key_t;
typedef int val_t;

typedef struct {
  key_t key;
  val_t val;
} elem_t;

elem_t elem(key_t key, val_t val);
std::ostream& operator<<(std::ostream& o, const elem_t& x);
```

El tipo `elem_t` representa un par (string,entero). La operación `elem` construye un par. La última operación simplemente hace que un valor `x` de tipo `elem_t` se pueda imprimir de la manera usual en C++: `std::cout << x`. Esto será útil para debuggear el código. Consultar con la cátedra cómo implementar el operador `<<` (es *muy* sencillo).

3.2. Listas Doblemente Enlazadas

Lo primero que debemos hacer es transformar a la implementación de listas doblemente enlazadas entregada en polimórfica. El lenguaje C++ soporta polimorfismo paramétrico mediante *templates*. Los detalles sobre cómo trabajar con templates serán dados en la práctica, y cualquier duda debe ser consultada con los ayudantes. A grandes rasgos, la implementación de listas debe pasar al archivo .h (descartando el archivo .cpp) con la siguiente signatura:

```
template<typename T> struct Node {
  T x;
  Node* prev;
  Node* next;
};

template<typename T> struct List {
  Node<T>* first;
  Node<T>* last;
  int len;
};

template<typename T> List<T> empty();
template<typename T> bool isEmpty(const List<T>& xs);
template<typename T> int len(const List<T>& xs);
template<typename T> void cons(T x, List<T>& xs);
template<typename T> void snoc(List<T>& xs, T x);
template<typename T> T head(const List<T>& xs);
template<typename T> T last(const List<T>& xs);
template<typename T> void tail(List<T>& xs);
template<typename T> void init(List<T>& xs);
template<typename T> void destroy(List<T>& xs);
template<typename T>
  std::ostream& operator<<(std::ostream& o, const List<T>& xs);
```

La idea es que las listas pueden ser de un tipo cualquiera `T`. Notar que este signatura es sólo un ejemplo. Una alternativa válida podría ser trabajar con punteros en lugar de referencias (por ejemplo, podríamos tener a `empty` retornando el tipo `List<T>*` en lugar de `List<T>`). Lo que *sí* es necesario es que la interface provea todas las operaciones listadas.

Ejercicio 2. Transformar la implementación de listas doblemente enlazada pedida como entrega en polimórfica mediante el uso de templates. No dudar en consultar con la cátedra cualquier duda sobre este ejercicio.

La próxima modificación requerida sobre listas será proveer *cursores*. Un cursor es una abstracción sobre un puntero a un nodo de una lista doblemente enlazada. Será conviente para implementar hashing y también el caché LRU. Básicamente, un cursor es simplemente una estructura conteniendo un puntero a un nodo de una lista, más la lista (esta última es necesaria porque algunas operaciones sobre cursores requerirán modificar los campos `first` y `last` del tipo `List`):

```
template<typename T> struct Cursor {
  List<T>* xs;  // Puntero a la lista completa
  Node<T>* n;   // Puntero al nodo representado por el cursor
};
```

Nuestro tipo cursor debe proveer las siguientes operaciones:

```
template<typename T> Cursor<T> cursor_head(List<T>& xs);
template<typename T> Cursor<T> cursor_last(List<T>& xs);
template<typename T> T cursor_elem(const Cursor<T>& c);
template<typename T> bool cursor_has_next(const Cursor<T>& c);
template<typename T> bool cursor_has_prev(const Cursor<T>& c);
template<typename T> void cursor_next(Cursor<T>& c);
template<typename T> void cursor_prev(Cursor<T>& c);
template<typename T> void cursor_update(Cursor<T>& c, T x);
template<typename T> void cursor_remove(Cursor<T>& c);
```

Donde:

- **cursor_head** retorna un cursor apuntando al primer elemento de `xs`.
- **cursor_last** retorna un cursor apuntando al último elemento de `xs`.
- **cursor_has_next** retorna si el cursor actual puede avanzar al siguiente nodo de la lista. Esta operación debe retornar `true` incluso para el último nodo. En este caso, al avanzar una posición el puntero `n` del tipo `Cursor` será `NULL`, en cuyo caso `cursor_has_next` sí debe retornar `false`.
- **cursor_has_prev** retorna si el cursor actual puede retroceder al nodo anterior de la lista. Esta operación debe retornar `true` incluso para el primer nodo. En este caso, al retroceder una posición el puntero `n` del tipo `Cursor` será `NULL`, en cuyo caso `cursor_has_prev` sí debe retornar `false`.
- **cursor_next** avanza el cursor `c` al siguiente nodo de la lista.
  Precondición: `cursor_has_next(c)` y `cursor_has_prev(c)` deben ser `true`.
- **cursor_prev** avanza el cursor `c` al siguiente nodo de la lista.
  Precondición: `cursor_has_next(c)` y `cursor_has_prev(c)` deben ser `true`.
- **cursor_elem** devuelve el elemento del nodo apuntado por el cursor `c`.
  Precondición: `cursor_has_next(c)` y `cursor_has_prev(c)` deben ser `true`.
- **cursor_update** Actualiza el elemento del nodo apuntado por el cursor `c`.
  Precondición: `cursor_has_next(c)` y `cursor_has_prev(c)` deben ser `true`.
- **cursor_remove** Borra de la lista el nodo apuntado por el cursor `c`.
  Precondición: `cursor_has_next(c)` y `cursor_has_prev(c)` deben ser `true`.
  Postcondición: el cursor `c` no se puede volver a utilizar después de invocar esta operación.
  Nota: Esta operación puede requerir modificar los campos `first` y `last` de la lista. Es por eso que el tipo `Cursor` tiene el puntero a la lista completa.

Ejercicio 3. Agregar a las listas doblemente enlazadas una implementación de cursores. La implementación debe hacer en el archivo .h donde están implementadas las listas polimórficas.

Ejercicio 4. Probar la implementación de listas. Las pruebas deben cubrir las operaciones de listas y las nuevas sobre cursores. Dado que las implementaciones de hashing y del caché LRU utilizarán listas de manera extensiva, es *fundamental* estar seguros de contar con una implementación de listas correcta antes de avanzar.

3.3. Hashing

La tabla hash requerida para el caché LRU debe asociar claves de tipo string a punteros a nodos de la lista, o sea, cursores apuntando a nodos cuyos valores son pares (string,entero). Recordar que dichos pares están modelados por el tipo `elem_t` (ver Secc. 3.1). Por lo tanto, comenzaremos la adaptación de la tabla hash distribuida por la lista de mail definiendo un tipo modelando un par (string,cursor):

```
// elem_t y key_t definidos en types.h, Cursor en list.h
struct _pair {
  key_t key;
  list::Cursor<elem_t> val;
};
```

Por otra parte, utilizaremos nuestro tipo polimórfico de listas doblemente enazadas para implenentar hashing abierto. Por lo tanto, nuestro tipo tabla hash pasará a ser algo similar a:

```
struct HashTbl {
  List<_pair>* t;  // Array de listas de tipo _pair
  int tsize;
};
```

Ejercicio 5. Adaptar la tabla hash distrbuida por la lista de mail de ED para que asocie strings a cursores, utilizando el tipo de listas doblemente enlazadas polimórficas para implementar hashing abierto. Una posible interface es la siguiente:

```
HashTbl empty(const int size);
bool isEmpty(const HashTbl& t);
int len(const HashTbl& t);
void add(HashTbl& t, key_t k, Cursor<elem_t> c);
bool has(const HashTbl& t, key_t k);
Cursor<elem_t> get(const HashTbl& t, key_t k);
void del(HashTbl& t, key_t k);
void destroy(HashTbl& t);
```

Nota: si bien la implementación tiene que incluir las operaciones mencionadas, los tipos mencionados (incluyendo `HashTbl`) son sólo un ejemplo. Al igual que con las listas, es posible reemplazar referencias por puntertos. Notar también que algunas operaciones (por ejemplo, `empty`) son homónimas con operaciones del tipo lista. Pueden evitar el solapamiento utilizando namespaces, o renombrando operaciones en algunos de los dos módulos.

Ejercicio 6. Probar todas las operaciones de la nueva implementación de hashing. Al igual que con las listas, es *fundamental* que las pruebas sean lo suficientemente exhaustivas como para estar seguros que la implementación es correcta. Esto es muy importante, dado que utilizaremos hashing para el caché LRU.

3.4. Caché LRU

El paso final consiste en implementar el módulo caché LRU. La idea es implementar las operaciones **add** y **get** tal como fueron descriptas en la Secc. 2, siendo usuario de los tipos abstractos lista doblemente enlazada y hash table. Concretamente:

Ejercicio 7. Implementar el tipo abstracto **LRUCache** en los archivos **lru.h** y **lru.cpp**; y las siguientes operaciones (todas triviales, excepto **add** y **get**):

```
LRUCache empty(const int maxSize);
bool isEmpty(const LRUCache& c);
int len(const LRUCache& c);
void add(LRUCache& c, elem_t x);
bool has(const LRUCache& c, key_t k);
elem_t get(LRUCache& c, key_t k);
void destroy(LRUCache& c);
std::ostream& operator<<(std::ostream& o, const LRUCache& c);
```

Las operaciones son las mismas que las de un map. En efecto, un caché LRU puede verse como un map de tamaño fijo. El parámetro `maxSize` de la operación `empty` denota el tamaño máximo del caché. La inserción de un elemento (`add`) sobre un caché cuyo tamaño (`len`) es igual a `maxSize` forzará el borrado del elemento menos recientemente accedido. Sugerimos que el operador `<<` simplemente imprima la lista de pares (clave,valor); esto es, el print de un caché LRU debe invocar al print de su lista.

Ejercicio 8. Probar la implementación de todas las operaciones del tipo `LRUCache`.

4. Consejos

C++ es un lenguaje muy poderoso, pero también complejo y con el que es fácil equivocarse. Nuestras sugerencias para evitar y/o minimizar dificultades son:

- **No procastinar:** Si empezás el trabajo tarde será difícil que lo completes, y de hacerlo, no va a ser para nada fácil. ¡Empezá con tiempo!
- **No dudar en consultar:** No dejes que el trabajo se transforme en una experiencia frustrante. Ante la menor complicación, consultá con la cátedra.
- **Probar código seguido:** El trabajo está estructurado de modo que cada ejercicio es usado por los ejercicios subsiguientes. Cualquier error temprano sin detectar va a ser una complicación. Deberías probar que *cada* ejercicio está resuelto correctamente antes de avanzar al próximo.