Diccionarios.

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.

30 de abril de 2021

1. Diccionarios.

- Los diccionarios (también llamados tablas asociativas o estructuras funcionales) sirven para asociar valores a claves.
- Cada clave aparece (como mucho) una vez en el diccionario, y tiene asociado un único valor.
- Las operaciones típicas son la inserción de pares ⟨clave, valor⟩, la búsqueda del valor asociado a una clave, o el borrado de una clave y su valor asociado.

Los vectores nos permiten implementar diccionarios en los cuales las claves con número enteros entre el cero y un cierto valor máximo. Se pueden también utilizar si las claves son número enteros pertenecientes a un intervalo `[min..max]` realizando una traslación al intervalo `[0..max-min]`. ¿Pero, que pasa si queremos definir una clave que no sea un número entero?, ¿y si tenemos un intervalo de números enteros mayor que el rango que se puede almacenar en un array?, ¿Y si nuestros valores, aun siendo números enteros son muy dispersos?. En este tema veremos como implementar una estructura de datos eficiente que nos permita manejar diccionarios con cualquier tipo de clave.

Un ejemplo típico del uso de un diccionario es el *problema de las concordancias*. Este problema consiste en, dado un texto, contar el número de veces que aparece en él cada palabra.

Visto desde otra prespectiva, queremos asociar a cada palabra el número de veces que aparece en el texto.

Definiremos un diccionario en el que:

- Las claves serán las palabras. Aparecen una única vez en el diccionario.
- El valor asociado a cada clave es el número de veces que aparece la palabra en el texto.

Dada una palabra del texto, se *busca* si está en el diccionario. Si la palabra ya está en el diccionario, se modifica su valor asociado, si no está en el diccionario, se *inserta*.

Operaciones del TAD diccionario de la STL .

La STL nos proporciona dos implementaciones de los diccionarios `map` y `unordered_map`. El primero se utiliza si se necesita mantener un orden en las claves, y permite recorrer todos los valores en el orden que se especifique. La segunda no mantiene ningún tipo de orden y recorrerá los valores según los tiene almacenados en memoria. Se hace notar que no se almacenan por orden de llegada ni por ningún otro orden que pueda ser definido por el usuario. Ambas tienen las siguientes operaciones:

- Constructor de un diccionario vacío,
- `pair<iterator,bool> insert (const value_type& val);` insertar un par ⟨ clave, valor ⟩. El tipo `value_type` está definido como un `pair<key_type, mapped_type>` donde `key_type` y `mapped_type` son los parámetros de la clase que indican el tipo de la clave y el valor. Si la clave ya existe en el diccionario devuelve un iterador a la clave ya existente y el valor `false`. Si la clave se inserta devuelve un iterador al elemento insertado y el valor `true`.
- `size_type count (const key_type& k) const;` averiguar si una clave pertenece al diccionario. Devuelve uno si la clave pertenece al diccionario y cero si no pertenece.
- `iterator find (const key_type& k);` devuelve un iterador a la clave si esta existe. Si no existe devuelve un iterador al final del diccionario, `map::end()`
- `mapped_type& at (const key_type& k);` Devuelve una referencia al valor asociado a una clave. Si la clave no existe lanza una excepción del tipo `out_of_range`.
- `mapped_type& operator[] (const key_type& k);` Devuelve una referencia al valor asociado a una clave. Si la clave no existe la inserta en el diccionario. Esta es una forma alternativa de insertar elementos en un diccionario.

- `size_type erase (const key_type& k);` eliminar una clave y su valor asociado. Devuelve el número de elementos eliminados que debe ser 0 o 1. El tipo de retorno puede utilizarse como un valor booleano ya que en C cero es equivalente a `false` y uno es equivalente a `true`.
- `void erase (iterator position);` eliminar el par ⟨ clave, valor ⟩ apuntado por el iterador. Existe otra versión de la operación que recibe dos iteradores y elimina todas las claves y sus valores asociados entre los dos iteradores.
- `bool empty() const;` indica si el diccionario está vacío.
- `size_type size() const;` devuelve el número de elementos del diccionario.
- iteradores que nos permitan recorrer todos los pares ⟨ clave, valor ⟩. Existe un iterador constante, y uno no constante que permite modificar el valor asociado a las claves. Si el diccionario tiene orden, el iterador lo recorrerá según ese orden y en este caso se proporciona también un iterador inverso que recorre el diccionario en orden inverso.

Uso del map de la STL .

Los diccionarios con orden se implementan utilizando árboles de búsqueda equilibrados, por lo que las operaciones de buscar, insertar y borrar tienen coste logarítmico respecto al número de elementos del diccionario.

La clase `map` tiene tres parámetros, la clave, el valor que no tiene porque ser del mismo tipo que la clave y el comparador. Es obligatorio dar los dos primeros parámetros al crear un objeto de la clase. El tercer parámetro es opcional y en caso de no darse se utiliza el operador menor. El tipo `value_type` es un renombramiento de un par ⟨ clave, valor ⟩ y se utiliza en la operación `insert`:

```
template <class key_type,class mapped_type,class key_compare=less<key_type>> class map;

using value_type = pair<const key_type, mapped_type>;
```

Ejemplo. Vamos a calcular la cantidad aportada por una serie de personas a una hucha común. La misma persona puede añadir distintas cantidades en distintos momentos. Dada la lista de las diversas aportaciones obtendremos una lista ordenada alfabéticamente con las aportaciones totales de cada persona.

Para resolver el problema utilizaremos un diccionario (o tabla) con clave el nombre de la persona y valor la cantidad aportada. El tipo utilizado es:

```
using tabla = std::map<std::string, int>;
```

Se puede definir un orden diferente en la tabla utilizando un tercer parámetro.

```
using tabla = std::map<std::string, int, std::greater<std::string>>;
```

Vamos a ver distintas formas de utilizar las operaciones para añadir a la tabla las cantidades aportadas. En la primera solución utilizaremos el método `insert` para añadir los elementos. Se observa que el método `insert` si el valor ya existe en la tabla devuelve `false` y un puntero a dicho valor y no lo modifica. Podemos modificar el valor a través de este puntero.

```
bool resuelveCaso(){
    int numPersonas; std::cin >> numPersonas;
    if (numPersonas == 0) return false;
    // Lee los datos de las personas y los acumula en la tabla
    tabla aportaciones;
    for (int i = 0; i < numPersonas; ++i) {
        std::string nombre; int cantidad;
        std::cin >> nombre >> cantidad;
        auto result = aportaciones.insert({nombre,cantidad});
        if (!result.second)
            result.first->second += cantidad;
    }
    // Escribe los resultados
    for (auto n : aportaciones)
```

```
        std::cout << n.first << ' ' << n.second << '\n';
    std::cout << "---\n";
    return true;
}
```

Otra forma de resolver el problema utilizando el método `insert` es utilizar el método `count` para comprobar si el elemento ya existe en la tabla. Añadir el elemento si no está en la tabla y en otro caso con el método `at` modificar su valor.

```
bool resuelveCaso(){
    int numPersonas; std::cin >> numPersonas;
    if (numPersonas == 0) return false;
    // Lee los datos de las personas y los acumula en la tabla
    tabla aportaciones;
    for (int i = 0; i < numPersonas; ++i) {
        std::string nombre; int cantidad;
        std::cin >> nombre >> cantidad;
        if (aportaciones.count(nombre) == 0)
            aportaciones.insert({nombre,cantidad});
        else
            aportaciones.at(nombre) += cantidad;
    }
    // Escribe los resultados
    for (auto n : aportaciones)
        std::cout << n.first << ' ' << n.second << '\n';
    std::cout << "---\n";
    return true;
}
```

Una tercera forma de resolver el problema es utilizando la sobrecarga del operador corchete. Observamos que este operador nos permite acceder al valor de una clave y modificarla. Además si la clave no existe en la tabla la añade. Observar también que el operador corchete añade la clave si no existe, mientras que el método `at` lanza una excepción si el valor no existe.

```
bool resuelveCaso(){
    int numPersonas; std::cin >> numPersonas;
    if (numPersonas == 0) return false;
    // Lee los datos de las personas y los acumula en la tabla
    tabla aportaciones;
    for (int i = 0; i < numPersonas; ++i) {
        std::string nombre; int cantidad;
        std::cin >> nombre >> cantidad;
        aportaciones[nombre] += cantidad;
    }
    // Escribe los resultados
    for (auto n : aportaciones)
        std::cout << n.first << ' ' << n.second << '\n';
    std::cout << "---\n";
    return true;
}
```

Por último observamos el bucle basado en iterador que se ha utilizado para mostrar los resultados. Dicho bucle recorre el diccionario en inorden y por lo tanto las claves se obtienen ordenadas según el orden especificado al crear la tabla.