# Diccionarios.

**ED Grupo A. Profesor: Isabel Pita.**
**Tipo A**

**Nombre del alumno:**

1. Escribe dos formas diferentes de añadir un par `<clave,valor>` a un diccionario y explica sus diferencias. Indica si existe alguna diferencia en las instrucciones cuando se añaden los elementos en un diccionario ordenado respecto a cuando se añaden en uno no ordenado. Explica el coste de cada una de ellas sobre los dos tipos de diccionarios.

   *Respuesta:*

   Las instrucciones son las mismas si se está utilizando un diccionario ordenado (`map`) o uno sin orden (`unordered_map`).

   *a*) `tabla.insert({clave, valor});`

   Si se utiliza un `map`, el coste del método `insert`, es logarítmico respecto al número de elementos de la tabla, ya que estos diccionarios están implementados sobre un árbol binario de búsqueda equilibrado.

   Si se utiliza un `unordered_map` el coste del método `insert` es constante ya que estos diccionarios están implementados sobre una tabla hash, que suponemos bien diseñada.

   *b*) `tabla[clave] = valor;`

   Si se utiliza un `map`, el coste del método `[]`, es logarítmico respecto al número de elementos de la tabla, ya que estos diccionarios están implementados sobre un árbol binario de búsqueda equilibrado.

   Si se utiliza un `unordered_map` el coste del método `[]` es constante ya que estos diccionarios están implementados sobre una tabla hash, que suponemos bien diseñada.

   *Diferencias entre las dos formas.* El método `insert` añade el par `<clave, valor>` al diccionario cuando la clave no se encuentra. Si la clave ya se encuentra en el diccionario no se realiza ninguna acción.

   El método `[]` añade la clave al diccionario cuando esta no se encuentra y le asigna como valor el valor por defecto del tipo. Si la clave ya se encuentra en el diccionario se modifica su valor con el valor asignado en la instrucción.

   Por lo tanto `insert` solo permite añadir nuevos elementos, mientras que `[]` nos permite también modificarlos. En cuanto a la eficiencia, el método `insert` guarda una única vez el valor en el diccionario, mientras que `[]` primero lo inicializa al valor por defecto del tipo y después le asigna el valor que se le quiera dar.

2. Indica cuál es el coste de crear un diccionario ordenado y el coste de crear un diccionario no ordenado. Justifica tu respuesta.

   *Respuesta:*

   El coste de crear un diccionario ordenado es $\mathcal{O}(1)$, esto es debe a que un diccionario ordenado está implementado con un árbol binario de búsqueda y crear un árbol binario de búsqueda vacío tiene coste constante.

   El coste de crear un diccionario sin orden de tamaño `TAM` es lineal en el tamaño `TAM`. Esto se debe a que un diccionario sin orden está implementado por medio de una tabla hash y estas a su vez se implementan por medio de un vector. Reservar memoria para los `TAM` elementos de un vector tiene coste lineal en `TAM`.

3. Indica las ventajas e inconvenientes de utilizar un diccionario ordenado frente a uno no ordenado.

   *Respuesta:*

   Un diccionario ordenado nos permite recorrer las claves guardadas en orden, mientras que al recorrer las claves de un diccionario no ordenado no podemos suponer que se realiza en ningún orden.

   Las operaciones de añadir un elemento, buscar un elemento y borrar un elemento sobre un diccionario ordenado tienen coste logarítmico en el número de elementos del diccionario, mientras que sobre un diccionario no ordenado tienen coste constante.

   La operación de crear un diccionario ordenado tiene coste constante, mientras que esta operación tiene coste lineal en el tamaño de la tabla si se utiliza un diccionario no ordenado.

   Los diccionarios ordenados aprovechan mejor el espacio, ya que toda la memoria reservada está ocupada, mientras que los diccionarios no ordenados no ocupan toda la memoria reservada, sino solo hasta alcanzar el factor de carga definido.

   Los diccionarios no ordenados se redimensionan cuando se alcanza el factor de carga, mientras que los diccionarios ordenados no se redimensionan.

4. Si la ocupación de una tabla hash sobrepasa su factor de carga debemos redimensionarla. Explica como se redimensiona una tabla hash abierta e indica el coste de hacerlo. Justifica tu respuesta.

   *Respuesta:*

   Para redimensionar una tabla hash abierta, se reserva memoria para un vector de tamaño el primer primo mayor que el doble del tamaño de la tabla existente. A continuación se recorre la tabla original, calculando el nuevo hash de cada clave y situando los nodos de las listas en su nueva posición en la nueva tabla. No se debe copiar ningún valor, sino que se enlaza la memoria ya reservada en su nueva posición.

   El coste de redimensionar la tabla es $\mathcal{O}(max(k, N))$, siendo $k$ el coste de reservar la memoria de la nueva tabla y N el número de elementos de la tabla original. El coste depende de dos cantidades, por una parte se debe reservar la memoria del nuevo vector, lo que depende de la cantidad de memoria que se reserva. Por otra parte se deben desplazar a su nueva posición todas las claves de la tabla original, por lo que depende del número de elementos de la tabla. Cómo no se copian los valores, desplazar un nodo de una lista a una nueva lista solo requiere la asignación de varios punteros, por lo tanto el coste de desplazar cada nodo es constante.

   Como el número de elementos de la nueva tabla es mayor que el número de elementos de la tabla original, el máximo será el número de elementos de la nueva tabla.

5. Indica como se distribuyen los siguientes pares `<clave, valor>` sobre una tabla hash abierta de tamaño 7, si la función hash considerada asocia a cada carácter de la cadena su posición en el abecedario y su factor de carga es 0.8. Los datos se añaden en el orden dado.

   `<"Ada", 3>, <"Bea", 5>, <"Fede", 1>, <"Eva", 5>, <"Blas", 5>`

| a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 |

   *Respuesta:*

```
h("Ada") = 1 + 4 + 1 = 6 % 7 = 6.
h("Bea") = 2 + 5 + 1 = 8 % 7 = 1.
h("Fede") = 6 + 5 + 4 + 5 = 20 % 7 = 6.
h("Eva") = 5 + 22 + 1 = 28 % 7 = 0.
h("Blas") = 2 + 12 + 1 + 19 = 34 % 7 = 6.
```

   En la lista de la posición 0 se sitúa `<"Eva",5>`.

   En la lista de la posición 1 se sitúa `<"Bea",5>`.

En la lista de la posición 6 se sitúan Ada, Fede y Blas. Como en la implementación vista en clase, los nuevos elementos se añaden por el principio de la lista, la lista de la posición 6 será `<"Blas", 5>, <"Fede", 1>,<"Ada", 3>`.

6. Indica si los siguientes pares `<clave, valor>` se pueden guardar en una tabla hash cerrada de tamaño 7 con factor de carga 0,8, ajustado al valor menor (truncamiento). En caso de que haya que dimensionar la tabla indica con qué valor se realizaría el redimensionado, suponiendo que los valores se van a añadir en el orden dado.

   `<4,"xx">, <6,"yy">, <4, "zz">, <5, "bb">, <5, "cc">, <7, "vv">, <6, "ff">, <9, "jj">, <1,"ww">, <5, "ss">, <3, "kk">, <7, "ll">, <2, "ff">`.

   *Respuesta:*

   En una tabla hash con tamaño 7 y factor de carga 0.8 se pueden guardar 5 elementos.

   $$0{,}8 = \frac{NumElem}{7} \Rightarrow NumElem = 7 * 0{,}8 = 5{,}6.$$

   En la entrada los valores repetidos solo se guardan una vez, por lo tanto se van guardando: `<4,"xx">`, `<6,"yy"> <5, "bb"> <7, "vv"> <9, "jj">`

   El primer valor con el que se debe redimensionar la tabla es `<1,"ww">`.

7. (1 punto) Dado el siguiente fragmento de código indica qué se muestra por pantalla al ejecutarlo. Indica el coste de cada una de las instrucciones del código.

```
1  std::unordered_map<std::string, int> tabla;
2  tabla.insert({"amarillo", 4});
3  tabla.insert({"rojo", 6});
4  tabla.insert({"verde", 2});
5  std::pair<std::unordered_map<std::string, int>::iterator,bool> par1;
6  par1 = tabla.insert({"amarillo", 6});
7  if (par1.second) std::cout << "primero\n";
8  else std::cout << "segundo\n";
9  std::pair<std::unordered_map<std::string, int>::iterator,bool> par2;
10 par2 = tabla.insert({"naranja", 1});
11 if (par2.second) std::cout << "primero\n";
12 else std::cout << "segundo\n";
13 std::pair<std::unordered_map<std::string, int>::iterator,bool> par3;
14 par3 = tabla.insert({"naranja", 8});
15 std::cout << par3.first->first << ' ' << par3.first->second << '\n';
```

   *Respuesta:*

   segundo
   primero
   naranja 1

   Explicación:

   En la línea 6 la operación de insertar no tiene efecto ya que la clave `amarillo` no tiene efecto. La operación devuelve `false` y por ello se escribe `segundo`.

   En la línea 10 se añade el elemento `naranja` ya que no se encuentra en el diccionario. La operación devuelve `true` y por ello se escribe `primero`.

   En la línea 14 se intenta añadir la clave `naranja`. Cómo ya se encuentra en la tabla la operación no tiene efecto y el valor sigue siendo el que ya se encontraba en la tabla. Por ello se escribe el valor 1 en la clave `naranja`.

   La primera instrucción en que se declara la tabla tiene coste lineal en el tamaño inicial de la tabla, ya que se crea una tabla hash. El resto de instrucciones tienen coste constante ya que el método `insert` sobre el tipo `unordered_map` tiene coste constante.

8. (1 punto) Dado el siguiente fragmento de código indica qué se muestra por pantalla al ejecutarlo. Indica el coste de cada una de las instrucciones del código.

```
1     std::map<std::string, int> tabla;
2     tabla["amarillo"] = 4;
3     tabla["verde"] = 1;
4     tabla["rojo"] = 3;
5     tabla["amarillo"] = 6;
6     std::cout << tabla["amarillo"] << '\n';
7     std::map<std::string, int>::iterator it = tabla.find("verde");
8     if (it != tabla.end()) {
9         std::cout << it->first << ' ' << it->second << '\n';
10        it->second = 3;
11        std::cout << it->first << ' ' << it->second << '\n';
12    }
13    std::cout << tabla["verde"];
```

*Respuesta:*

6
verde 1
verde 3
3

Explicación:

En las líneas 2,3,y 4 se añaden al diccionario las claves `amarillo`, `verde`, y `rojo`. En la línea 5 se modifica el valor de la clave `amarillo` al valor 6 y en la línea 6 se escribe el nuevo valor de la clave `amarillo`.

En la línea 7 se obtiene un iterador a la clave `verde`. Si esta clave existe, en la línea 9 se escriben los valores de la clave y el valor de la clave `verde` a través de los valores del iterador.

En la línea 10 se modifica el valor asociado a la clave `verde` a través del iterador y en la línea 11 se escribe el nuevo valor de la clave `verde`.

En la línea 13 se escribe el valor de la clave `verde`.

El coste de los métodos `[]` y `find` sobre un diccionario ordenado (`map`) es lógarítmico en el número de elementos del diccionario. El resto de operaciones e instrucciones son de coste constante. Hay que tener en cuenta que el acceso a un valor con el operador `[]` es de coste logarítmico, aunque se encuentre en la instrucción de escritura.

9. (1 punto) Dado el siguiente fragmento de código indica qué se muestra por pantalla al ejecutarlo. Indica el coste de cada una de las instrucciones del código.

```
1     std::map<std::string, int> tabla;
2     tabla["amarillo"] = 4;
3     tabla["verde"] = 1;
4     tabla["rojo"] = 3;
5     std::map<std::string, int>::iterator it = tabla.find("verde");
6     std::cout << it->first << ' ' << it->second << '\n';
7     tabla.erase(it);
8     for (auto p : tabla)
9         std::cout << p.first << ' ' << p.second << '\n';
```

*Respuesta:*

verde 1
amarillo 4
rojo 3

En las líneas 2,3 y 4 se añaden las claves `amarillo`, `verde` y `rojo` al diccionario.

En la línea 5 se obtiene un iterador a la clave `verde` con el método `find`.

En la línea 6 se escriben los valores de la clave y valor apuntados por el iterador.

En la línea 7 se elimina la clave `verde` a través del iterador que se ha obtenido con el método `find`.

En las líneas 8 y 9 se muestran los valores de la tabla con un bucle que la recorre. Como el diccionario es un `map` el iterador del bucle recorre las claves en orden. Se muestran por lo tanto las claves `amarillo` y `rojo` en este orden. La clave `verde` se ha eliminado del diccionario.

El método `erase` elimina de la tabla el par `<clave, valor>` apuntado por el iterador.

El coste de los métodos `[]`y `find` sobre un diccionario ordenado (`map`) es lógarítmico en el número de elementos del diccionario. El resto de operaciones e instrucciones son de coste constante. Observad que el método `erase` tiene como argumento un iterador, por ello su coste es constante.

10. Explica si la siguiente afirmación es cierta o falsa. *En una tabla hash cerrada, el número de elementos que se pueden almacenar depende del método de resolución de colisiones empleado. En general se pueden almacenar más valores si se utilizan varias funciones hash para resolver las colisiones que si se utiliza una única función hash con búsqueda lineal de la siguiente posición libre.*

*Respuesta:*

La afirmación es falsa. El número de elementos que se almacenan en una tabla hash depende del tamaño de la tabla y del factor de carga admitida. El método de resolución de colisiones afecta a la forma en que se guardan los elementos en la tabla, y por lo tanto a las colisiones que se originarán. Cuando se produce una colisiónel nuevo elemento se guarda en una posición que depende del método de resolución de colisiones empleado, pero al final ocupa una posición, tanto si se ha producido una colisión como si no.