UNIVERSIDAD COMPLUTENSE MADRID

Tema 7: Diccionarios (El TAD map)

Miguel Gómez-Zamalloa
Estructuras de Datos y Algoritmos (EDA)
Grado en Desarrollo de Videojuegos
Facultad de Informática

Temario

- Análisis de eficiencia
  1. Análisis de eficiencia de los algoritmos ✔
- Algoritmos
  2. Divide y vencerás (DV), o *Divide-and-Conquer* ✔
  3. Vuelta atrás (VA), o *backtracking*
- Estructuras de datos
  4. Especificación e implementación de TADs ✔
  5. Tipos de datos lineales ✔
  6. Tipos de datos arborescentes ✔
  7. **Diccionarios (El TAD map)**
  8. Aplicación de TADs

Contenidos

Introducción

Introducción

Problema de ejemplo

Nos piden procesar un texto para producir un listado ordenado con todas sus palabras y para cada una su número de apariciones.

- Para ello querríamos disponer de un TAD (al que llamaremos `map`) que nos permita almacenar palabras sin repeticiones (claves) junto con su número de apariciones (valores).
- Se debe poder insertar nuevos pares clave/valor (`insert` o `op[]`), consultar si una clave está (`count`), borrar pares (`erase`), actualizar el valor de una clave (`op[]`) y recorrer ordenado (iteradores).
- Y lo queremos genérico, tanto en el tipo de las claves como en el de los valores.

Introducción

```
void calcularApariciones(){
  ifstream in(``texto.txt'');
  string palabra;
  map<string,int> words; // Así instanciaríamos el TAD map

  cin >> palabra:
  while (!cin.fail()){
    if (!words.count(palabra)){
      words.insert({palabra,1});
      // Alternativa con op[]: words[palabra] = 1;
    } else {
      words[palabra] = words[palabra] + 1;
      // Versión más sencilla y eficiente: words[palabra]++;
    }
    cin >> palabra;
  }
  in.close();

  // Para recorrer (imprimir) usamos un iterador de pares
  for (auto it = words.begin(); it != words.end(); ++it){
      cout << it->first << " " << it->second << endl;
  }
```

Introducción

- El recorrido, al usar iteradores también lo podríamos escribir así:

```
for (auto par : words)
  cout << par.first << " " << par.second << endl;
```

- O mejor aún, a partir de C++17, también así:

```
for (auto [word,reps] : words)
  cout << word << " " << reps << endl;
```

Introducción

La interfaz del TAD `map` sería esta:

```
template <class Key, class Val, class Comp = std::less<Key> >
class map {
public:
  map(); // Construye map vacío

  int count(Key const& c) const; // Consulta si existe c

  bool insert(pair<Key,Val> const& cv); // Inserta nuevo par

  // Consulta, inserta o permite actualizar valor asociado a c
  Val& operator[](Key const& c);

  bool erase(Key const& c); // Borra par de clave c

  int size() const; // Devuelve número de pares almacenados

  iterator begin(); // Iterador al primero

  iterator end(); // Iterador tras último
};
```

Introducción

- Podríamos implementar el TAD `map` mediante vectores o listas de pares, en ambos casos ordenados. Tendríamos esto:

Operación	Vector ordenado	Lista ordenada
`count`	$\mathcal{O}(log(n))$	$\mathcal{O}(n)$
`insert`	$\mathcal{O}(n)$	$\mathcal{O}(n)$
`operator[]`	$\mathcal{O}(log(n))$*	$\mathcal{O}(n)$
`erase`	$\mathcal{O}(n)$	$\mathcal{O}(n)$
`size`	$\mathcal{O}(1)$	$\mathcal{O}(1)$

- En las listas, aunque no hace falta abrir/cerrar huecos para las inserciones/eliminaciones, **no se puede hacer búsqueda binaria!** De ahí las complejidades lineales en todas las operaciones.
- (*) El `operator[]` podría tener que insertar cuando no se encuentra la clave buscada, en cuyo caso sería $\mathcal{O}(n)$.

El TAD map mediante BSTs

El TAD map mediante BSTs

- Para mejorarlo, podemos usar árboles binarios de búsqueda (BSTs) extendidos para que almacenen pares clave/valor.
- En ese caso (asumiendo que les dotamos de lógica de reequilibrado) tendríamos complejidad $\mathcal{O}(log(n))$ para las cuatro operaciones!
- Toda la gestión relacionada con la operación de orden se haría únicamente con las claves.
- El recorrido *inorden* quedará ordenado por clave.
- La implementación será muy parecida a la del TAD `set`.

El TAD map mediante BSTs

```
template <class Key, class Val, class Comp = std::less<Key> >
class map {
protected:
  using clave_valor = std::pair<const Key, Val>;

  /* Clase nodo que almacena internamente la pareja <clave, valor>
  y punteros al hijo izquierdo y al hijo derecho. */
  struct TreeNode {
    clave_valor cv;
    Link iz, dr;
    TreeNode(clave_valor const& e, Link i = nullptr,
             Link d = nullptr) : cv(e), iz(i), dr(d) {}
  };
  using Link = TreeNode*;

  Link raiz; // puntero a la raíz de la estructura

  int nelems; // número de parejas <clave, valor>

  Comp menor; // objeto función que compara claves
```

El TAD map mediante BSTs

```
public:
  // constructor (map vacío)
  map(Comp c = Comp()) : raiz(nullptr), nelems(0), menor(c) {}
```

- Construcción, destrucción, `empty`, `size` y `operator==` serían exactamente iguales que en TAD `set`.
- `count`, `insert` y `erase` son también esencialmente iguales (excepto que hay que acceder a la clave del nodo cuando corresponda).
- La gestión del iterador también será análoga.
- Añadiremos también la operación

  ```
  Val const& at(Key const& c) const;
  ```

  que devuelve el valor asociado a una clave o lanza excepción si no se encuentra, con implementación análoga a `count`.

El TAD map mediante BSTs - operator[]

```
public:
  Val& operator[](Key const& c) {
    return corchete(c, raiz);
  }

protected:
  Val& corchete(Key const& c, Link& a) {
    if (a == nullptr) {
      // se inserta la nueva clave, con un valor por defecto
      a = new TreeNode(clave_valor(c,Val()));
      ++nelems;
      return a->cv.second;
    }
    else if (menor(c, a->cv.first)) return corchete(c, a->iz);
    else if (menor(a->cv.first, c)) return corchete(c, a->dr);
    else // la clave ya está, se devuelve el valor asociado
      return a->cv.second;
  }
```

El TAD map mediante BSTs - Operación `find`

Para evitar búsquedas redundantes, se proporciona un método adicional de búsqueda por clave que devolverá un iterador al par buscado (que sería el iterador `end()` si la clave no se encuentra):

```
iterator find(Key const& c) { return iterator(this, c); }

class iterator {// Nuevo constructor en class iterator
...
  iterator(map<Key,Val,Comp> const* m, Key const& c) {
    act = m->raiz;
    bool encontrado = false;
    while (act != nullptr && !encontrado) {
      if (m->menor(c, act->cv.first)) {
        ancestros.push(act);
        act = act->iz;
      } else if (m->menor(act->cv.first, c)) act = act->dr;
      else encontrado = true;

    if (!encontrado) // act es nullptr -> iterador end()
      ancestros = std::stack<Link>(); // Vacía la pila
}
```

El TAD map mediante BSTs - Operación `find`

- Usando el nuevo método `find` el `if` de nuestro ejemplo de la introducción quedaría así:

```
auto itPalabra = words.find(palabra);
if (itPalabra == words.end()) // Si es la primera aparición
  words[palabra] = 1;
else
  itPalabra->second++; // El second sería el valor
```

- Evitamos una búsqueda redundante cuando la palabra está.
- Se puede evitar la otra búsqueda redundante si el `insert` devolviese iterador al elemento insertado (ver `map` de STL).

```
pair<iterator,bool> res = words.insert({palabra,1});
if (!res.second) res.first->second++;
```

El TAD map mediante BSTs - Complejidades

- Asumiendo que disponemos de soporte para asegurar que los árboles se mantienen equilibrados tendríamos las siguientes complejidades:

Operación	Implementación con BST
`count`	$\mathcal{O}(log(n))$
`insert`	$\mathcal{O}(log(n))$
`operator[]`	$\mathcal{O}(log(n))$
`at`	$\mathcal{O}(log(n))$
`find`	$\mathcal{O}(log(n))$
`erase`	$\mathcal{O}(log(n))$
`size`	$\mathcal{O}(1)$
`begin`	$\mathcal{O}(log(n))$*

- (*) Se podría conseguir fácilmente que sea $\mathcal{O}(1)$ dejando guardado un puntero al nodo de más a la izquierda (ver `map` de la STL).

Tablas Dispersas (hash maps)

Tablas Dispersas (hash maps)

Idea clave:

Almacenar en un vector los *valores* y usar las *claves* como índices.

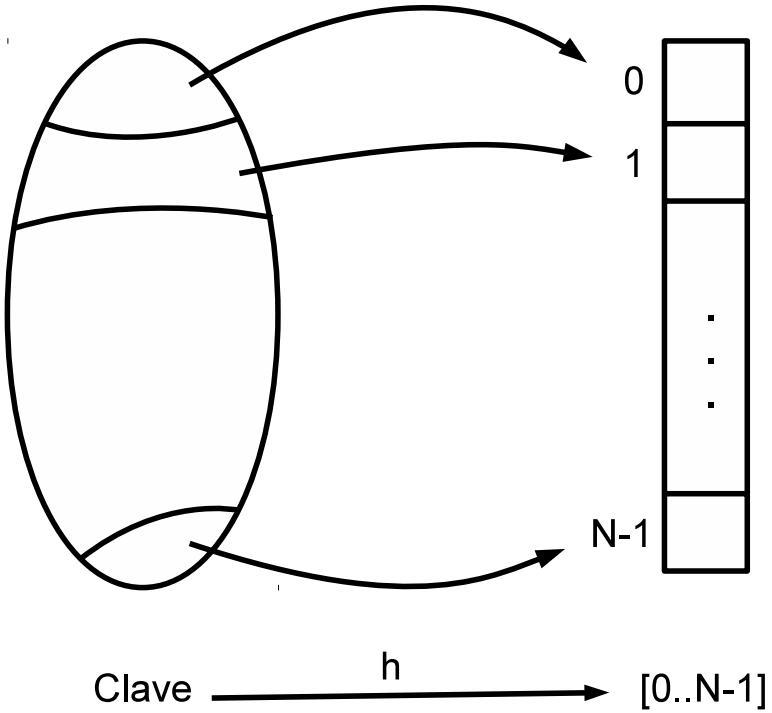
- Esto sería inmediato si por ejemplo las claves fuesen enteros en un rango, como pasaba en el problema de los anagramas del tema 1 (eran char de 'a' a 'z'). En ese caso tendríamos complejidades constantes!
- ¿Qué pasa si por ejemplo las claves son palabras? No sería factible tener un array con tantos elementos como palabras en el alfabeto.
- Necesitamos un mecanismo que permita establecer una correspondencia entre un conjunto de claves potencialmente muy grande y un vector de valores mucho más pequeño.
- Este mecanismo será la función de localización o *hash*.

La función hash

- Sea N el tamaño del vector, nos basaremos en una función

$$h : Clave \rightarrow [0..N-1]$$

de manera que dada una clave c, $h(c)$ represente la posición del vector que debería contener su valor asociado.



La función hash

- Como el número de posibles claves será en general mucho mayor que N podrá haber claves a las que se asocie la misma posición del vector.
- Se dice que se produce una *colisión* cuando para dos claves c y c':

$$c \neq c' \wedge h(c) = h(c')$$

- Ejemplo: Supongamos $N = 16$, un `map<string,int>` para almacenar edades asociadas a personas, y esta función de localización:

$$h(c) = ord(ult(c)) \ mod \ 16$$

donde $ult(c)$ devuelve el último carácter de la cadena, y *ord* el código ASCII de un carácter (en C++ sería `c.back()%16`). Por ejemplo:

$$\begin{aligned} \text{h("Fred")} &= \text{ord('d') mod 16} = 100 \text{ mod } 16 = 4 \\ \text{h("Joe")} &= \text{ord('e') mod 16} = 101 \text{ mod } 16 = 5 \\ \text{h("John")} &= \text{ord('n') mod 16} = 110 \text{ mod } 16 = 14 \end{aligned}$$

La función hash

- Para que la búsqueda funcione de manera óptima, las funciones de localización deben tener las siguientes propiedades:
  - *Eficiencia*: el coste de calcular $h(c)$ debe ser bajo.
  - *Uniformidad*: el reparto de claves entre posiciones del vector debe ser lo más uniforme posible. Idealmente, para una clave c elegida al azar la probabilidad de que $h(c) = i$ debe valer $1/N$ para cada $i \in [0..N-1]$.
- La función de localización anterior es eficiente pero no uniforme $\Rightarrow$ La probabilidad de que un nombre acabe en 'a' es mucho mayor de que lo haga en 'z'.
- Se podría mejorar fácilmente por ejemplo sumando los códigos ASCII de (algunos de) los caracteres de la cadena.

Tablas abiertas vs. cerradas

- La probabilidad de tener colisiones, incluso con una función uniforme, es muy alta. Por ejemplo:

$$h(\text{"Fred"}) = h(\text{"David"}) = h(\text{"Violet"}) = h(\text{Roland"}) = 4$$

- Hay dos estrategias distintas para manejas colisiones:
  - *Tablas abiertas*: cada posición del vector almacena una lista de pares clave-valor con todos los pares que colisionan en dicha posición.
  - *Tablas cerradas*: si al insertar un par clave-valor se produce colisión, se busca otra posición del vector vacía donde almacenarlo (técnicas de *relocalización*).
- Ambas estrategias se utilizan en la práctica. En EDA solo veremos la estrategia de tablas abiertas.

Tablas abiertas

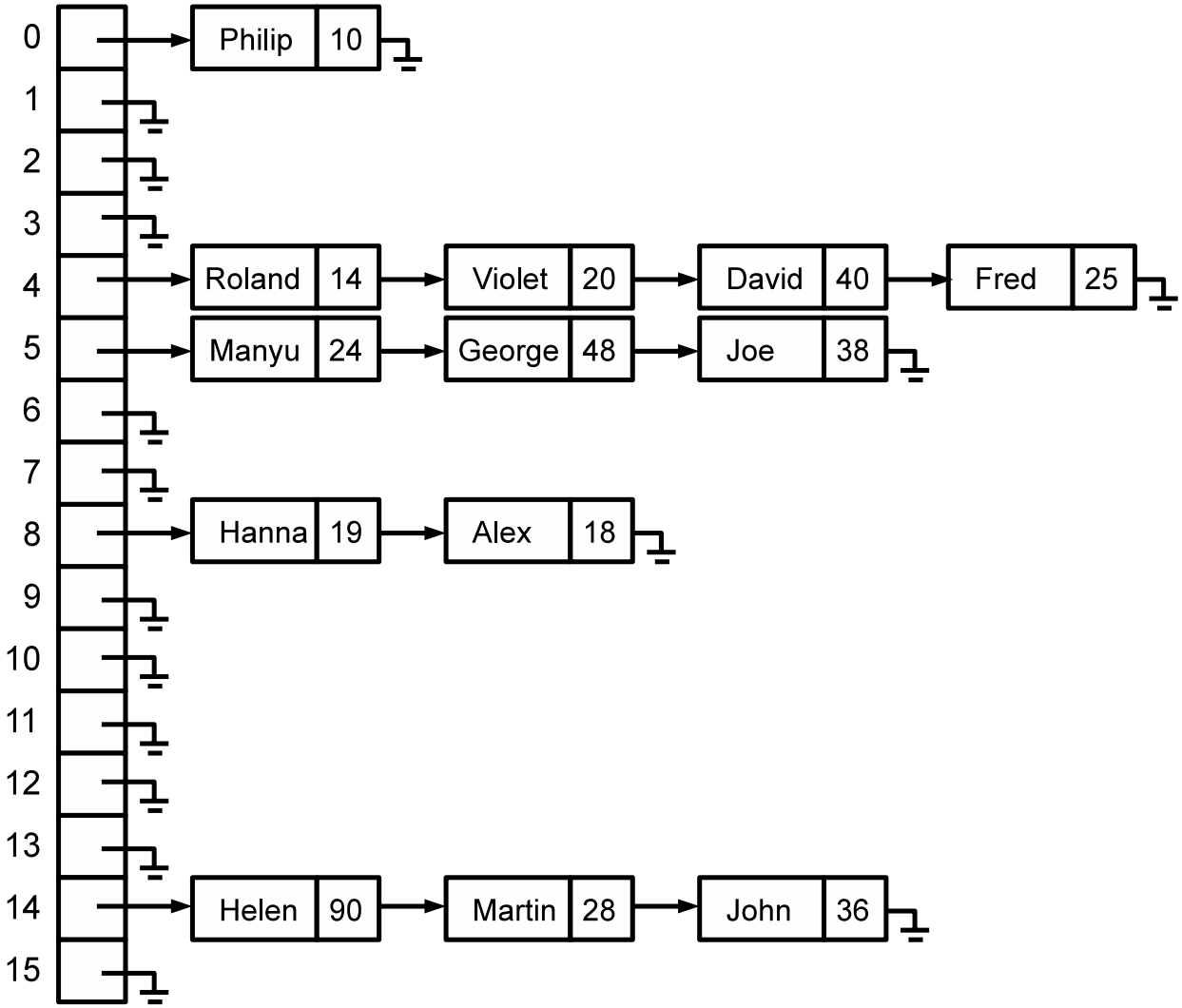
Ejemplo: Tras la siguiente secuencia de inserciones sobre una tabla abierta con $N = 16$:

```
unordered_map<std::string, int> edades; // (*)
edades.insert({"Fred", 25});        edades.insert({"Alex", 18});
edades.insert({"Philip", 10});      edades.insert({"Joe", 38});
edades.insert({"John", 36});        edades.insert({"Hanna", 19});
edades.insert({"David", 40});       edades.insert({"Martin", 28});
edades.insert({"Violet", 20});      edades.insert({"George", 48});
edades.insert({"Helen", 90});       edades.insert({"Manyu", 24});
edades.insert({"Roland", 14});
```

(*) Llamaremos a la clase `unordered_map` para coincidir con la clase análoga de la STL de C++. En Java por ejemplo se denomina `HashMap` mientras que en C# es `Dictionary`.

Tablas abiertas

el resultado en memoria sería este:



Tablas abiertas - Tasa de ocupación

- Una característica muy importante en una tabla hash es la *tasa de ocupación*.

Tasa de ocupación α

Se define como la relación entre el número de pares almacenados (nelems) y el número de posiciones del vector (N).

$$\alpha = nelems/N$$

- En el ejemplo anterior, tenemos $nelems = 13$ y $N = 16$, por tanto:

$$\alpha = 13/16 = 0,8125$$

- Cuando la tasa de ocupación se acerca a 1 la probabilidad de coliciones sería muy alta.
- Por tanto las tablas se redimensionarán para que la tasa de ocupación siempre quede por debajo de un umbral, por ejemplo 0,75.

Implementación - Estructura de datos

```
template <class Key, class Val, class Hash = std::hash<Key>,
          class Pred = std::equal_to<Key>>
class unordered_map {
public:
  using clave_valor = std::pair<const Key, Val>;

protected:
  struct ListNode {
    clave_valor cv;
    Link sig;
    ListNode(clave_valor const& e, Link s=nullptr): cv(e), sig(s){}
  };
  using Link = ListNode*; // Alias por comodidad

  static const int TAM_INICIAL = 17; // tamaño inicial de la tabla
  static const int MAX_CARGA = 75; // max ocupación permitida 75%

  vector<Link> array; // vector de listas (de pares) de colisión

  int nelems; // número de pares almacenados

  Hash hash; // objeto función para el hash de las claves
  Pred pred; // objeto función para comparar claves
```

Implementación - Construcción y destrucción

```
unordered_map(int n=TAM_INICIAL, Hash h = Hash(), Pred p=Pred()):
              array(n, nullptr), nelems(0), hash(h), pred(p) {}

~unordered_map() { libera(); }

protected:
void libera() { // Libera memoria del vector de listas
  for (int i = 0; i < array.size(); ++i) {
    // liberamos los nodos de la lista array[i]
    Link act = array[i];
    while (act != nullptr) {
      Link aux = act;
      act = act->sig;
      delete aux;
    }
    array[i] = nullptr;
  }
}
```

- La memoria del vector se libera automáticamente al invocarse al destructor de `vector`.

Implementación - Localización

- Implementamos un método que busca un nodo con una clave dada.
- A parte de devolver el puntero al nodo buscado (`nullptr` si no se encuentra), devuelve un puntero al nodo anterior (necesario para la eliminación).

```
 /** Busca un nodo a partir de "pos" (el primero de la lista
  * correspondiente) que contenga c. Si lo encuentra, "pos" quedará
  * apuntando a dicho nodo y "ant" al nodo anterior. Si no lo
  * encuentra "pos" quedará apuntando a nullptr. */
bool localizar(Key const& c, Link& ant, Link& pos) const {
  ant = nullptr;
  while (pos != nullptr) {
    if (pred(c, pos->cv.first)) return true;
    else {
      ant = pos; pos = pos->sig;
    }
  }
  return false;
}
```

Implementación - Inserción

```
public:
bool insert(clave_valor const& cv) {
  int i = hash(cv.first) % array.size();
  Link ant, pos = array[i];
  if (localizar(cv.first, ant, pos)) { // la clave ya existe
    return false;
  } else {
    if (muy_llena()) { // Se supera máx tasa ocupación permitida
      amplia(); // Se redimensiona -> lo vemos después
      i = hash(cv.first) % array.size();
    }
    array[i] = new ListNode(cv, array[i]);
    ++nelems;
    return true;
  }
}

protected:
bool muy_llena() const {
  return 100.0 * nelems / array.size() > MAX_CARGA;
}
```

Implementación - Eliminación

```
public:
bool erase(Key const& c) {
  int i = hash(c) % array.size();
  Link ant, pos = array[i];
  if (localizar(c, ant, pos)) {
    if (ant == nullptr) // El nodo es el primero de la lista
      array[i] = pos->sig;
    else
      ant->sig = pos->sig;
    delete pos;
    --nelems;
    return true;
  } else
  return false;
}
```

Implementación - Operaciones at y []

```
Val const& at(Key const& c) const {
  int i = hash(c) %array.size();
  Link ant, pos = array[i];
  if (localizar(c, ant, pos)) return pos->cv.second;
  else throw std::out_of_range("La clave no se encuentra");
}

Val& operator[](Key const& c) {
  int i = hash(c) %array.size();
  Link ant, pos = array[i];
  if (localizar(c, ant, pos)) {
    return pos->cv.second;
  } else { // Se inserta por la izquierda
    array[i] = new ListNode(clave_valor(c,Val()), array[i]);
    ++nelems;
    return array[i]->cv.second;
  }
}
```

Implementación - count, empty y size

```
int count(Key const& c) const  {
  int i = hash(c) % array.size();
  Link ant, pos = array[i];
  return localizar(c, ant, pos) ? 1 : 0;
}

bool empty() const {
  return nelems == 0;
}

int size() const {
  return nelems;
}
```

Implementación - Redimensión

```
void amplia() {
  vector<Link> nuevo(siguiente_primo(array.size()*2), nullptr);
  for (int j = 0; j < array.size(); ++j) {
    Link act = array[j];
    while (act != nullptr) {
      Link a_mover = act;
      act = act->sig;
      int i = hash(a_mover->cv.first) % nuevo.size();
      a_mover->sig = nuevo[i];
      nuevo[i] = a_mover;
    }
  }
  swap(array, nuevo);
}
```

- Está demostrado que el uso de números primos para el tamaño del vector mejora el rendimiento.
- Observa que al cambiar el tamaño hay que rehashear. Así se evitan las colisiones en la nueva tabla!
- Observa tb que reutilizamos los nodos, simplemente los movemos.

Implementación - Iterador

- Implementaremos el iterador para que recorra la tabla de arriba a abajo y de izquierda a derecha por cada lista de colisión.
- Puesto que no hay ningún orden aparente, se podría haber seguido cualquier otra estrategia.
- El iterador necesita guardar el puntero al nodo y el índice del vector donde está.

```
class iterator {
protected:
  friend class unordered_map;
  unordered_map<Key,Val,Hash,Pred>* tabla; // puntero a la tabla
  Link act; // nodo actual
  int ind;  // índice de la lista de colisión actual
```

Implementación - Iterador

```
// Construcción de iterador al primer elemento o al último
iterator(unordered_map<...>* t, bool first = true) : tabla(t) {
  if (first) { // Se busca el primer nodo de la tabla
    ind = 0;
    while (ind < tabla->array.size() &&
           tabla->array[ind] == nullptr) {
      ++ind;
    } // Tenemos O(array.size()) en caso peor
    act = (ind < tabla->array.size() ? tabla->array[ind]
                                     : nullptr);
  } else { // Se construye el iterator end() con act=nullptr
    act = nullptr;
    ind = tabla->array.size();
  }
}

void next() { // Busca el siguiente nodo a act
  if (act == nullptr) throw std::out_of_range(...);
  act = act->sig;
  while (act == nullptr && ++ind < tabla->array.size()) {
    act = tabla->array[ind];
  } // Tenemos O(array.size()) en caso peor
}
```

Implementación - Iterador

```
  public: // class iterator, interna de unordered_map
    clave_valor& operator*() const {
      if (act == nullptr) throw std::out_of_range(...);
      return act->cv;
    }

    iterator& operator++() { // ++ prefijo
      next();
      return *this;
    }

    bool operator==(iterator const& that) const {
      return act == that.act;
    }
  };

public: // class unordered_map
  iterator begin() { return iterator(this); }

  iterator end() { return iterator(this, false); }
}; // class unordered_map
```

Implementación - Iterador

Por ejemplo, un recorrido para imprimir todos los elementos contenidos en nuestra tabla de edades podría ser así:

```
unordered_map<string, int> edades;

... // Insertar elementos en la tabla

for (auto it = edades.begin(); it != edades.end(); ++it)
  cout << "(" << it->first << ",  " << it->second << ")\n";
```

Usando un bucle *range-based-for* podría escribirse de esta otra forma (solo válido de esta forma a partir de la versión C++17):

```
  for (auto [nombre, edad] : t)
     cout << "(" << nombre << ",  " << edad << ")\n";
```

Un análisis basto podría determinar que la complejidad del recorrido sería $\mathcal{O}(n * n)$. Sin embargo, al pensarlo bien vemos que en el caso peor se pasa por todos los nodos y posiciones del vector una vez $\Rightarrow \mathcal{O}(n)$.

Implementación - Operación `find`

Finalmente implementamos la operación `find` para buscar una clave devolviendo el correspondiente iterador:

```
  class iterator {
    ...
    // iterador a una clave
    Iterador(unordered_map<...>* t, Key const& c) : tabla(t) {
      ind = tabla->hash(c) % tabla->array.size();
      Link ant;
      act = tabla->array[ind];
      if (!tabla->localizar(c, ant, act)) { // iterador al final
        act = nullptr; ind = tabla->array.size();
      }
    }
    ...
  }; //fin de la clase iterator

// En la clase unordered_map
iterator find(Clave const& c) {
  return iterator(this, c);
}
```

Complejidad de las operaciones

- En este caso hablamos de complejidades en promedio:

Operación	Implementación con tabla hash
`count`	$\mathcal{O}(1)$
`insert`	$\mathcal{O}(1)$*
`operator[]`	$\mathcal{O}(1)$
`at`	$\mathcal{O}(1)$
`find`	$\mathcal{O}(1)$
`erase`	$\mathcal{O}(1)$
`size`	$\mathcal{O}(1)$
`begin`	$\mathcal{O}(1)$**

- Hay que entender que aunque sean $\mathcal{O}(1)$ en promedio no son operaciones gratuitas (excepto `size`). Incluso podrían llegar a tener casos peores $\mathcal{O}(n)$ con funciones hash que funcionen muy mal.
- (**) En implementación vista sería $\mathcal{O}(n)$ pero sería facil hacerlo $\mathcal{O}(1)$.
- (*) Aquí tenemos el coste de la posible redimensión que como otras veces decimos que queda amortizado.

Conclusiones

- Ventajas de `unordered_map` (tablas hash): Menor complejidad promedio y por lo tanto más probabilidad de tener mejor rendimiento.
- Ventajas de `map` (BSTs): Complejidad logarítmica garantizada, recorrido ordenado y posibilidad de acceder a mínimo (y máximo) en $\mathcal{O}(1)$.
- Ambas implementaciones se usan en la práctica.

Regla general:

Elegiremos `map` cuando se vaya a aprovechar el recorrido ordenado o se necesite consultar o borrar el mínimo o el máximo. En caso contrario, elegiremos `unordered_map`.