TD3: Algoritmos y Estructuras de Datos

Prof. Agustín Garassino, Gervasio Pérez

Segundo Semestre de 2024

Clase Teórica 12
Tabla de Hash

Resumen

En la clase de hoy veremos

- Direccionamiento directo
- Funciones de hash
- Encadenamiento
- Direccionamiento Abierto
- Manejo de colisiones: Encadenamiento y Direccionamiento Abierto
- Uso en C++

Motivación

Quisiéramos tener un Conjunto/Diccionario que nos provea

- Inserción
- Búsqueda
- Eliminación

...en O(1) *(o casi)*. £Es esto posible?

Motivación

Quisiéramos tener un Conjunto/Diccionario que nos provea

- Inserción
- Búsqueda
- Eliminación

...en O(1) *(o casi)*. £Es esto posible?

Idea: Almacenar elementos en una tabla (vector) para tener acceso a cada elemento en O(1).

Problema: £cómo ubicar la posición asociada a cada elemento?

Direccionamiento directo

Escenario más sencillo: mis claves son enteros que pertenecen a un conjunto $0, 1, \ldots, m-1$ donde m es un valor no demasiado grande.

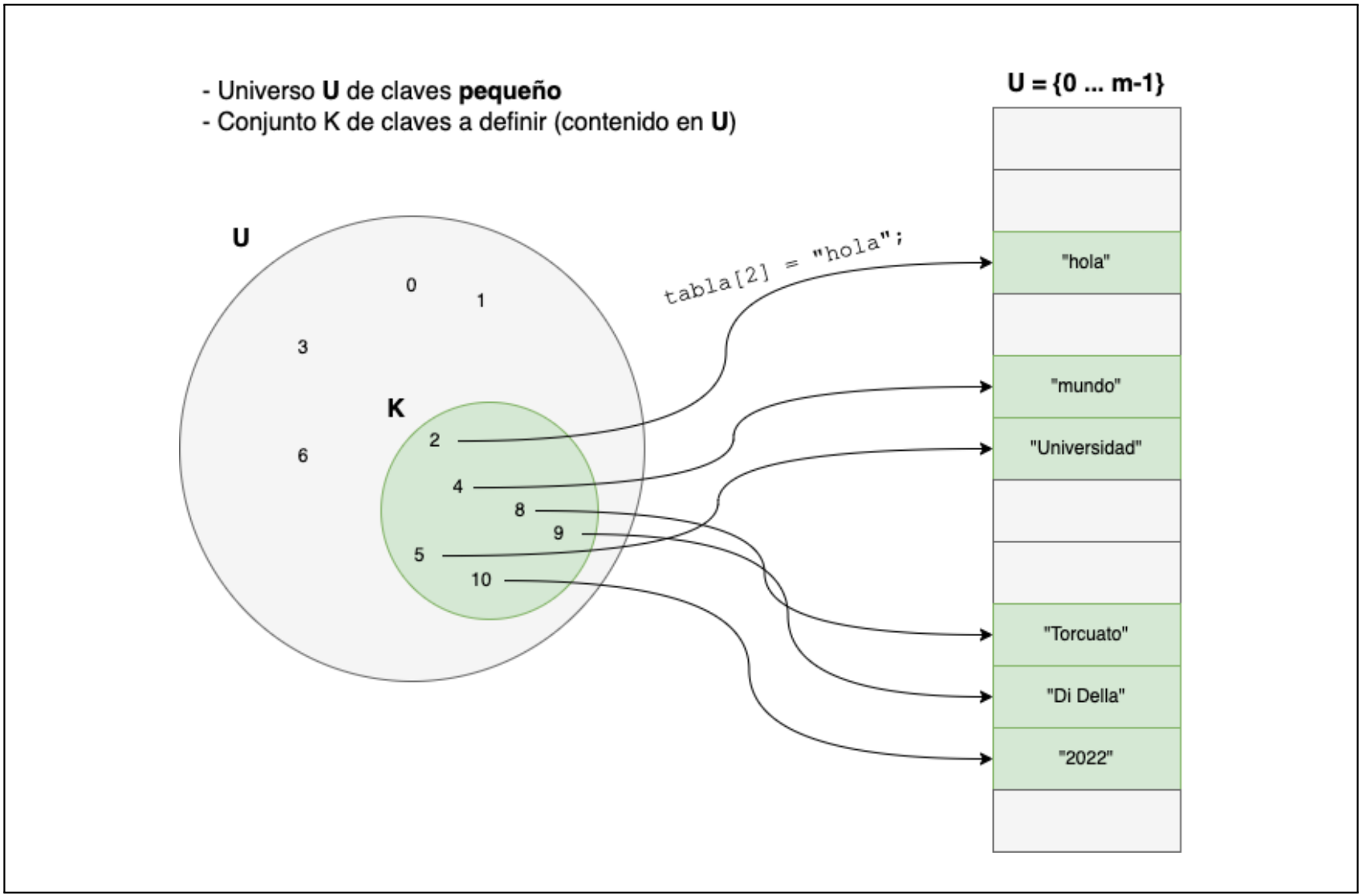
Direccionamiento directo

Escenario más sencillo: mis claves son enteros que pertenecen a un conjunto $0, 1, \ldots, m-1$ donde m es un valor no demasiado grande.

Solución: creo un `vector` de m posiciones. Mis operaciones de buscar/definir/borrar se reducen a acceder a la posición i para manipular la clave i.

Direccionamiento directo

- Ejemplo: diccionario con claves de tipo **`int`** y valores de tipo `string`
- Estructura de representación: `vector<string> tabla`



Direccionamiento directo: Problemas

- En principio sólo funciona para claves enteras.
- Si el universo de posibles valores **U** es demasiado grande, almacenar una tabla de tamaño $|\mathbf{U}|$ puede ser impracticable.
- El conjunto **K** de valores que realmente se van a almacenar puede ser mucho más chico que **U**, y la mayor parte del espacio de la tabla estaría desperdiciado.

Direccionamiento directo: Problemas

- En principio sólo funciona para claves enteras.
- Si el universo de posibles valores **U** es demasiado grande, almacenar una tabla de tamaño $|\mathbf{U}|$ puede ser impracticable.
- El conjunto **K** de valores que realmente se van a almacenar puede ser mucho más chico que **U**, y la mayor parte del espacio de la tabla estaría desperdiciado.

Idea

- Una tabla de tamaño $O(|\mathbf{K}|)$
- Una función $h : \mathbf{U} \to 0 \ldots |\texttt{tabla}| - 1$.
  - `tabla[h(c)] = v;` almacena en la clave c el valor v.
  - Llamamos a h la función de hash

Función de hash: propiedades

- Es deseable que costo de evaluar h sea $O(1)$, y lo asumiremos durante la clase.

Función de hash: propiedades

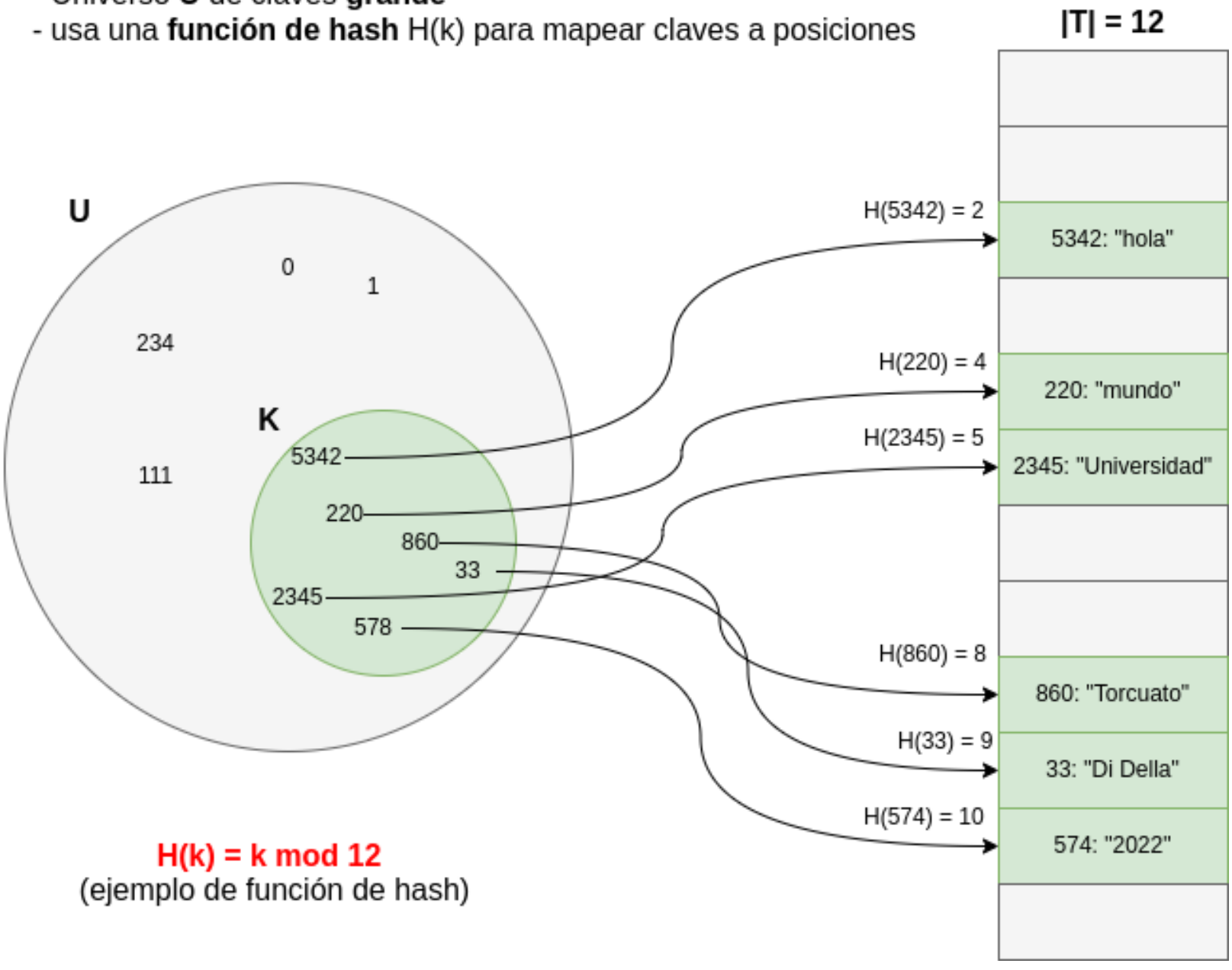
- Es deseable que costo de evaluar h sea $O(1)$, y lo asumiremos durante la clase.
- Hay más valores distintos en $\mathbf{U}$ que hashes resultantes posibles, por lo que valores $c_1, c_2 \in \mathbf{U}$ tales que $h(c_1) = h(c_2)$.
  - Se dice que hay una colisión entre c_1 y c_2
  - La performance de nuestra tabla se va degradando a medida que aumenta la cantidad.
  - Es deseable que h tenga una distribución uniforme:
    que el conjunto de valores posibles $\mathbf{U}$ se reparta, a través de h uniformemente en el conjunto de valores de salida.

Función de hash: propiedades

- Es deseable que costo de evaluar h sea $O(1)$, y lo asumiremos durante la clase.
- Hay más valores distintos en $\mathbf{U}$ que hashes resultantes posibles, por lo que valores $c_1, c_2 \in \mathbf{U}$ tales que $h(c_1) = h(c_2)$.
  - Se dice que hay una colisión entre c_1 y c_2
  - La performance de nuestra tabla se va degradando a medida que aumenta la cantidad.
  - Es deseable que h tenga una distribución uniforme: que el conjunto de valores posibles $\mathbf{U}$ se reparta, a través de h uniformemente en el conjunto de valores de salida.
- Ejemplos
  - Método de división: $h(k) = k \bmod |t|$, usando $|t|$ primo y no muy cercano a una potencia de 2.
  - Método de la multiplicación: $h(k) = \lfloor |t| \times (parte\_decimal(kA)) \rfloor$, siendo A una constante adecuada.

Tabla de Hash

- Universo **U** de claves **grande**
- usa una **función de hash** H(k) para mapear claves a posiciones



Colisiones

- $H(k_1) == H(k_2)$ ==> *colisión entre k_1 y k_2*
- **Ejemplo**: **definir clave 112 con string "cruel"**

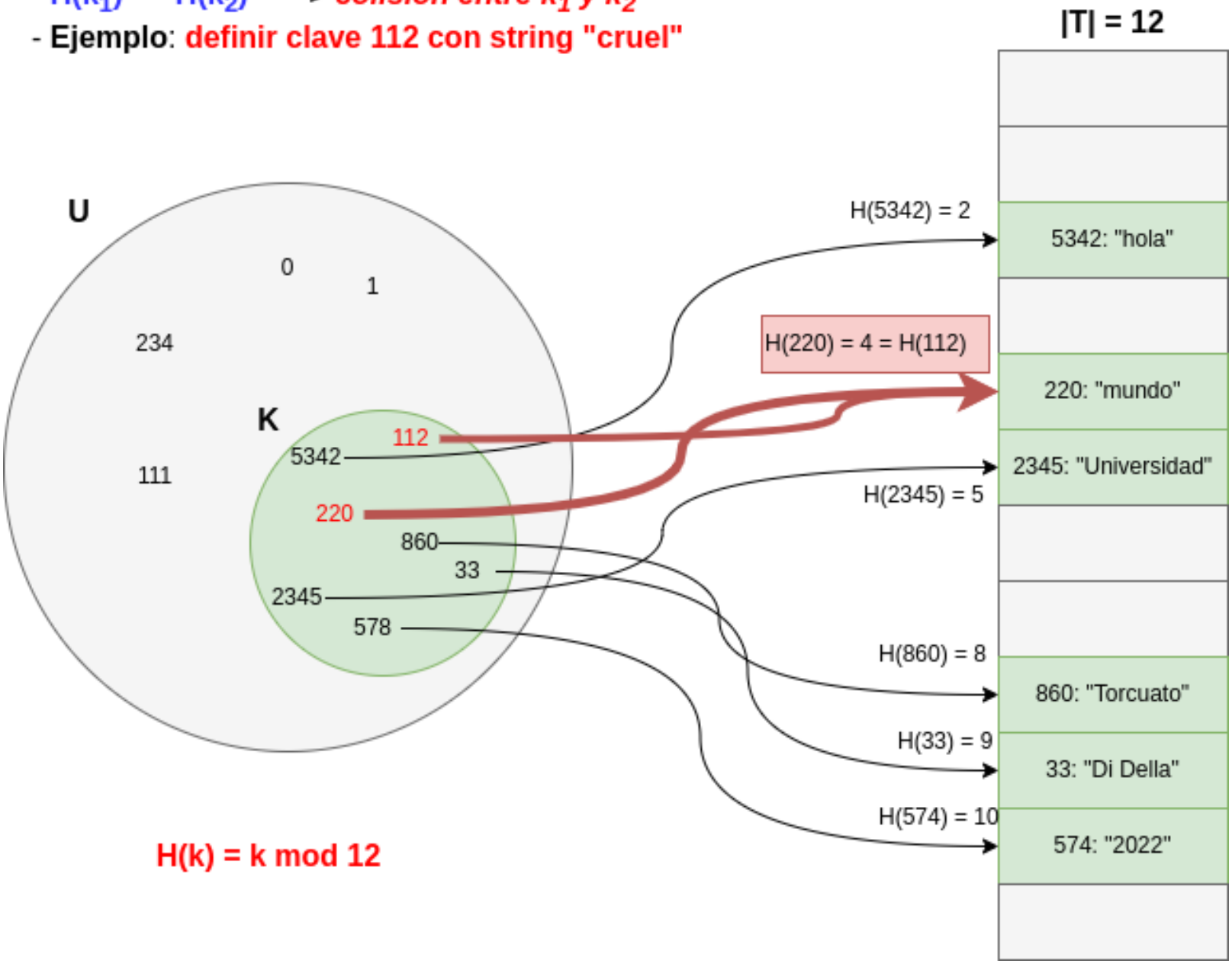


Tabla de hash: colisiones

Dado que hay más claves posibles que posiciones de tabla, pueden ocurrir colisiones: múltiples claves $k_1, k_2, \ldots, k_n$ que se mapeen a una misma posición de la tabla.

Tabla de hash: colisiones

Dado que hay más claves posibles que posiciones de tabla, pueden ocurrir colisiones: múltiples claves $k_1, k_2, \ldots, k_n$ que se mapeen a una misma posición de la tabla.

Ante colisiones se suele aplicar dos técnicas:

- Encadenamiento: En cada posición de la tabla se mantiene una **lista** de las claves guardadas en esa posición.
- Direccionamiento Abierto: Se guarda una única clave en cada posición. Ante una colisión, se busca otra posición para la clave a almacenar (con algún método).

Encadenamiento

Estrategia: ante una colisión, simplemente almacenamos las claves que colisionan en una lista.

Encadenamiento

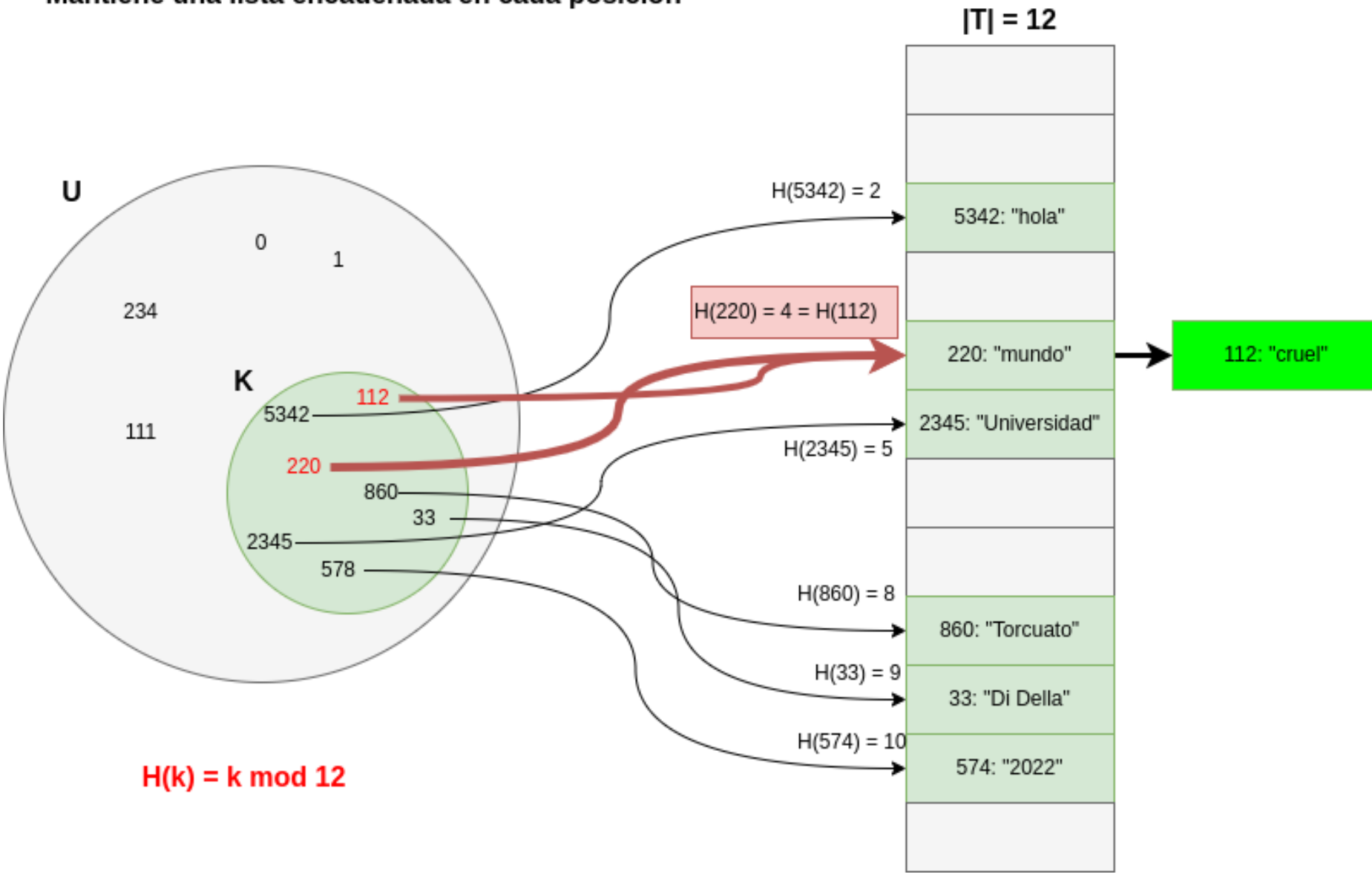
Estrategia: ante una colisión, simplemente almacenamos las claves que colisionan en una lista.

Operaciones

- Agregar una clave c: calcular $h(c)$ y agregar la clave a la lista correspondiente.
  - si podemos asumir que la clave no existe, agregamos al principio/final
  - en caso contrario hay que recorrer toda la lista para manejar evitar potenciales repetidos
- Buscar/borrar una clave c: calcular el hash $h(c)$ y recorrer la lista correspondiente.

Colisiones: Encadenamiento

- Mantiene una lista encadenada en cada posición



Encadenamiento: Complejidad en peor caso

Peor caso: todas las palabras hashean al mismo slot.

Complejidades:

- Agregar una clave asumiendo que no existe: $O(1)$
- Agregar una clave potencialmente existente: $O(n)$
- Buscar una clave: $O(n)$
- Borrar una clave: $O(n)$

Notar que las operaciones más complejas requireren calcular el hash de la clave y luego recorrer la lista correspondiente.

Encadenamiento – complejidad en caso promedio

Para el caso promedio vamos a asumir:

1. función de hashing con distribución uniforme
2. tamaño de la tabla proporcional al tamaño de claves utilizadas

Notar que:

- De [1] obtenemos que el tamaño esperado para la lista asociada a una clave cualquiera es $\alpha = n/|t|$.
- De [2] sabemos que $n \in O(|t|)$, por lo que $\alpha = n/|t| \in O(1)$

Por lo tanto, tenemos:

- Agregar una clave: $O(1)$
- Buscar una clave: $O(1)$
- Borrar una clave: $O(1)$

Direccionamiento Abierto

Estrategia: ante una colisión, buscamos una nueva posición para la clave repetida.

Direccionamiento Abierto

Estrategia: ante una colisión, buscamos una nueva posición para la clave repetida.

Se modifica la implementación de modo que:

1. La función de hash para que tome un parámetro extra que representa el número de intento.
2. Los algoritmos intenten sucesivamente hasta dar con la posición deseada

Recordar que $|t| < |U|$, por lo que esta estrategia introduce la posibilidad de fallo en caso de que la tabla se llene.

Direccionamiento Abierto: Técnicas de búsqueda

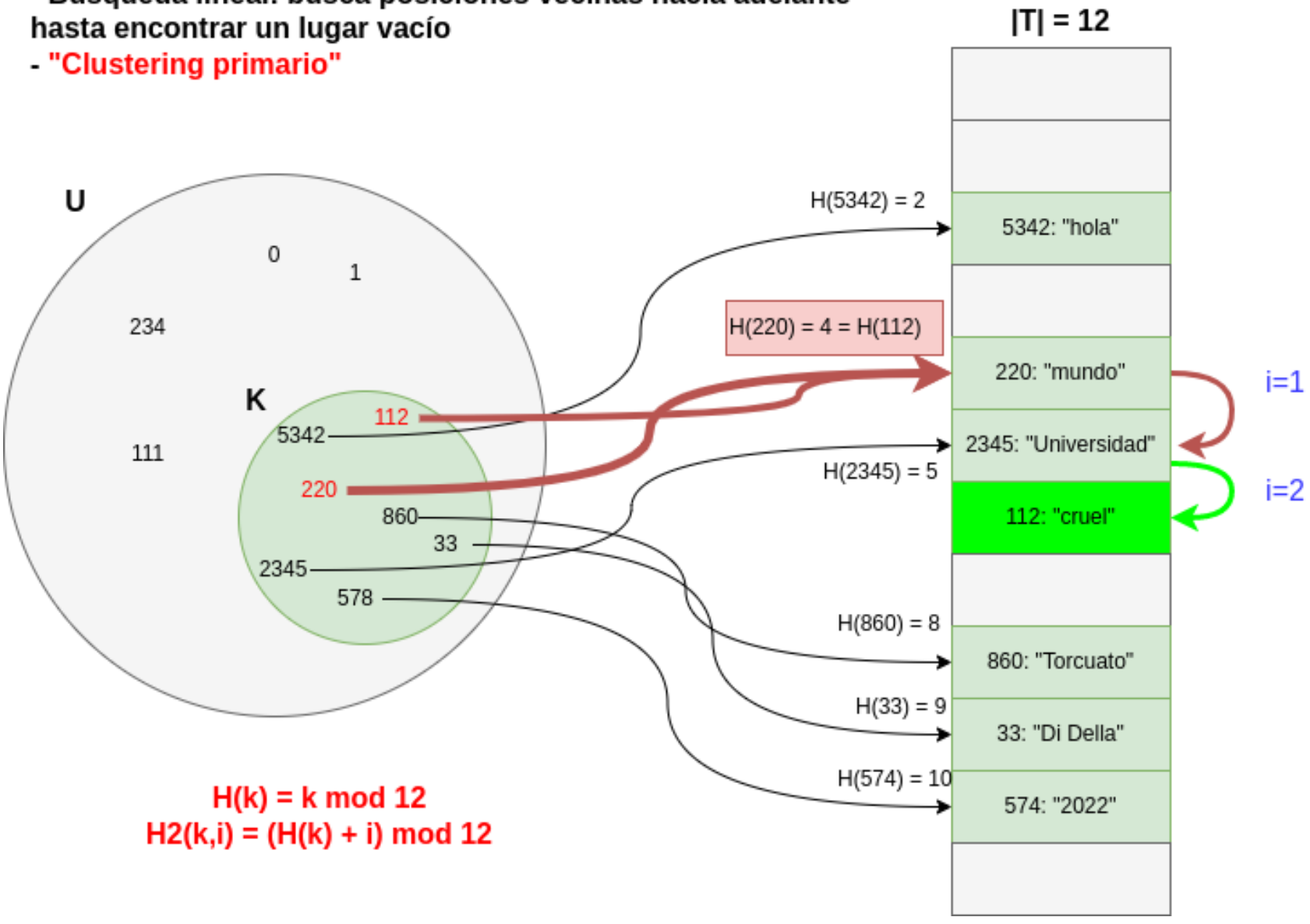
Técnicas de búsqueda

- Búsqueda lineal: Buscamos una posición vacía a partir de la posición inicial.
  $h_2(k,i) = (h(k) + i)\ mod\ |\texttt{tabla}|$
- Búsqueda cuadrática: buscamos nuevas posiciones con saltos cuadráticos desde la posición inicial.
  $h_2(k,i) = (h(k) + c_1 * i + c_2 * i^2)\ mod\ |\texttt{tabla}|$
- Hashing doble: aplicamos otra función de hash para obtener una nueva posición candidata.
  $h_2(k,i) = (h(k) + i * h'(k))\ mod\ |\texttt{tabla}|$

Cada una de estas técnicas tiene sus ventajas y desventajas en lo que respecta a la distribución final de los elementos, performance y uso de cache.

Colisiones: Direccionamiento Abierto

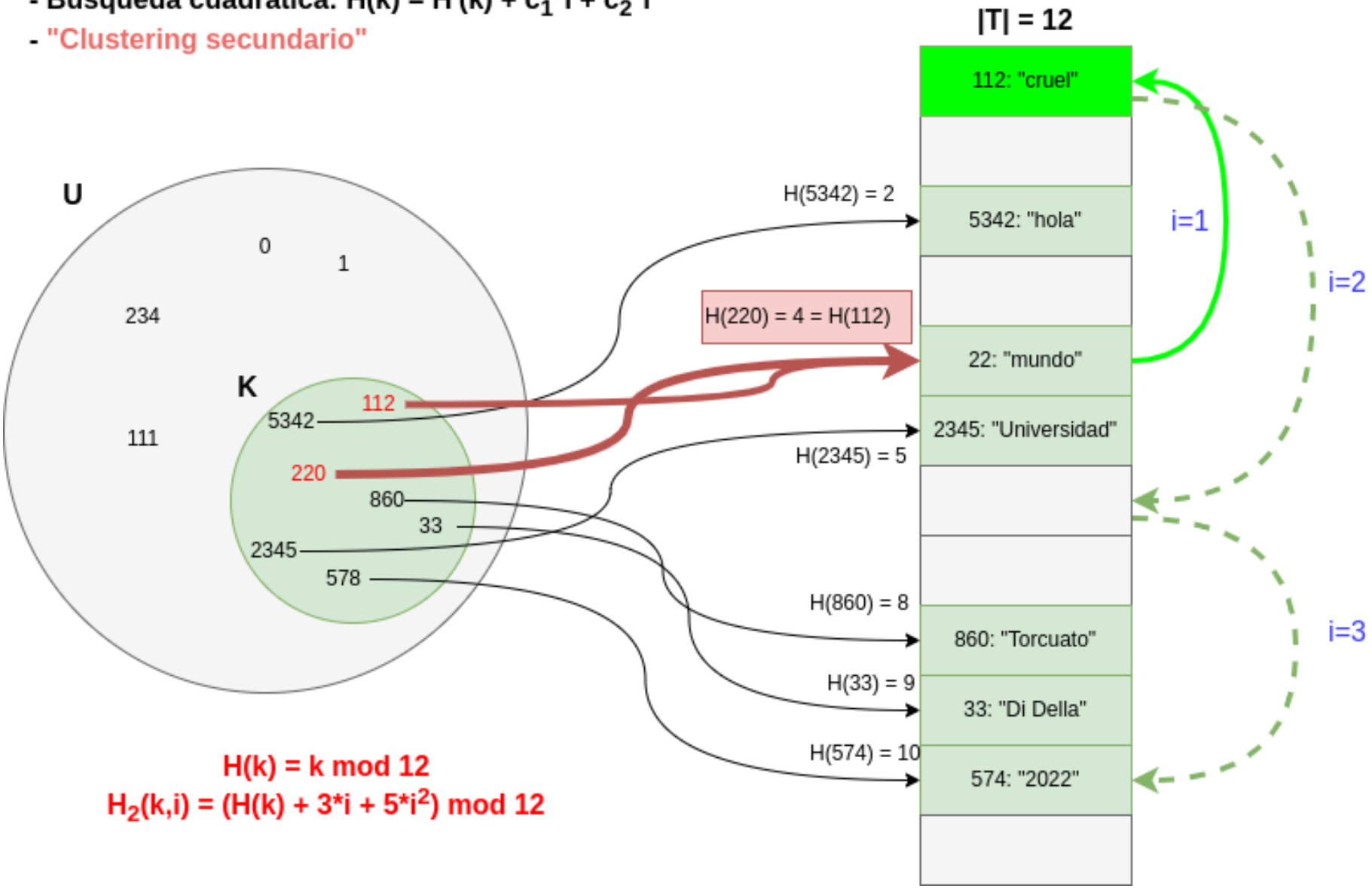
- Búsqueda lineal: busca posiciones vecinas hacia adelante hasta encontrar un lugar vacío
- "Clustering primario"



H(k) = k mod 12
H2(k,i) = (H(k) + i) mod 12

Colisiones: Direccionamiento Abierto

- **Búsqueda cuadrática: $H(k) = H'(k) + c_1*i + c_2*i^2$**
- **"Clustering secundario"**

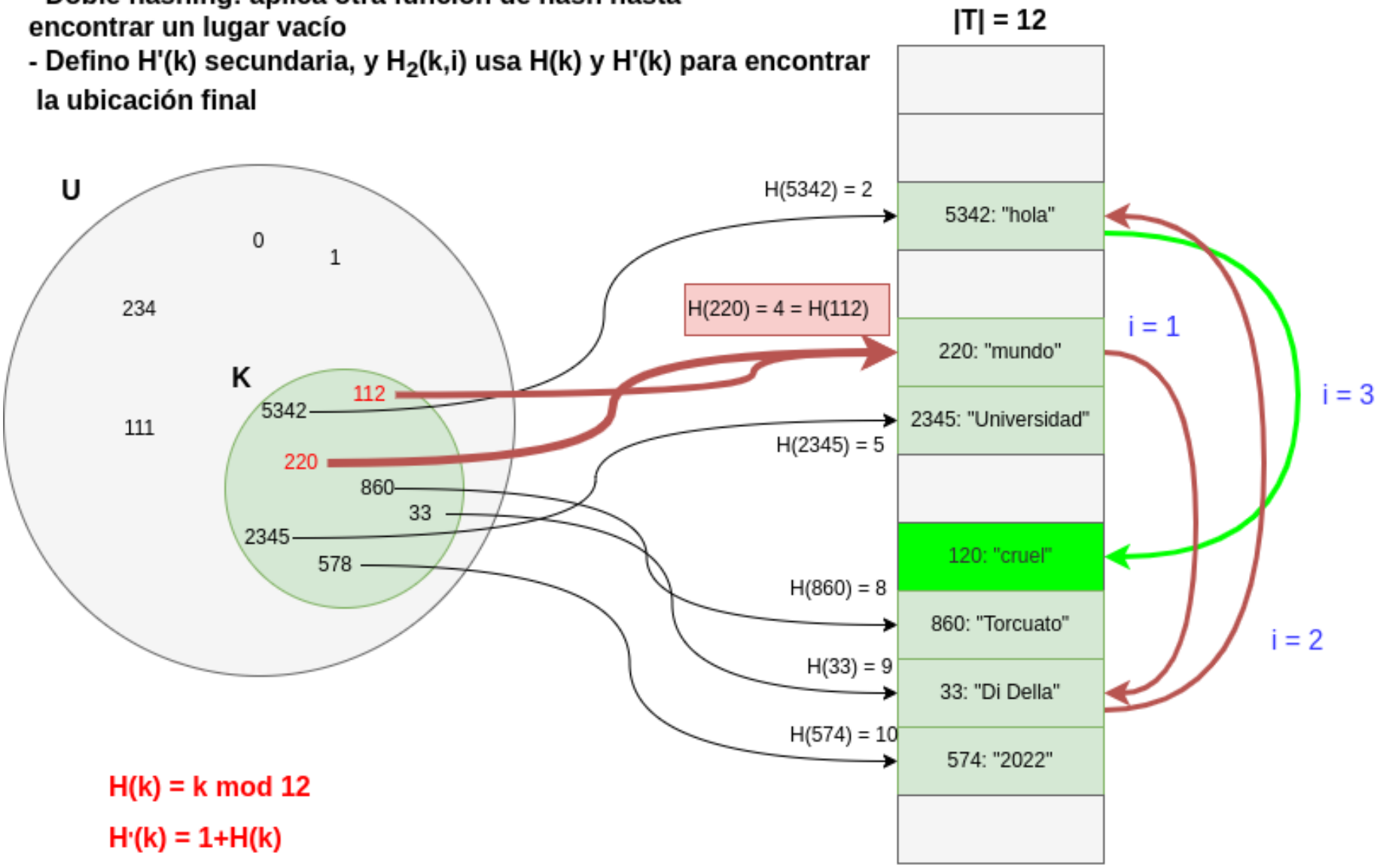


$H(k) = k \bmod 12$

$H_2(k,i) = (H(k) + 3*i + 5*i^2) \bmod 12$

Colisiones: Direccionamiento Abierto

- Doble hashing: aplica otra función de hash hasta encontrar un lugar vacío
- Defino H'(k) secundaria, y $H_2(k,i)$ usa H(k) y H'(k) para encontrar la ubicación final



$H(k) = k \bmod 12$

$H'(k) = 1 + H(k)$

$H_2(k,i) = (H(k) + i * H'(k)) \bmod 12$

Hashing con tipos arbitrarios

Las funciones de hash toman números enteros.

Si queremos usar un tipo de datos T podemos:

1. Interpretar cada instancia de T como un número y luego aplicarle h,
2. Definir una función de hash específica para el tipo T.

```
1  // Ejemplo: hash para string, suma simple de los caracteres
2  long hash(const std::string & s) {
3      long res = 0;
4      for (char c : s)
5        res += c;
6      return res;
7  }
```

Tabla de hash en la std

- `std::unordered_set<T>` y `std::unordered_map<T,V>` están implementados sobre tablas de hash.
- Son equivalentes a `std::set` y `std::map`.
- Búsqueda, inserción y borrado en $O(1)$ para el caso promedio.

Tabla de hash en la std

- `std::unordered_set<T>` y `std::unordered_map<T,V>` están implementados sobre tablas de hash.
- Son equivalentes a `std::set` y `std::map`.
- Búsqueda, inserción y borrado en $O(1)$ para el caso promedio.

Requisitos: que el tipo T tenga definido **`operator==`** (comparación) y una función de hash, provista de la siguiente manera:

```
1  struct mi_hash {
2      size_t operator() (const T& key) const;
3  }
```

C++ std provee una buena **`struct hash<T>`** para muchos tipos básicos T (numéricos, strings, punteros...), pero si no está definida es necesario implementar este template para el T específico.

Tabla de hash en la std

```
1   struct mi_hash_string {
2       size_t operator() (const string& key) const{
3           long res = 0;
4           for (char c : key)
5               res += c;
6           return res;
7       }
8   };
9
10  int main(){
11      // Utilizando la función de hash de C++;
12      unordered_set<string> d1;
13      d1.insert("hola");
14
15      // Utilizando mi función de hash;
16      unordered_set<string, mi_hash_string> d2;
17      d2.insert("hola");
18  }
```

Tabla de hash en la std

```cpp
struct point {
    double x, y;
    bool operator==(const point& other) const {
        return x == other.x && y == other.y;
    }};
// defino el hash de std para point
template<> struct std::hash<point> {
    size_t operator()(const point& p) const {
        return p.x*p.x + p.y*p.y;
    }
};
int main() {
    std::unordered_set<point> s;
    s.insert({1,1});
    s.insert({2,2});
    for (auto p : s) {
        std::cout << p.x << "," << p.y << std::endl;
    }}
```

T13: Resumen de hoy

Tablas de hash (Cormen, Capítulo 11)

- Direccionamiento directo
- Funciones de hash
- Encadenamiento
- Direccionamiento Abierto
- Manejo de colisiones: Encadenamiento y Direccionamiento Abierto
- Uso en C++

Tablas de hash en C++

- `https://en.cppreference.com/w/cpp/container/unordered_set`
- `https://en.cppreference.com/w/cpp/container/unordered_map`
- `https://en.cppreference.com/w/cpp/utility/hash`