Programación Avanzada y Estructuras de Datos

6. Conjuntos y mapas

Víctor M. Sánchez Cartagena

Grado en Ingeniería en Inteligencia Artificial
Dep. Lenguajes y Sistemas Informáticos
Universidad de Alicante

27 de noviembre de 2024

Índice

Ejemplo introductorio

Queremos contar el número de palabras *diferentes* que hay en un texto. Se puede iterar fácilmente sobre las palabras de un texto con el operador >>. ¿Qué tipos de datos podríamos utilizar?

```
ifstream f;
    f.open("texto.txt");
    if(f.is_open()){
            string s;
            f >> s;
            while(!f.eof()){
                cout << "Palabra: " << s << endl;
                //Añadir palabra a tipo de datos
                f>>s;
            }
            f.close();
            //Imprimir número de palabras diferentes
    }
```

Ejemplo introductorio

Queremos contar el número de palabras *diferentes* que hay en un texto. Se puede iterar fácilmente sobre las palabras de un texto con el operador >>. ¿Qué tipos de datos podríamos utilizar?

Ejemplo introductorio

Queremos contar el número de palabras *diferentes* que hay en un texto. Se puede iterar fácilmente sobre las palabras de un texto con el operador `>>`. ¿Qué tipos de datos podríamos utilizar?

- Vector (desordenado), agregando cada palabra al final, comprobando antes que no estaba
- Vector (ordenado), agregando cada palabra de manera que queden ordenadas, comprobando antes que no estaba
- Árbol AVL

Ejemplo introductorio

Solución con vector desordenado:

```
ifstream f;
f.open("texto.txt");
if(f.is_open()){
        vector<string> v;
        string s;
        f >> s;
        while(!f.eof()){
            cout << "Palabra: " << s << endl;
            //Añadir palabra a tipo de datos
            int found=false;
            for(int i=0; i<v.size();i++)
                if(v[i]==s){
                    found=true;
                    break;}
            if (! found)
                v.push_back(s);
            f>>s;
        }
        f.close();
        //Imprimir número de palabras diferentes
        cout << "Hay " << v.size() << " palabras diferentes" << endl;
}
```

Ejemplo introductorio

Pregunta

¿Cuál es la complejidad temporal de comprobar si un elemento no está e insertarlo si es necesario? ¿y de obtener el número de elementos final?

Operación	Caso mejor	Caso peor
vector desordenado: insertar		
vector desordenado: contar		
vector ordenado: insertar		
vector ordenado: contar		
árbol AVL: insertar		
árbol AVL: contar		

*Si se modifica ligeramente para almacenar en un atributo entero el número de elementos

Ejemplo introductorio

Pregunta

¿Cuál es la complejidad temporal de comprobar si un elemento no está e insertarlo si es necesario? ¿y de obtener el número de elementos final?

Operación	Caso mejor	Caso peor
vector desordenado: insertar	$\Omega(1)$	$O(n)$
vector desordenado: contar	$\Omega(1)$	$O(1)$
vector ordenado: insertar	$\Omega(1)$	$O(n)$
vector ordenado: contar	$\Omega(1)$	$O(1)$
árbol AVL: insertar	$\Omega(1)$	$O(\log(n))$
árbol AVL: contar	$\Omega(1)$	$O(1)$*

*Si se modifica ligeramente para almacenar en un atributo entero el número de elementos

Ejemplo introductorio

¿Por qué no hacemos que el tipo de datos (clase) contenga una operación para insertar un elemento únicamente si no está?

```
ifstream f;
f.open("texto.txt");
if(f.is_open()){
        set<string> st;
        string s;
        f >> s;
        while(!f.eof()){
            cout << "Palabra: " << s << endl;
            st.insert(s);
            f>>s;
        }
        f.close();
        cout << "Hay " << st.size() << " palabras distintas" << endl;
}
```

TAD conjunto

Especificación del TAD conjunto

Definición: Colección de *n* elementos almacenados sin un orden definido, tal que todos los elementos son distintos

TAD conjunto

Especificación del TAD conjunto

Operaciones:

- Obtiene el número de elementos almacenados en el conjunto

```
int size() const;
```

- Añade el elemento `e` al conjunto.

```
void insert(const Elem &e);
```

- Devuelve `true` si el elemento `e` se encuentra en el conjunto y `false` en caso contrario

```
bool find(const Elem &e) const;
```

- Elimina el elemento `e`. Devuelve `true` si el elemento estaba en el conjunto y `false` en caso contrario

```
bool erase(const Elem& e);
```

Especificación del TAD conjunto

Operaciones:

- Devuelve un iterador que apunta al primer elemento. Devuelve el mismo valor que `end()` si el conjunto está vacío

```
Iterador begin() const;
```

- Devuelve un iterador que apunta al elemento imaginario que se encuentra tras el último elemento del conjunto

```
Iterador end() const;
```

- Devuelve el elemento en la posición aputada por el iterador `it`

```
Elem get(Iterador it) const;
```

TAD conjunto

Especificación del TAD conjunto

Ejemplos (conjunto inicialmente vacío):

Operación	Salida	Contenido del conjunto
`insert(7)`		
`insert(4)`		
`size()`		
`insert(4)`		
`size()`		
`get(c.begin())`		
`find(7)`		
`find(8)`		
`erase(8)`		
`erase(7)`		
`size()`		

TAD conjunto

Especificación del TAD conjunto

Ejemplos (conjunto inicialmente vacío):

Operación	**Salida**	**Contenido del conjunto**
`insert(7)`	-	$\{7\}$
`insert(4)`		
`size()`		
`insert(4)`		
`size()`		
`get(c.begin())`		
`find(7)`		
`find(8)`		
`erase(8)`		
`erase(7)`		
`size()`		

TAD conjunto

Especificación del TAD conjunto

Ejemplos (conjunto inicialmente vacío):

Operación	Salida	Contenido del conjunto
`insert(7)`	-	{7}
`insert(4)`	-	{7,4}
`size()`		
`insert(4)`		
`size()`		
`get(c.begin())`		
`find(7)`		
`find(8)`		
`erase(8)`		
`erase(7)`		
`size()`		

TAD conjunto

Especificación del TAD conjunto

Ejemplos (conjunto inicialmente vacío):

Operación	**Salida**	**Contenido del conjunto**
`insert(7)`	-	{7}
`insert(4)`	-	{7,4}
`size()`	2	{7,4}
`insert(4)`		
`size()`		
`get(c.begin())`		
`find(7)`		
`find(8)`		
`erase(8)`		
`erase(7)`		
`size()`		

TAD conjunto

Especificación del TAD conjunto

Ejemplos (conjunto inicialmente vacío):

Operación	**Salida**	**Contenido del conjunto**
`insert(7)`	-	{7}
`insert(4)`	-	{7,4}
`size()`	2	{7,4}
`insert(4)`	-	{7,4}
`size()`		
`get(c.begin())`		
`find(7)`		
`find(8)`		
`erase(8)`		
`erase(7)`		
`size()`		

TAD conjunto

Especificación del TAD conjunto

Ejemplos (conjunto inicialmente vacío):

Operación	**Salida**	**Contenido del conjunto**
`insert(7)`	-	{7}
`insert(4)`	-	{7,4}
`size()`	2	{7,4}
`insert(4)`	-	{7,4}
`size()`	2	{7,4}
`get(c.begin())`		
`find(7)`		
`find(8)`		
`erase(8)`		
`erase(7)`		
`size()`		

Especificación del TAD conjunto

Ejemplos (conjunto inicialmente vacío):

Operación	Salida	Contenido del conjunto
`insert(7)`	-	{7}
`insert(4)`	-	{7,4}
`size()`	2	{7,4}
`insert(4)`	-	{7,4}
`size()`	2	{7,4}
`get(c.begin())`	7 o 4	{7,4}
`find(7)`		
`find(8)`		
`erase(8)`		
`erase(7)`		
`size()`		

TAD conjunto

Especificación del TAD conjunto

Ejemplos (conjunto inicialmente vacío):

Operación	**Salida**	**Contenido del conjunto**
`insert(7)`	-	{7}
`insert(4)`	-	{7,4}
`size()`	2	{7,4}
`insert(4)`	-	{7,4}
`size()`	2	{7,4}
`get(c.begin())`	7 o 4	{7,4}
`find(7)`	true	{7,4}
`find(8)`		
`erase(8)`		
`erase(7)`		
`size()`		

TAD conjunto

Especificación del TAD conjunto

Ejemplos (conjunto inicialmente vacío):

Operación	**Salida**	**Contenido del conjunto**
`insert(7)`	-	{7}
`insert(4)`	-	{7,4}
`size()`	2	{7,4}
`insert(4)`	-	{7,4}
`size()`	2	{7,4}
`get(c.begin())`	7 o 4	{7,4}
`find(7)`	true	{7,4}
`find(8)`	false	{7,4}
`erase(8)`		
`erase(7)`		
`size()`		

TAD conjunto

Especificación del TAD conjunto

Ejemplos (conjunto inicialmente vacío):

Operación	**Salida**	**Contenido del conjunto**
`insert(7)`	-	{7}
`insert(4)`	-	{7,4}
`size()`	2	{7,4}
`insert(4)`	-	{7,4}
`size()`	2	{7,4}
`get(c.begin())`	7 o 4	{7,4}
`find(7)`	true	{7,4}
`find(8)`	false	{7,4}
`erase(8)`	false	{7,4}
`erase(7)`		
`size()`		

TAD conjunto

Especificación del TAD conjunto

Ejemplos (conjunto inicialmente vacío):

Operación	**Salida**	**Contenido del conjunto**
`insert(7)`	-	{7}
`insert(4)`	-	{7,4}
`size()`	2	{7,4}
`insert(4)`	-	{7,4}
`size()`	2	{7,4}
`get(c.begin())`	7 o 4	{7,4}
`find(7)`	true	{7,4}
`find(8)`	false	{7,4}
`erase(8)`	false	{7,4}
`erase(7)`	true	{4}
`size()`		

Especificación del TAD conjunto

Ejemplos (conjunto inicialmente vacío):

Operación	Salida	Contenido del conjunto
`insert(7)`	-	{7}
`insert(4)`	-	{7,4}
`size()`	2	{7,4}
`insert(4)`	-	{7,4}
`size()`	2	{7,4}
`get(c.begin())`	7 o 4	{7,4}
`find(7)`	true	{7,4}
`find(8)`	false	{7,4}
`erase(8)`	false	{7,4}
`erase(7)`	true	{4}
`size()`	1	{4}

TAD conjunto

Especificación del TAD conjunto

Operaciones (conjunto *fusionable*):

- Reemplaza al conjunto actual (A) con la unión de A y B: $A \leftarrow A \cup B$

```
void union(const Conjunto& B);
```

- Reemplaza al conjunto actual (A) con la intersección de A y B: $A \leftarrow A \cap B$

```
void interseccion(const Conjunto& B);
```

Implementación del TAD conjunto

Determinación de la representación:

- Vector desordenado
- Vector ordenado
- Árbol AVL
- Vector de bits

Implementación del TAD conjunto

Determinación de la representación:

- Complejidad espacial (n: número de elementos almacenados en el conjunto; u: tamaño conjunto universal):
  - Vector desordenado:
  - Vector ordenado:
  - Árbol AVL:
  - Vector de bits:

Implementación del TAD conjunto

Determinación de la representación:

- Complejidad espacial (n: número de elementos almacenados en el conjunto; u: tamaño conjunto universal):
  - Vector desordenado: $O(n)$
  - Vector ordenado: $O(n)$
  - Árbol AVL: $O(n)$
  - Vector de bits: $O(u)$

Implementación del TAD conjunto

Determinación de la representación:

- Complejidad temporal (n: número de elementos almacenados en el conjunto; u: tamaño conjunto universal):

Operación	v. desordenado	v. ordenado	AVL	v. bits
`find`				
`insert`				
`union`				

Implementación del TAD conjunto

Determinación de la representación:

- Complejidad temporal (n: número de elementos almacenados en el conjunto; u: tamaño conjunto universal):

Operación	v. desordenado	v. ordenado	AVL	v. bits
`find`	$O(n)$	$O(\log(n))$	$O(\log(n))$	$O(1)$
`insert`	$O(n)$	$O(n)$	$O(\log(n))$	$O(1)$
`union`	$O(n^2)$	$O(n)$	$O(n \cdot \log(n))$	$O(u)$

Índice

Tablas hash

Pregunta

¿Podemos reducir la complejidad espacial del vector de bits sin renunciar a la complejidad temporal constante para búsqueda e inserción?

Tablas hash

Pregunta

¿Podemos reducir la complejidad espacial del vector de bits sin renunciar a la complejidad temporal constante para búsqueda e inserción?

Sí, empleando una **tabla hash**

Tablas hash

- Se divide el conjunto universal en N subconjuntos diferentes.
- Tabla hash: **vector** en el que a cada posición le corresponde un subconjunto del conjunto universal

Para poder mantener tiempos constantes de búsqueda e inserción necesitamos:

- Decidir qué subconjunto del conjunto universal corresponde a cada posición. **Función de hash**: objeto a almacenar $\rightarrow$ posición que le corresponde en la tabla hash
- Un "plan B" para cuando la posición que corresponde a un objeto está ocupada: **estrategia de redispersión**

Función de hash

- Si los elementos a almacenar en la tabla hash son enteros, simplemente se emplea la siguiente **función de compresión**, que garantiza un valor en $[0, N-1]$

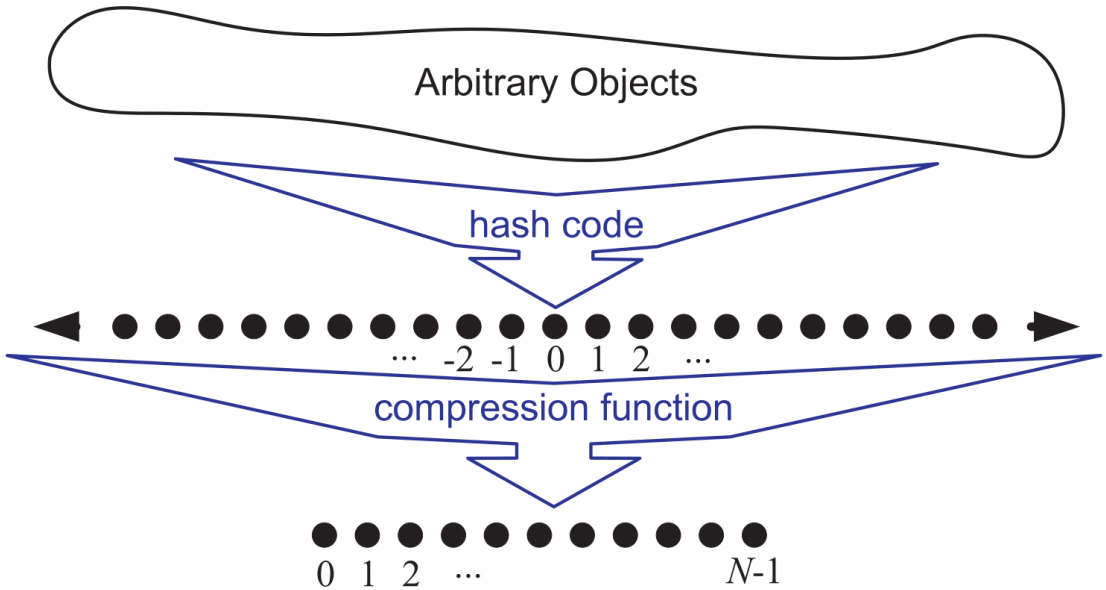
$$h(x) = |x| \text{ mód } N$$

  donde:
  - x: elemento a almacenar en la tabla
  - $h(x)$: posición que le corresponde
  - N: tamaño del vector (tabla hash)
- Ejemplos ($N = 11$):
  - $h(1) = 1 \text{ mód } 11 = 1$
  - $h(200) = 200 \text{ mód } 11 = 2$
  - $h(-30) = 30 \text{ mód } 11 = 8$
- Existen otras funciones de compresión más sofisticadas, como MAD (*multiply add and divide*)

Función de hash

- Si los elementos a almacenar no son enteros, es necesario convertirlos a enteros antes de aplicar la función de compresión
- El valor entero asociado a un objeto abitrario se denomina **código hash**

Función hash = código hash + función de compresión



fuente: Goodrich, M. T., Tamassia, R., & Mount, D. M. (2011). Data structures and algorithms in C++. John Wiley & Sons.

Función de hash

Características de una buena función de hash:

- Se puede calcular rápidamente
- Distribuye los elementos del conjunto universal de manera uniforme en las posiciones de la tabla hash

Pregunta

Queremos guardar cadenas de caracteres (`string`) en una tablas hash. Sabemos que nuestras cadenas contendrán palabras en español. Utilizamos el código ASCII del primer caracter de la cadena como código hash. ¿Es una buena función de hash?¿Se puede mejorar?

Función de hash

Características de una buena función de hash:

- Se puede calcular rápidamente
- Distribuye los elementos del conjunto universal de manera uniforme en las posiciones de la tabla hash

Pregunta

Queremos guardar cadenas de caracteres (`string`) en una tablas hash. Sabemos que nuestras cadenas contendrán palabras en español. Utilizamos el código ASCII del primer caracter de la cadena como código hash. ¿Es una buena función de hash?¿Se puede mejorar?

No. La frecuencia de la primera letra de las palabras no es uniforme: pocas palabras empiezan por x o y, por ejemplo. Habría que usar información sobre todos los caracteres de la cadena

Estrategia de redispersión

- **Colisión**: al insertar x, la casilla $h(x)$ está ocupada
- **Estrategia de redispersión**: ¿qué hago cuando se produce una colisión?

Dos familias de estrategias:

- Hash cerrado (*Open Addressing*)
- Hash abierto (*Separate Chaining*)

Estrategia de redispersión: hash cerrado

- Cuando la posición $h(x)$ está ocupada (colisión), se reintenta en las posiciones $h_1(x)$, $h_2(x)$, ..., $h_{N-1}(X)$
- $h_i(x) \rightarrow$ función de redispersión para el reintento número i
- Hay $N - 1$ funciones de redispersión diferentes
- Si tras probar las $N - 1$ funciones de redispersión, no es posible insertar en ninguna, la tabla está llena
- Una buena función de redispersión es imprescincible para obtener una complejidad promedio $\Theta(1)$ en búsquedas

Estrategia de redispersión: hash cerrado

Estrategia de redispersión lineal

- La estrategia más simple
- Se intenta insertar en la siguiente posición de la tabla
- Al llegar al final se vuelve al principio
- $h_i(x) = h_{i-1}(x) + 1 \mod N$ $(h_0(x) = h(x))$
- $h_i(x) = h(x) + i \mod N$

Estrategia de redispersión: hash cerrado

Ejemplo

Inserta los elementos 23, 14, 9, 6, 30, 12, 18 en una tabla hash con $N = 7$ y estrategia de redispersión lineal (hash cerrado). Cuenta los intentos (número de accesos a la tabla) para cada inserción

Estrategia de redispersión: hash cerrado

Ejemplo

Inserta los elementos 23, 14, 9, 6, 30, 12, 18 en una tabla hash con $N = 7$ y estrategia de redispersión lineal (hash cerrado). Cuenta los intentos (número de accesos a la tabla) para cada inserción

14	18	23	9	30	12	6
0	1	2	3	4	5	6

14 intentos en total

Estrategia de redispersión: hash cerrado

Pregunta

¿Cuańtos intentos son necesarios para insertar el valor 25 en la tabla anterior?¿Cuál es la complejidad de la inserción en el peor caso?

14	18	23	9	30	12	6
0	1	2	3	4	5	6

Estrategia de redispersión: hash cerrado

Pregunta

¿Cuańtos intentos son necesarios para insertar el valor 25 en la tabla anterior?¿Cuál es la complejidad de la inserción en el peor caso?

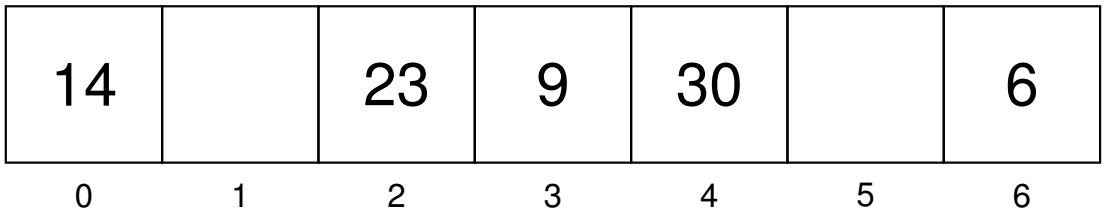
14	18	23	9	30	12	6
0	1	2	3	4	5	6

7 intentos $\rightarrow O(n)$ ($n =$ número de elementos del conjunto)

Estrategia de redispersión: hash cerrado

Pregunta

¿Qué pasa si intentamos buscar el 30 en esta tabla? ¿Cómo sería el algoritmo de búsqueda en una tabla hash con dispersión cerrada?¿Y su complejidad en el peor caso?



Estrategia de redispersión: hash cerrado

Pregunta

¿Qué pasa si intentamos buscar el 30 en esta tabla? ¿Cómo sería el algoritmo de búsqueda en una tabla hash con dispersión cerrada?¿Y su complejidad en el peor caso?

14		23	9	30		6
0	1	2	3	4	5	6

- $h(30) = 2$; en la casilla 2 está el 23; pero el 30 está sí en la tabla

Estrategia de redispersión: hash cerrado

Pregunta

¿Qué pasa si intentamos buscar el 30 en esta tabla? ¿Cómo sería el algoritmo de búsqueda en una tabla hash con dispersión cerrada?¿Y su complejidad en el peor caso?

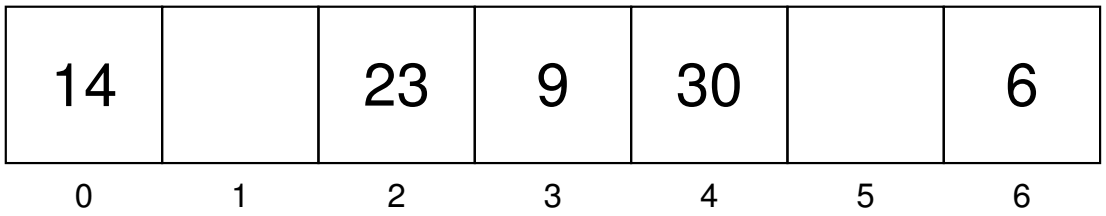
14		23	9	30		6
0	1	2	3	4	5	6

- $h(30) = 2$; en la casilla 2 está el 23; pero el 30 está sí en la tabla
- Hay que buscar en las casillas $h_i(x)$ hasta encontrar el elemento o llegar a una casilla vacía $\rightarrow O(n)$

Estrategia de redispersión: hash cerrado

Pregunta

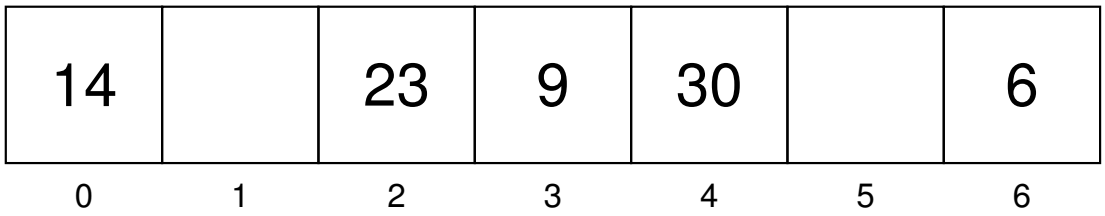
¿Qué pasa si borramos el 9 y luego intentamos buscar el 30? ¿Cómo solucionarías el problema?



Estrategia de redispersión: hash cerrado

Pregunta

¿Qué pasa si borramos el 9 y luego intentamos buscar el 30? ¿Cómo solucionarías el problema?



- La búsqueda del 30 se detendría en la casilla 3

Estrategia de redispersión: hash cerrado

Pregunta

¿Qué pasa si borramos el 9 y luego intentamos buscar el 30? ¿Cómo solucionarías el problema?

14		23	9	30		6
0	1	2	3	4	5	6

- La búsqueda del 30 se detendría en la casilla 3
- Los borrados se marcan con un valor especial: la casilla cuenta como vacía para la inserción pero como llena para la búsqueda:

14		23	X	30		6
0	1	2	3	4	5	6

Estrategia de redispersión: hash cerrado

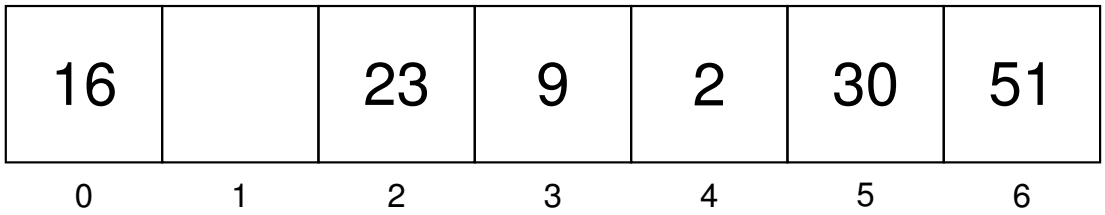
Ejercicio

Inserta los elementos 23, 9, 2, 30, 51, 16 en una tabla hash con $N = 7$ y estrategia de redispersión lineal (hash cerrado). Cuenta los intentos (número de accesos a la tabla) para cada inserción

Estrategia de redispersión: hash cerrado

Ejercicio

Inserta los elementos 23, 9, 2, 30, 51, 16 en una tabla hash con $N = 7$ y estrategia de redispersión lineal (hash cerrado). Cuenta los intentos (número de accesos a la tabla) para cada inserción



1, 2, 3, 4, 5, y 6 intentos respectivamente. Total: 21

Estrategia de redispersión: hash cerrado

- x e y son elementos **sinónimos** si y sólo si $h(x) = h(y)$
- La estrategia de redispersión lineal hace que el número de intentos se incremente drásticamente cuando hay muchos elementos sinónimos
- Este fenómeno se llama **amontonamiento**

Estrategia de redispersión: hash cerrado

Estrategia de redispersión aleatoria

- El siguiente intento se realiza c casillas más adelante (en lugar de una)
- c y N no deben tener factores primos comunes mayores que 1
- $h_i(x) = h_{i-1}(x) + c \mod N \ (h_0(x) = h(x))$
- $h_i(x) = h(x) + c \cdot i \mod N$
- Continúa habiendo amontonamiento

Estrategia de redispersión: hash cerrado

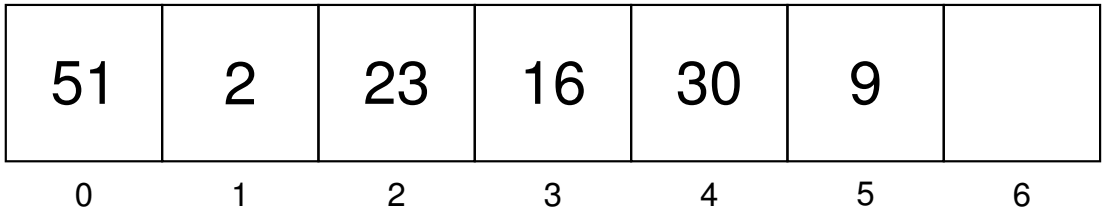
Ejercicio

Inserta los elementos 23, 9, 2, 30, 51, 16 en una tabla hash con $N = 7$ y estrategia de redispersión aleatoria con $c = 3$ (hash cerrado). Cuenta los intentos (número de accesos a la tabla) para cada inserción

Estrategia de redispersión: hash cerrado

Ejercicio

Inserta los elementos 23, 9, 2, 30, 51, 16 en una tabla hash con $N = 7$ y estrategia de redispersión aleatoria con $c = 3$ (hash cerrado). Cuenta los intentos (número de accesos a la tabla) para cada inserción



1, 2, 3, 4, 5, y 6 intentos respectivamente. Total: 21

Estrategia de redispersión: hash cerrado

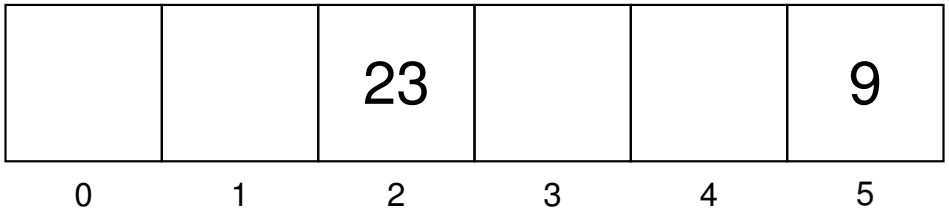
Ejercicio

Inserta los elementos 23, 9, 2, 30, 51, 16 en una tabla hash con $N = 6$ y estrategia de redispersión aleatoria con $c = 3$ (hash cerrado). Cuenta los intentos (número de accesos a la tabla) para cada inserción

Estrategia de redispersión: hash cerrado

Ejercicio

Inserta los elementos 23, 9, 2, 30, 51, 16 en una tabla hash con $N = 6$ y estrategia de redispersión aleatoria con $c = 3$ (hash cerrado). Cuenta los intentos (número de accesos a la tabla) para cada inserción



Sólo se pueden insertar 23 y 9. 1 y 2 intentos respectivamente

Estrategia de redispersión: hash cerrado

Estrategia de redispersión con segunda función hash

- El siguiente intento se realiza un número variable $k(x)$ de casillas más adelante
- $k(x)$ depende del elemento a ser insertado
- $k(x) = (x \mod N - 1) + 1$
- N debe ser primo para evitar que $k(x)$ tenga factores primos en común con N
- $h_i(x) = h_{i-1}(x) + k(x) \mod N$ $(h_0(x) = h(x))$
- $h_i(x) = h(x) + k(x) \cdot i \mod N$
- Evita el amontonamiento

Estrategia de redispersión: hash cerrado

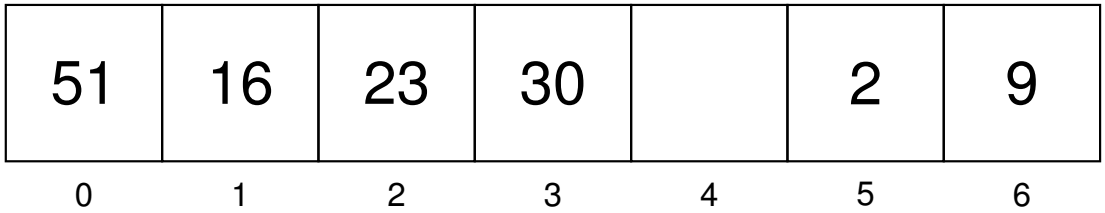
Ejercicio

Inserta los elementos 23, 9, 2, 30, 51, 16 en una tabla hash con $N = 7$ y estrategia de redispersión con segunda función hash (hash cerrado). Cuenta los intentos (número de accesos a la tabla) para cada inserción

Estrategia de redispersión: hash cerrado

Ejercicio

Inserta los elementos 23, 9, 2, 30, 51, 16 en una tabla hash con $N = 7$ y estrategia de redispersión con segunda función hash (hash cerrado). Cuenta los intentos (número de accesos a la tabla) para cada inserción



1, 2, 2, 2, 4, y 5 intentos respectivamente. Total: 16

Estrategia de redispersión: hash cerrado

Ejercicio

Inserta los elementos 23, 14, 9, 6, 30, 12, 18 en una tabla hash con $N = 7$ y estrategia de redispersión segunda función hash (hash cerrado). Cuenta los intentos (número de accesos a la tabla) para cada inserción

Estrategia de redispersión: hash cerrado

Ejercicio

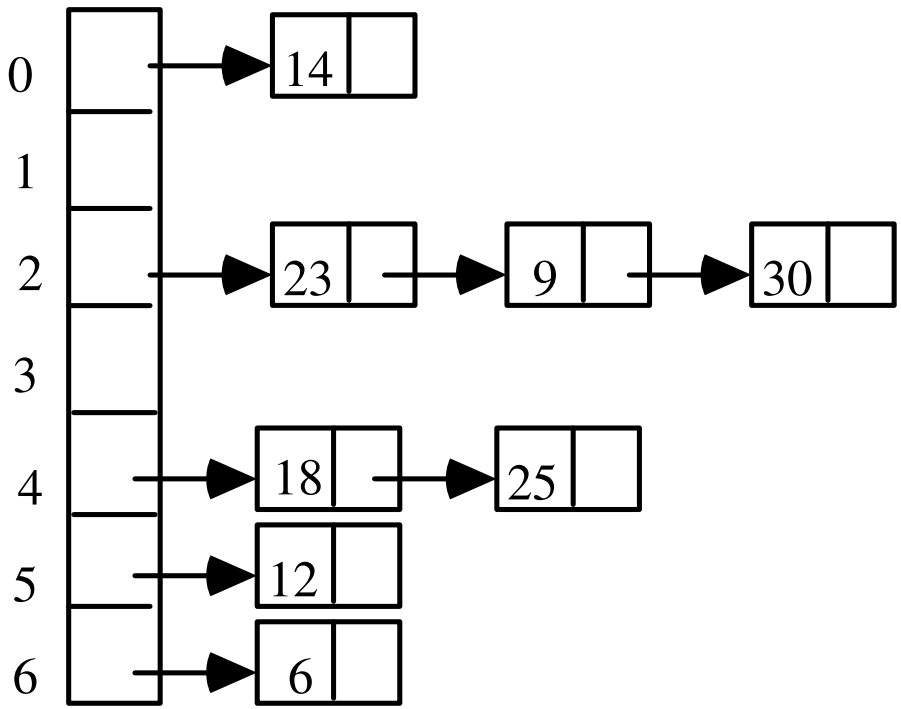
Inserta los elementos 23, 14, 9, 6, 30, 12, 18 en una tabla hash con $N = 7$ y estrategia de redispersión segunda función hash (hash cerrado). Cuenta los intentos (número de accesos a la tabla) para cada inserción

14	6	23	30	18	12	9
0	1	2	3	4	5	6

11 intentos en total

Estrategia de redispersión: hash abierto

- Las colisiones se resuelven mediante una lista enlazada (se inserta al final)
- Elimina las colisiones entre elementos no sinónimos
- El número de intentos se calcula como la longitud de la lista enlazada + 1



Estrategia de redispersión: hash abierto

Ejercicio

Inserta los elementos 23, 14, 9, 6, 30, 12, 18 en una tabla hash con $N = 7$ y estrategia de redispersión hash abierto. Cuenta los intentos (número de accesos a la tabla) para cada inserción

Estrategia de redispersión: hash abierto

Ejercicio

Inserta los elementos 23, 14, 9, 6, 30, 12, 18 en una tabla hash con $N = 7$ y estrategia de redispersión hash abierto. Cuenta los intentos (número de accesos a la tabla) para cada inserción

(el resultado es la tabla de la diapositiva anterior, sin el elemento 25).
Total: 10 intentos

Factor de carga

Sea n el número de elementos almacenados en la tabla hash, y N el tamaño de la tabla:

Factor de carga: $\lambda = \frac{n}{N}$

- Hash cerrado: $\lambda \in [0, 1]$
- Hash abierto: $\lambda \in [0, \infty)$

Complejidades promedio

- Las complejidades en el peor caso para insertar y buscar en una tabla hash están en $O(n)$
- Para factores de carga bajos y buenas funciones de hash, el coste promedio de ambas operaciones está en $\Theta(1)$
  - Hash cerrado: $\lambda < 0.5$
  - Hash abierto: $\lambda < 0.9$
- Cuando el factor de carga sobrepasa el límite, hay que crear una nueva tabla más grande (normalmente el doble): *rehashing*
- El coste amortizado promedio de insertar n elementos (incluyendo la creación de tantas nuevas tablas como sea necesario) está en $\Theta(n)$
- Las tablas hash suelen ser la implementación de referencia para conjuntos, salvo que se desee listar sus elementos en orden

Ejercicio adicional

Ejercicio

Inserta en una tabla hash ($N = 11$) los siguientes elementos: 23, 14, 10, 15, 3, 5, 7, 8, 36, 47, 4. Cuenta los intentos totales. Emplea las siguientes estrategies de redispersión:

- Hash cerrado, redispersión lineal (37 intentos)
- Hash cerrado, redispersión aleatoria con $c = 4$ (34 intentos)
- Hash cerrado, segunda función de hash (22 intentos)
- Hash abierto (18 intentos)

Índice

Conjuntos en C++ STL

Hay disponibles dos implementaciones del TAD conjunto en la biblioteca C++ STL:

- `set`: implementado como un árbol rojo-negro (variante del AVL). Al listar sus elementos, éstos aparecen en orden según su `operator<`
- `unordered_set`: implementado como una tabla hash con redispersión **abierta** (**hay que definir funcion hash para tipos de datos propios**) Al listar sus elementos, éstos no aparecen en ningún orden particular.

Características:

- Ambas soportan las mismas operaciones principales
- Son implementaciones de conjunto *no fusionables*
- Hay unas pocas operaciones dependientes de la implementación: por ejemplo, gestionar factor de carga y *rehashing*
- Más información: `https://cplusplus.com/reference/stl/`

Conjuntos en C++ STL: `set`

```cpp
#include<set>
using namespace std;

//...

//Es necesario que el tipo tenga un operator<
set<int> conjunto;

//conjunto vacío: 0 elementos
cout << conjunto.size() << endl;

//Inserción: reequilibrado automático
conjunto.insert(10);
conjunto.insert(20);
conjunto.insert(30);
conjunto.insert(40);
conjunto.insert(50);
conjunto.insert(20);

//5 elementos
cout << conjunto.size() << endl;
```

Conjuntos en C++ STL: `set`

- Búsqueda: `find` devuelve un iterador

```
if(conjunto.find(40)!=conjunto.end())
     cout << "encontrado" << endl;
else
     cout << "no encontrado" << endl;
```

- Búsqueda: `count` devuelve 1 o 0

```
if(conjunto.count(40)==1)
     cout << "encontrado" << endl;
else
     cout << "no encontrado" << endl;
```

- Borrado: `erase` devuelve el número de elementos borrados

```
cout <<conjunto.erase(40) << endl; //1
cout <<conjunto.erase(35) << endl; //0
```

Conjuntos en C++ STL: `set`

- Iteración en orden ascendente

```
//Imprime: 10 20 30 50
for (set<int>::iterator it=conjunto.begin();
                                    it!=conjunto.end(); ++it)
     cout << ' ' << *it;
cout << endl;
```

- Iteración en orden descendente

```
//Imprime: 50 30 20 10
for (set<int>::reverse_iterator it=conjunto.rbegin();
                                    it!=conjunto.rend(); ++it)
     cout << ' ' << *it;
cout << endl;
```

Conjuntos en C++ STL: `set`

- El tipo de datos `auto` permite simplificar mucho el código:

```
//Imprime: 10 20 30 50
for (auto it=conjunto.begin();
                                  it!=conjunto.end(); ++it)
     cout << ' ' << *it;
cout << endl;

//Imprime: 50 30 20 10
for (auto it=conjunto.rbegin();
                                  it!=conjunto.rend(); ++it)
     cout << ' ' << *it;
cout << endl;
```

Conjuntos en C++ STL: `set`

Pregunta

¿Cómo implementarías un método que devuelve el mayor elemento almacenado en un conjunto? ¿Y el menor?

Conjuntos en C++ STL: unordered_set

```cpp
#include<unordered_set>
using namespace std;

//...

//Es necesario que C++ sepa calcular la función hash del tipo
unordered_set<int> conjunto;

//conjunto vacío: 0 elementos
cout << conjunto.size() << endl;

//Inserción
conjunto.insert(10);
conjunto.insert(20);
conjunto.insert(30);
conjunto.insert(40);
conjunto.insert(50);
conjunto.insert(20);

//5 elementos
cout << conjunto.size() << endl;
```

Conjuntos en C++ STL: `unordered_set`

- Búsqueda: `find` devuelve un iterador

```
if(conjunto.find(40)!=conjunto.end())
      cout << "encontrado" << endl;
else
      cout << "no encontrado" << endl;
```

- Búsqueda: `count` devuelve 1 o 0

```
if(conjunto.count(40)==1)
      cout << "encontrado" << endl;
else
      cout << "no encontrado" << endl;
```

- Borrado: `erase` devuelve el número de elementos borrados

```
cout <<conjunto.erase(40) << endl; //1
cout <<conjunto.erase(35) << endl; //0
```

Conjuntos en C++ STL: `unordered_set`

- Iteración en orden arbitrario (depende de hash)

```
//Imprime: 50 30 20 10
for (unordered_set<int>::iterator it=conjunto.begin();
                        it!=conjunto.end(); ++it)
     cout << ' ' << *it;
cout << endl;
```

- No hay iteración en orden ascendente o descendente
- Se puede simplificar el código con `auto`

```
//Imprime: 50 30 20 10
for (auto it=conjunto.begin();
                        it!=conjunto.end(); ++it)
     cout << ' ' << *it;
cout << endl;
```

Conjuntos en C++ STL: `unordered_set`

- Tenemos acceso a las características de la tabla hash
  - Tamaño de la tabla: `bucket_count()`
  - Factor de carga: `load_factor()`
  - Posición donde está almacenado un elemento: `bucket(valor)`

```
cout << "Tamaño:" << conjunto.bucket_count() << endl;
cout << "Factor de carga:" << conjunto.load_factor() << endl;

//Imprime: 50(11) 30(4) 20(7) 10(10)
for (unordered_set<int>::iterator it=conjunto.begin();
                        it!=conjunto.end(); ++it)
     cout << ' ' << *it << '(' <<  conjunto.bucket(*it)  << ')';
cout << endl;
```

Conjuntos en C++ STL

Pregunta

Vuelve a implementar el programa que calcula el número de palabras distintas en un texto empleando conjuntos de la biblioteca STL de C++. ¿Qué sería más eficiente: `set` o `unordered_set`?

Índice

Ejemplo introductorio

Queremos contar el número de palabras diferentes que hay en un texto **y saber la frecuencia de cada una**. Se puede iterar fácilmente sobre las palabras de un texto con el operador >>. ¿Qué tipos de datos podríamos utilizar?

```
ifstream f;
    f.open("texto.txt");
    if(f.is_open()){
            string s;
            f >> s;
            while(!f.eof()){
                cout << "Palabra: " << s << endl;
                //Añadir palabra a tipo de datos e incrementar frecuencia
                f>>s;
            }
            f.close();
            //Imprimir número de palabras diferentes con su frecuencia
    }
```

Ejemplo introductorio

Queremos contar el número de palabras diferentes que hay en un texto **y saber la frecuencia de cada una**. Se puede iterar fácilmente sobre las palabras de un texto con el operador >>. ¿Qué tipos de datos podríamos utilizar?

Ejemplo introductorio

Queremos contar el número de palabras diferentes que hay en un texto **y saber la frecuencia de cada una**. Se puede iterar fácilmente sobre las palabras de un texto con el operador >>. ¿Qué tipos de datos podríamos utilizar?

Solución ineficiente: vector de palabras y vector de frecuencias

Ejemplo introductorio

Solución con dos vectores:

```
ifstream f;
f.open("texto.txt");
if(f.is_open()){
        vector<string> palabras;
        vector<int> frecuencias;
        string s;
        f >> s;
        while(!f.eof()){
            //Añadir palabra a tipo de datos e incrementar frecuencia
            int posicion=-1;
            for(int i=0; i<palabras.size();i++)
                if(palabras[i]==s){
                    posicion=i;
                    break;}
            if (posicion==-1){
                palabras.push_back(s); frecuencias.push_back(0);
                posicion=frecuencias.size()-1; }
            frecuencias[posicion]+=1;
            f>>s;
        }
        f.close();
}
```

Ejemplo introductorio

Solución con dos vectores:

```
 //.....
 f.close();

//Imprimir palabras diferentes:
cout << "Hay " << palabras.size() << " diferentes" << endl;

//Imprimir la frecuencia de cada una:
for(int i=0; i<palabras.size(); i++)
     cout << palabras[i] << ": " << frecuencias[i] << endl;
```

Ejemplo introductorio

Queremos contar el número de palabras diferentes que hay en un texto **y saber la frecuencia de cada una**. Se puede iterar fácilmente sobre las palabras de un texto con el operador >>. ¿Qué tipos de datos podríamos utilizar?

Ejemplo introductorio

Queremos contar el número de palabras diferentes que hay en un texto **y saber la frecuencia de cada una**. Se puede iterar fácilmente sobre las palabras de un texto con el operador >>. ¿Qué tipos de datos podríamos utilizar?

- Solución ineficiente: vector de palabras y vector de frecuencias $\rightarrow O(n)$ por cada inserción, siendo n el número de palabras diferentes

Ejemplo introductorio

Queremos contar el número de palabras diferentes que hay en un texto **y saber la frecuencia de cada una**. Se puede iterar fácilmente sobre las palabras de un texto con el operador >>. ¿Qué tipos de datos podríamos utilizar?

- Solución ineficiente: vector de palabras y vector de frecuencias $\rightarrow O(n)$ por cada inserción, siendo n el número de palabras diferentes
- Solución eficiente: ¿podemos asociar un valor (la frecuencia) a cada elemento de un conjunto? El tipo abstracto de datos resultante se llama **mapa**

TAD mapa

Especificación del TAD mapa

Definición: Colección de n pares clave-valor, de manera que cada uno de esos pares tiene una clave diferente. No hay un orden particular entre los pares.

Especificación del TAD mapa

Operaciones:

- Obtiene el número de pares clave-valor en el mapa

```
int size() const;
```

- Añade el par *c*-*v* si el mapa no tenía ningún par con clave *c*. En caso contrario, sustituye el valor del par existente por *v*

```
void put(const Clave &c, const Valor &v);
```

- Devuelve un iterador que apunta al par clave-valor con clave *c*. Si la clave no existe, devuelve un iterador que apunta a `end()`

```
IteradorMapa find(const Clave &c) const;
```

TAD mapa

Especificación del TAD mapa

- Elimina el par con clave `c`. Devuelve `true` si un par con clave `c` existía en el mapa y `false` en caso contrario

```
bool erase(const Clave &c);
```

- Devuelve un iterador que apunta al primer par. Devuelve el mismo valor que `end()` si el conjunto está vacío

```
Iterador begin() const;
```

- Devuelve un iterador que apunta al par imaginario que se encuentra tras el último par del mapa

```
Iterador end() const;
```

Especificación del TAD mapa

- Devuelve la clave del par aputado por el iterador `it`

```
Clave get(IteradorMapa it) const;
```

- Devuelve el valor del par aputado por el iterador `it`

```
Valor get(IteradorMapa it) const;
```

Especificación del TAD mapa

Ejemplos (mapa inicialmente vacío):

Operación	Salida	Contenido del mapa
`put(5,A)`		
`put(7,B)`		
`put(2,C)`		
`put(2,E)`		
`size()`		
`find(7)`		
`find(4)`		
`find(2)`		
`erase(5)`		
`erase(6)`		
`find(5)`		

Especificación del TAD mapa

Ejemplos (mapa inicialmente vacío):

Operación	Salida	Contenido del mapa
`put(5,A)`	-	{(5,A)}
`put(7,B)`		
`put(2,C)`		
`put(2,E)`		
`size()`		
`find(7)`		
`find(4)`		
`find(2)`		
`erase(5)`		
`erase(6)`		
`find(5)`		

Especificación del TAD mapa

Ejemplos (mapa inicialmente vacío):

Operación	Salida	Contenido del mapa
`put(5,A)`	-	{(5,A)}
`put(7,B)`	-	{(5,A), (7,B)}
`put(2,C)`		
`put(2,E)`		
`size()`		
`find(7)`		
`find(4)`		
`find(2)`		
`erase(5)`		
`erase(6)`		
`find(5)`		

Especificación del TAD mapa

Ejemplos (mapa inicialmente vacío):

Operación	Salida	Contenido del mapa
`put(5,A)`	-	{(5,A)}
`put(7,B)`	-	{(5,A), (7,B)}
`put(2,C)`	-	{(5,A), (7,B), (2,C)}
`put(2,E)`		
`size()`		
`find(7)`		
`find(4)`		
`find(2)`		
`erase(5)`		
`erase(6)`		
`find(5)`		

Especificación del TAD mapa

Ejemplos (mapa inicialmente vacío):

Operación	Salida	Contenido del mapa
`put(5,A)`	-	{(5,A)}
`put(7,B)`	-	{(5,A), (7,B)}
`put(2,C)`	-	{(5,A), (7,B), (2,C)}
`put(2,E)`	-	{(5,A), (7,B), (2,E)}
`size()`		
`find(7)`		
`find(4)`		
`find(2)`		
`erase(5)`		
`erase(6)`		
`find(5)`		

Especificación del TAD mapa

Ejemplos (mapa inicialmente vacío):

Operación	Salida	Contenido del mapa
`put(5,A)`	-	{(5,A)}
`put(7,B)`	-	{(5,A), (7,B)}
`put(2,C)`	-	{(5,A), (7,B), (2,C)}
`put(2,E)`	-	{(5,A), (7,B), (2,E)}
`size()`	3	{(5,A), (7,B), (2,E)}
`find(7)`		
`find(4)`		
`find(2)`		
`erase(5)`		
`erase(6)`		
`find(5)`		

Especificación del TAD mapa

Ejemplos (mapa inicialmente vacío):

Operación	Salida	Contenido del mapa
`put(5,A)`	-	{(5,A)}
`put(7,B)`	-	{(5,A), (7,B)}
`put(2,C)`	-	{(5,A), (7,B), (2,C)}
`put(2,E)`	-	{(5,A), (7,B), (2,E)}
`size()`	3	{(5,A), (7,B), (2,E)}
`find(7)`	p:(7,B)	{(5,A), (7,B), (2,E)}
`find(4)`		
`find(2)`		
`erase(5)`		
`erase(6)`		
`find(5)`		

Especificación del TAD mapa

Ejemplos (mapa inicialmente vacío):

Operación	Salida	Contenido del mapa
`put(5,A)`	-	{(5,A)}
`put(7,B)`	-	{(5,A), (7,B)}
`put(2,C)`	-	{(5,A), (7,B), (2,C)}
`put(2,E)`	-	{(5,A), (7,B), (2,E)}
`size()`	3	{(5,A), (7,B), (2,E)}
`find(7)`	p:(7,B)	{(5,A), (7,B), (2,E)}
`find(4)`	end	{(5,A), (7,B), (2,E)}
`find(2)`		
`erase(5)`		
`erase(6)`		
`find(5)`		

Especificación del TAD mapa

Ejemplos (mapa inicialmente vacío):

Operación	Salida	Contenido del mapa
`put(5,A)`	-	{(5,A)}
`put(7,B)`	-	{(5,A), (7,B)}
`put(2,C)`	-	{(5,A), (7,B), (2,C)}
`put(2,E)`	-	{(5,A), (7,B), (2,E)}
`size()`	3	{(5,A), (7,B), (2,E)}
`find(7)`	p:(7,B)	{(5,A), (7,B), (2,E)}
`find(4)`	end	{(5,A), (7,B), (2,E)}
`find(2)`	p:(2,E)	{(5,A), (7,B), (2,E)}
`erase(5)`		
`erase(6)`		
`find(5)`		

Especificación del TAD mapa

Ejemplos (mapa inicialmente vacío):

Operación	Salida	Contenido del mapa
`put(5,A)`	-	{(5,A)}
`put(7,B)`	-	{(5,A), (7,B)}
`put(2,C)`	-	{(5,A), (7,B), (2,C)}
`put(2,E)`	-	{(5,A), (7,B), (2,E)}
`size()`	3	{(5,A), (7,B), (2,E)}
`find(7)`	p:(7,B)	{(5,A), (7,B), (2,E)}
`find(4)`	end	{(5,A), (7,B), (2,E)}
`find(2)`	p:(2,E)	{(5,A), (7,B), (2,E)}
`erase(5)`	true	{(7,B), (2,E)}
`erase(6)`		
`find(5)`		

Especificación del TAD mapa

Ejemplos (mapa inicialmente vacío):

Operación	Salida	Contenido del mapa
`put(5,A)`	-	{(5,A)}
`put(7,B)`	-	{(5,A), (7,B)}
`put(2,C)`	-	{(5,A), (7,B), (2,C)}
`put(2,E)`	-	{(5,A), (7,B), (2,E)}
`size()`	3	{(5,A), (7,B), (2,E)}
`find(7)`	p:(7,B)	{(5,A), (7,B), (2,E)}
`find(4)`	end	{(5,A), (7,B), (2,E)}
`find(2)`	p:(2,E)	{(5,A), (7,B), (2,E)}
`erase(5)`	true	{(7,B), (2,E)}
`erase(6)`	false	{(7,B), (2,E)}
`find(5)`		

Especificación del TAD mapa

Ejemplos (mapa inicialmente vacío):

Operación	Salida	Contenido del mapa
`put(5,A)`	-	{(5,A)}
`put(7,B)`	-	{(5,A), (7,B)}
`put(2,C)`	-	{(5,A), (7,B), (2,C)}
`put(2,E)`	-	{(5,A), (7,B), (2,E)}
`size()`	3	{(5,A), (7,B), (2,E)}
`find(7)`	p:(7,B)	{(5,A), (7,B), (2,E)}
`find(4)`	end	{(5,A), (7,B), (2,E)}
`find(2)`	p:(2,E)	{(5,A), (7,B), (2,E)}
`erase(5)`	true	{(7,B), (2,E)}
`erase(6)`	false	{(7,B), (2,E)}
`find(5)`	end	{(7,B), (2,E)}

Implementación del TAD mapa

Determinación de la representación:

- Árbol AVL (o rojo-negro): cada nodo contiene la clave y puntero al valor. Los nodos se comparan por clave
- Tablas hash: cada posición contiene la clave y puntero al valor. La función hash se calcula sobre la clave

Índice

Mapas en C++ STL

Hay disponibles dos implementaciones del TAD mapa en la biblioteca C++ STL:

- `map`: implementado como un árbol rojo-negro (variante del AVL). Al listar sus pares, éstos aparecen en orden según el `operator<` de su clave
- `unordered_map`: implementado como una tabla hash con redispersión **abierta** (**hay que definir funcion hash para claves de tipos de datos propios**). Al listar sus pares, éstos no aparecen en ningún orden particular

Características:

- Ambas soportan las mismas operaciones principales
- Hay unas pocas operacions dependientes de la implementación: por ejemplo, gestionar factor de carga y *rehashing*
- Más información: `https://cplusplus.com/reference/stl/`

Mapas en C++ STL: `map`

```cpp
#include<map>
using namespace std;

//...

//Es necesario que el tipo de la clave tenga un operator<
map<int,string> mapa;

//mapa vacío: 0 elementos
cout << mapa.size() << endl;

//Inserción: reequilibrado automático
mapa.insert(pair<int,string>(10,"diez"));
mapa.insert(pair<int,string>(20,"viente"));
mapa.insert(pair<int,string>(30,"treinta"));
mapa.insert(pair<int,string>(40,"cuarenta"));
mapa.insert(pair<int,string>(50,"cincuenta"));
//No sobreescribe
mapa.insert(pair<int,string>(20,"veinte"));

//5 elementos
cout << mapa.size() << endl;
```

Mapas en C++ STL: `map`

- Búsqueda: `find` devuelve un iterador

```
if(mapa.find(40)!=mapa.end())
      cout << "encontrado" << endl;
else
      cout << "no encontrado" << endl;
```

- Se puede emplear el iterador para acceder al valor o modificarlo

```
auto it=mapa.find(40);
if(it!=mapa.end()){
      cout << "encontrado: "<< it->second << endl;
      it->second="Cuarenta";}
else
      cout << "no encontrado" << endl;
```

- Búsqueda: `count` devuelve 1 o 0

```
if(mapa.count(40)==1)
      cout << "encontrado" << endl;
else
      cout << "no encontrado" << endl;
```

Mapas en C++ STL: `map`

- El operador `[]` permite accesos para lectura y escritura. Si la clave no existe, crea una con un valor obtenido mediante el constructor por defecto del tipo valor

```
cout <<mapa[40] << endl; //Cuarenta
mapa[40]="cuarenta";
mapa[20]="veinte";
cout <<mapa[15] << endl; //Cadena vacía
```

- Borrado: `erase` devuelve el número de elementos borrados

```
cout <<mapa.erase(40) << endl; //1
cout <<mapa.erase(35) << endl; //0
```

Mapas en C++ STL: `map`

- Iteración en orden ascendente (permite modificaciones de valores)

```
//Imprime: 10->diez 15-> 20->veinte 30->treinta 50->cincuenta
for (auto it=mapa.begin();
                                    it!=mapa.end(); ++it){
     cout << ' ' << it->first << "->" << it->second;
     it->second=it->second+";";
}
cout << endl;
```

- Iteración en orden descendente (permite modificaciones de valores)

```
//Imprime: 50->cincuenta; 30->treinta; 20->veinte; 15->; 10->diez;
for (auto it=mapa.rbegin();
                                    it!=mapa.rend(); ++it){
     cout << ' ' << it->first << "->" << it->second;
}
cout << endl;
```

Mapas en C++ STL: `map`

- Como los iteradores permiten modificar el mapa, cuando el mapa sobre el que queremos iterar no se puede modificar (es constante), debemos utilizar **iteradores constantes**:

```
//Imprime: 10->diez; 15->; 20->veinte; 30->treinta; 50->cincuenta;
for (auto it=mapa.cbegin();
                                    it!=mapa.cend(); ++it)
     cout << ' ' << it->first << "->" << it->second;
cout << endl;

//Imprime: 50->cincuenta; 30->treinta; 20->veinte; 15->; 10->diez;
for (auto it=mapa.crbegin();
                                    it!=mapa.crend(); ++it){
     cout << ' ' << it->first << "->" << it->second;
}
cout << endl;
```

Mapas en C++ STL: unordered_map

```cpp
#include<unordered_map>
using namespace std;

//...

//Es necesario que C++ sepa calcular la función hash del tipo de la clave
unordered_map<int,string> mapa;

//mapa vacío: 0 elementos
cout << mapa.size() << endl;

//Inserción
mapa.insert(pair<int,string>(10,"diez"));
mapa.insert(pair<int,string>(20,"viente"));
mapa.insert(pair<int,string>(30,"treinta"));
mapa.insert(pair<int,string>(40,"cuarenta"));
mapa.insert(pair<int,string>(50,"cincuenta"));
//No sobreescribe
mapa.insert(pair<int,string>(20,"veinte"));

//5 elementos
cout << mapa.size() << endl;
```

Mapas en C++ STL: `unordered_map`

- Búsqueda: `find` devuelve un iterador

```
if(mapa.find(40)!=mapa.end())
      cout << "encontrado" << endl;
else
      cout << "no encontrado" << endl;
```

- Se puede emplear el iterador para acceder al valor o modificarlo

```
auto it=mapa.find(40);
if(it!=mapa.end()){
      cout << "encontrado: "<< it->second << endl;
      it->second="Cuarenta";}
else
      cout << "no encontrado" << endl;
```

- Búsqueda: `count` devuelve 1 o 0

```
if(mapa.count(40)==1)
      cout << "encontrado" << endl;
else
      cout << "no encontrado" << endl;
```

Mapas en C++ STL: `unordered_map`

- El operador `[]` permite accesos para lectura y escritura. Si la clave no existe, crea una con un valor obtenido mediante el constructor por defecto del tipo valor

```
cout <<mapa[40] << endl; //Cuarenta
mapa[40]="cuarenta";
mapa[20]="veinte";
cout <<mapa[15] << endl; //Cadena vacía
```

- Borrado: `erase` devuelve el número de elementos borrados

```
cout <<mapa.erase(40) << endl; //1
cout <<mapa.erase(35) << endl; //0
```

Mapas en C++ STL: unordered_map

- Iteración en orden arbitrario (permite modificaciones de valores)

```
//Imprime: 15-> 50->cincuenta 30->treinta 20->veinte 10->diez
for (auto it=mapa.begin();
                                    it!=mapa.end(); ++it){
     cout << ' ' << it->first << "->" << it->second;
     it->second=it->second+";";
}
cout << endl;
```

- Como los iteradores permiten modificar el mapa, cuando el mapa sobre el que queremos iterar no se puede modificar (es constante), debemos utilizar **iteradores constantes**:

```
//Imprime: 15->; 50->cincuenta; 30->treinta; 20->veinte; 10->diez;
for (auto it=mapa.cbegin();
                                   it!=mapa.cend(); ++it)
     cout << ' ' << it->first << "->" << it->second;
cout << endl;
```

Mapas en C++ STL: `unordered_map`

- Tenemos acceso a las características de la tabla hash
  - Tamaño de la tabla: `bucket_count()`
  - Factor de carga: `load_factor()`
  - Posición donde está almacenado un elemento: `bucket(clave)`

```
cout << "Tamaño:" << mapa.bucket_count() << endl;
cout << "Factor de carga:" << mapa.load_factor() << endl;

//Imprime:
//15->;(2) 50->cincuenta;(11) 30->treinta;(4)
// 20->veinte;(7) 10->diez;(10)
for (auto it=mapa.begin();
                        it!=mapa.end(); ++it)
     cout << ' ' << it->first << "->" << it->second << '(' <<  mapa.b
cout << endl;
```

Mapas en C++ STL

Ejercicio

Vuelve a implementar el programa que calcula el número de palabras distintas en un texto y sus frecuencias empleando mapas de la biblioteca STL de C++. ¿Qué sería más eficiente: `map` o `unordered_map`?

Ejercicio

Imagina que tenemos una serie de documentos de texto y queremos implementar un buscador que, dada una palabra, nos diga rápidamente en qué documento(s) se encuentra. ¿Cómo lo implementarías?