Tablas dispersas.

Prof. Isabel Pita

Explicación de las transparencias de clase

Código desarrollado por el profesor A. Verdejo para la asignatura de Estructuras de Datos de la Facultad de Informática de la UCM.

5 de mayo de 2020

1. Tablas dispersas.

- Las tablas dispersas se utilizan para implementar diccionarios en los que el orden entre las claves no es importante.
- Las tablas dispersas almacenan los pares $\langle$ clave, valor $\rangle$ en un vector de tamaño N. El tamaño del vector N es un número primo. Se selecciona así para evitar colisiones en la tabla.
- Cada clave c se convierte en un índice válido del vector a través de una *función de dispersión*, $h(c)$. La función asocia a cada posible valor de la clave un índice del vector.

  El dominio de las claves (valores posibles de las claves) puede ser mucho mayor que el número de índices del vector, por lo tanto la función asignará a varias claves el mismo índice. Esto no debe preocuparnos, mas adelante veremos varias formas de resolver este problema.

  Por ejemplo, tenemos una aplicación que debe hacer un estudio sobre las personas de una provincia española. Se utilizará un diccionario con clave el NIF de la persona. El número de NIFs posibles es muy grande, sin embargo el número real de claves que debemos guardar será como máximo el número de personas de esa provincia, que es mucho menor que el total de NIFs.

- La función de dispersión debe ser fácil de calcular y debe distribuir las claves de manera uniforme entre las posiciones del vector. Cuando dos claves cumplen $h(c_1) \equiv h(c_2)$ se dice que son *sinónimas*. Se produce una *colisión*.

  Para que el diccionario sea eficiente, el tamaño del vector debe se similar al número de claves utilizadas en la aplicación, en nuestro ejemplo el número de personas de la provincia, de forma que en media a cada persona se le asocie un índice. Pero esto no evita que la función de dispersión asocie a varias claves el mismo índice. La función de dispersión será mejor cuantas menos colisiones produzca, o sea cuanto mejor aproveche el espacio del vector.

La función de dispersión .

- Si la clave es un número, se puede utilizar tal cual.
- Si es una cadena, hay que convertirla en un número. Sumar el código ASCII de sus caracteres puede no dar buenos resultados. Mejor tener en cuenta la posición de los caracteres y obtener un número mucho mayor. Para una clave $c_0c_1c_2 \ldots c_{L-1}$ calcular $\sum_{i=0}^{L-1} c_{L-i-1}37^i$.

Esta función hash se implementaría como:

```
unsigned int hash(string const& clave) {
   unsigned int val = 0;
```

```
   // evaluacion de un polinomio por la regla de Horner
   for (char c : clave)
      val = val * 37 + c;
   return val;
}
```

Funciones de dispersión en la STL .

- La STL incluye (en la librería `functional`) una plantilla de funciones objeto para calcular funciones de dispersión.

  ```
  template <class Clave>
  class hash {
  public:
     size_t operator()(Clave const& c) const;
  };
  ```

  La plantilla está ya instanciada para muchos tipos comunes, incluyendo `std::string`. Por ejemplo, la implementación de la función hash para el tipo `std::string` podria ser asi:

  ```
  template <>
  class hash<std::string> {
  public:
     size_t operator()(std::string const& clave) {
        size_t val = 0;
        for (char c : clave)
        val = val * 37 + c;
       return val;
    }
  };
  ```

Tablas dispersas abiertas .

- Existen diversas formas de resolver las *colisiones*. Cada forma da lugar a un tipo de tabla hash.
- Estudiaremos las tablas dispersas abiertas. En estas tablas, el vector que forma la tabla mantiene en cada componente una lista y los pares de claves sinónimas se colocan en ella.

- El *factor de carga*, λ, de una tabla es la relación entre el número de pares que contiene y su tamaño N. Se recomienda que el factor de carga de una tabla no supere el 0.8, esto supone que cada lista en media tiene menos de un elemento.

- La función que busca el valor asociado a una clave en la tabla se implementa siguiendo los siguientes pasos:

  1. Se calcula la función hash (función de dispersión) de la clave. Esto nos da un número entero positivo.
  2. Accedemos a la posición del vector dada por la función hash.
  3. Se recorre la lista asociada a esa posición.
  4. Si se encuentra la clave, se devuelve el valor asociado.
  5. Si no se encuentra la clave, el valor no está en la tabla.
  6. El coste de la operación buscar en el caso peor es del orden de la longitud de la lista en la que hay que buscar el elemento.

- La función que añade un par $\langle$ clave, valor $\rangle$ a la tabla se implementa siguiendo los siguientes pasos:

  1. Se calcula la función hash (función de dispersión) de la clave. Esto nos da un número entero positivo.
  2. Accedemos a la posición del vector dada por la función hash.
  3. Se recorre la lista asociada a esa posición.
  4. Si se encuentra la clave, estamos intentando insertar una clave repetida.
  5. Si no se encuentra la clave, se inserta el par en la lista.
  6. El coste de la operación insertar en el caso peor es del orden de la longitud de la lista en la que hay que insertar el elemento.

- La función que elimina una clave y su valor asociado de una tabla es similar a la anterior, pero eliminando el elemento de la lista.

- Por lo tanto suponiendo que la función de dispersión es uniforme, la longitud media de una lista es λ, el coste de las operaciones de búsqueda, inserción o borrado es casi constante, suponiendo las tablas bien diseñadas.

Tablas hash en la STL. El tipo `unordered_map` .

La librería STL ofrece un TAD, el `unordered_map` que implementa las tablas hash.

```
template < class Key,                                    // unordered_map::key_type
           class T,                                      // unordered_map::mapped_type
           class Hash = hash<Key>,                       // unordered_map::hasher
           class Pred = equal_to<Key>,                   // unordered_map::key_equal
           class Alloc = allocator< pair<const Key,T> >  // unordered_map::allocator_type
           > class unordered_map;
```

Para declarar un `unordered_map` debemos dar obligatoriamente

- el tipo de la clave y
- el tipo del valor asociado.

Opcionalmente se puede dar:

- La función hash que se quiere aplicar sobre la clave. Esta función se definirá en un objeto función. Por defecto se asocia la función hash que la STL tiene definida sobre el tipo de las claves. En la asignatura utilizaremos siempre la función por defecto.
- Un predicado (función que tiene dos parámetros del tipo de la clave y devuelve un booleano) que define cuando dos claves son iguales. Por defecto se utiliza el operador `==` definido para el tipo de las claves.
- El tipo utilizado para almacenar la tabla. Por defecto se utiliza el que indica el sistema.

Ejemplo de como declarar una tabla hash:

```
#include <unordered_map>


int main() {
   std::unordered_map<std::string,int> tabla;
}
```

Las operaciones sobre las tablas hash son las mismas que sobre las tablas ordenadas (`map`). La única diferencia se encuentra en el uso del iterador que sobre las tablas hash recorre los elementos según estén guardados en la tabla, y por lo tanto sin ningún orden asociado.

El coste de las operaciones es prácticamente constante y por lo tanto son más eficientes que las correspondientes operaciones sobre el TAD `map`.

Cuando se debe utilizar cada tabla .

Las tablas hash(`unordered_map`) se deben utilizar para resolver aquellos problemas en los que se necesita *buscar* el valor asociado a una clave, *insertar* pares de clave y valor, y *borrar* una clave con su valor asociado, pero no se requiere recorrer la tabla en ningún orden especial.

Las tablas ordenadas (`map`) se deben utilizar cuando se requiera recorrer la tabla en algún orden especial respecto de las claves. El orden es siempre respecto a las claves, no respecto de los valores asociados.