Diccionarios

Alberto Verdejo

Dpto. de Sistemas Informáticos y Computación
Universidad Complutense de Madrid

Bibliografía

- R. Peña. *Diseño de Programas: Formalismo y Abstracción*. Tercera edición. Pearson Prentice-Hall, 2005.
  Capítulo 8
- N. Martí Oliet, Y. Ortega Mallén y A. Verdejo. *Estructuras de datos y métodos algorítmicos: 213 ejercicios resueltos*. Segunda edición, Garceta, 2013.
  Capítulo 7
- M. A. Weiss. *Data Structures and Algorithm Analysis in C++*. Fourth edition. Pearson, 2014.
  Capítulos 4 y 5

Problema de las concordancias

El problema de las concordancias consiste en, dado un texto, contar el número de veces que aparece en él cada palabra, y producir una lista ordenada alfabéticamente por palabras, donde cada palabra aparece acompañada del número de veces que ha aparecido en el texto.

Problema de las concordancias

El problema de las concordancias consiste en, dado un texto, contar el número de veces que aparece en él cada palabra, y producir una lista ordenada alfabéticamente por palabras, donde cada palabra aparece acompañada del número de veces que ha aparecido en el texto.

Un Anillo para gobernarlos a todos
Un Anillo para encontrarlos
Un Anillo para atraerlos a todos
y atarlos en las tinieblas
en la Tierra de Mordor
donde se extienden las Sombras

```
a 2
anillo 3
atarlos 1
atraerlos 1
de 1
donde 1
en 2
encontrarlos 1
extienden 1
gobernarlos 1
...
```

Problema de las concordancias

```
void a_minusculas(string & s) {
    for (char & c : s)
        if ('A' <= c  && c <= 'Z')  // tolower(c) en <cctype>
            c = 'a' + (c - 'A');
}

void concordancias(istream & texto) {

    std::map<string, int> concor;

    // recuperar palabras contándolas
    string palabra;
    while (texto >> palabra) {
        a_minusculas(palabra);
        ++concor[palabra];
    }
    // mostrarlas
    for (auto const& cv : concor)
        cout << cv.first << ' ' << cv.second << '\n';
}
```

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 $\langle clave, valor \rangle$, la búsqueda del valor asociado a una clave, o el borrado de una clave y su valor asociado.
- En ocasiones las claves tienen orden y los pares se mantienen ordenados según su clave. Se pueden recorrer en orden.
- Los árboles binarios de búsqueda son una buena implementación de los diccionarios cuando las claves están ordenadas.
- Las tablas dispersas (*hash tables*) son una buena implementación cuando no es necesario el orden entre claves.

TAD de los diccionarios

El TAD de los diccionarios, `map<Clave,Valor>` o `unordered_map<Clave,Valor>`, cuenta con las siguientes operaciones:

- crear el diccionario vacío,
- insertar un par ⟨*clave*, *valor*⟩, `bool insert(clave_valor const& cv)`
- averiguar si una clave pertenece al dominio del diccionario, `int count(Clave const& c) const`
- consultar el valor asociado a una clave, si existe, `Valor const& at(Clave const& c) const`
- consultar y poder modificar el valor asociado a una clave, insertándola si no existe, `Valor & operator[](Clave const& c)`
- eliminar una clave, `bool erase(Clave const& c)`
- determinar si el diccionario está vacío, `bool empty() const`
- iteradores que nos permitan recorrer todos los pares ⟨*clave*, *valor*⟩

Implementación mediante ABBs

Los árboles binarios de búsqueda son una buena implementación de los diccionarios cuando las claves tienen orden y queremos poder recorrer los pares ⟨*clave*, *valor*⟩ en orden.

En cada nodo del árbol guardamos un par ⟨*clave*, *valor*⟩, y los nodos se mantienen ordenados por claves: para todo nodo, las claves en los nodos de su hijo izquierdo son menores y las claves en los nodos de su hijo derecho son mayores.

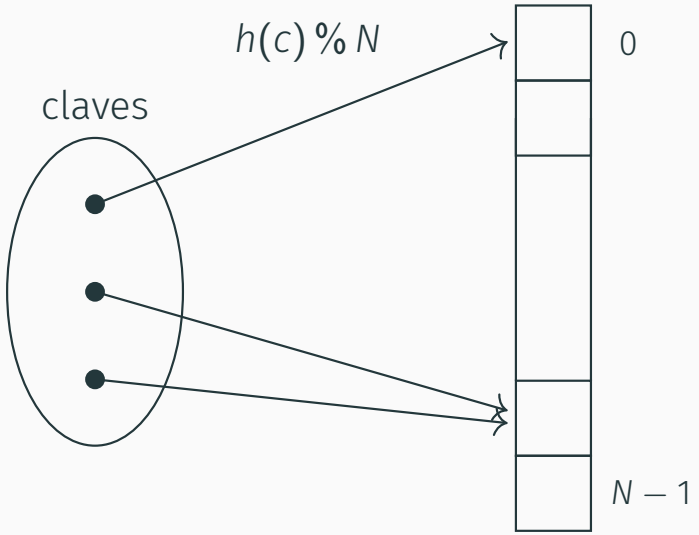
`treemap_eda.h`

std::map

```
template < class Key,                     // map::key_type
           class T,                       // map::mapped_type
           class Compare = less<Key>,     // map::key_compare
           > class map;

using value_type = pair<const Key, T>;

mapped_type      & at(key_type const& k);
mapped_type const& at(key_type const& k) const;

pair<iterator,bool> insert(value_type const& val);

iterator erase(const_iterator position);
size_t   erase(key_type const& k);
```

Tablas dispersas

Las tablas dispersas almacenan los pares $\langle clave, valor \rangle$ en un vector de tamaño N, primo. Cada clave c se convierte en un índice válido del vector a través de una función de dispersión, $h(c)$.



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 $h(c_1) \equiv_N h(c_2)$ se dice que son sinónimas. Se produce una colisión.

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 $c_0 c_1 c_2 \ldots c_{L-1}$ calcular $\sum_{i=0}^{L-1} c_{L-i-1} \cdot 37^i$.

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

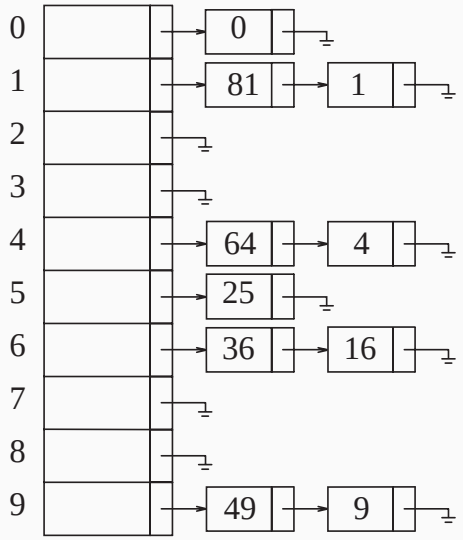
```
template <class Clave>
class hash {
public:
    size_t operator()(Clave const& c) const;
};
```

- La plantilla está ya instanciada para muchos tipos comunes, incluyendo `std::string`. Por ejemplo, podría ser así:

```
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

Una posibilidad para resolver las colisiones es colocar los pares con claves sinónimas en una misma lista.



El factor de carga, λ, de una tabla es la relación entre el número de pares que contiene y su tamaño N. Suponiendo que la función de dispersión es uniforme, la longitud media de una lista es λ. Ese es el coste de las operaciones de búsqueda, inserción o borrado (*casi* constante).

Implementación de diccionarios mediante tablas

Las tablas dispersas abiertas son una buena implementación de los diccionarios cuando no es necesario recorrer los pares $\langle clave, valor \rangle$ en orden.

Cada lista guarda en sus nodos pares $\langle clave, valor \rangle$ con claves sinónimas. Las operaciones de búsqueda, inserción o borrado primero convierten la clave en cuestión en una posición del vector (a través de la función de dispersión), y después realizan la operación sobre la lista en esa posición.

Si el factor de carga se mantiene bajo y la función de distribución es uniforme, el coste de todas las operaciones es constante.

`hashmap_eda.h`

Problemas

- 28 - Me pilló el toro
- 29 - Capítulos repetidos
- 30 - Actualización de un diccionario
- 31 - Referencias cruzadas
- 32 - Eligiendo deporte
- 33 - Ranking de un concurso de programación

- ACR 109 - Liga de pádel
- ACR 214 - Abdicación de un Rey
- ACR 338 - Detectando copiones