Tablas de Hash.

Vamos a pensar en el siguiente planteamiento: imaginemos que un jefe nos encarga una estructura de datos en la cual se van a guardar enteros [hasta 999 enteros (no precisamente 1000 datos)] y las operaciones que se van a hacer son la inserción, el borrado y la búsqueda. El jefe nos aclara que lo que le importa es el desempeño, es decir, que sea completamente eficiente. La respuesta que se nos ocurre en 30 segundos es aquella que involucra arreglos. Si sabemos que los números son entre 0 y 999, podemos implementar un arreglo con 999 casillas e insertar el elemento entero en su casilla correspondiente; en otras palabras, si el elemento a insertar es el 7, lo insertamos en la casilla 7, si es el 321, en la casilla 321. Esta misma lógica se puede seguir para la búsqueda y el borrado de los elementos. Vaya, son números que de entrada ya sabemos dónde van en un arreglo. Para implementar esta estructura podemos utilizar contadores [si se inserta más de una vez un número aumentamos contador (de entrada está en 0)] o incluso booleanos [si está es true, si no es false]. ¿Cuánto nos tardamos en insertar en una estructura así? Pues es un tiempo constante porque ya conocemos el índice donde tendríamos que insertar, buscar o eliminar en la estructura. No importa si metemos diez datos o un millón, el tiempo es el mismo. Pero hay que ser sinceros, aquí estamos haciendo bastantes restricciones que facilitan el problema. Lo que queremos es más o menos una estructura así.

Ahora bien, lo que queremos hacer es que para un dato que alguien quiera insertar en la estructura de datos, vamos a tratar de convertir ese dato de alguna forma en un número. En la posición indicada por ese número de la tabla [la cual va a ser implementada como un arreglo] es donde se va a insertar. Si lográramos hacer eso bien, se va a tener una estructura muy eficiente. Esta idea de transformar un dato arbitrario en número se hace mediante una función que recibe cualquier tipo de datos y da como resultado un entero; esta función se llama **función de hash**. Es importante decir que la estructura no es algo venido del cielo a resolver todos los problemas, sino que al ganar esta eficiencia también perdemos algunas cosas y no sirven para todo. Las **operaciones** que queremos hacer en esta estructura son operaciones de inserción, de borrado y de búsqueda.

Características *de la función de hash*.

- Que pueda mapear el dominio de los datos al rango del arreglo, es decir, tenemos que garantizar que el elemento quede en el rango del tamaño de la tabla, que no se salga.

- ¿Cómo podemos minimizar las colisiones? Por colisiones nos referimos a que dos o más datos diferentes puedan mapearse en la misma posición. Podemos lograr ese objetivo si la función puede distribuir los datos uniformemente en la tabla.
- Que sea determinista para que podamos recuperar los datos de la tabla, pues si se hace de manera aleatoria no vamos a saber exactamente dónde está el dato y vamos a tener que recorrer todo.
- Que su complejidad sea *O(1)*, es decir, que sea de complejidad constante. No tendría sentido que la función se tarde mucho.

Algunas funciones de hash.

- **Residuo**. Se utiliza para que el tamaño de la función quede entre 0 y tamaño 1.
- **Doblado**. Para texto: se toma el valor ASCII de las letras y se mezclan sus posiciones. Para entender el porqué de esta función solo hay que recordar el ejemplo del diccionario.
- **Cuadrado**. Se eleva el número al cuadrado y se toman los bits de en medio.
- **MD5, SHA1**. En el mundo de la seguridad y de la criptografía (recordemos el one-time pad) también se utilizan las funciones de hash. Estos convierten cualquier texto en una cadena de 128 o de 256 bits (no recordamos bien) y son *one-way*. Esto quiere decir que es muy fácil calcular el hash de un password, pero es muy complicado recuperar el password del hash. Esto hace que sea criptográficamente seguro. MD5 y SHA1 son funciones de estas. Incluso MD5 sirve para otras cosas: si se desea mandar un mensaje largo en internet se puede mandar el mensaje junto con el MD5, eso hace que se pueda verificar que el mensaje sea el mismo. Esta función de hash puede servir en algunos aspectos (algunos en general) para criptografía, pero en nuestro caso se utiliza para las tablas de hash. Es medio tardada la función, pero cumple con las propiedades solicitadas.

*Algo previo a las **colisiones***.

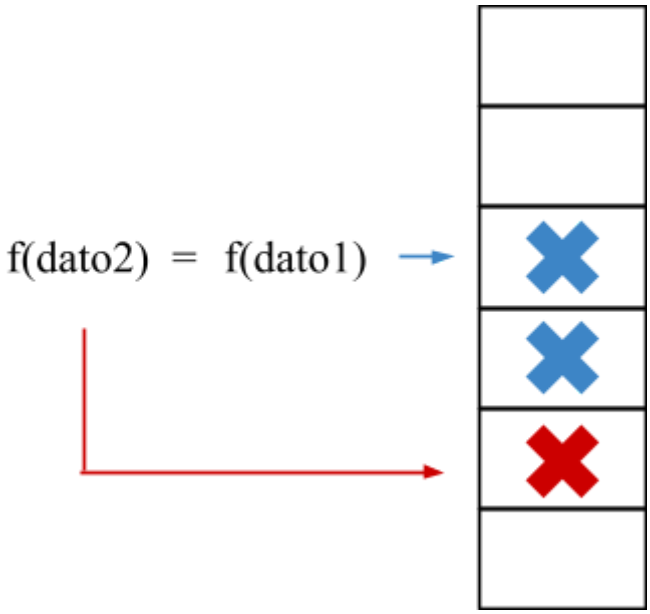
Ya que no podemos este ideal de que sea una función inyectiva (uno a uno), tenemos que sentarnos a pensar cuál es la solución más efectiva para este problema. La pregunta es: ¿qué tan probable es que exista una colisión? Esto lo podemos abordar de una manera mundana con el **Birthday attack**: ¿cuántas personas tengo que seleccionar de forma aleatoria para que la probabilidad de que haya una pareja que comparta cumpleaños sea de 0.5? Sinceramente el autor de este documento no tiene idea de cómo se hace porque no ha llegado a la materia de Cálculo de Probabilidades I, pero si se desea conocer cómo sacar el resultado, solo hay que ver la clase del día correspondiente o buscarlo en

internet. De todas maneras la respuesta es **n = 23** (realmente es 22.5, pero no podemos invitar media persona). Lo interesante de esto es que en clase se hizo el experimento de esta pregunta con 19 personas en el salón y después de preguntar a 4 personas, se encontró que sí había un cumpleaños repetido. Entonces esto prueba que las colisiones sí son probables.

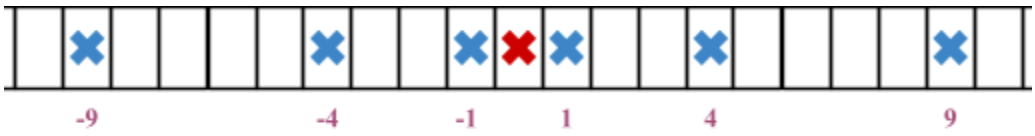
Resolución *de colisiones*.

Has dos estrategias:

- **Reubicación**. Busca otra posición en la tabla. Si se quiere insertar algo y la posición ya está ocupada, se tiene que buscar otra posición vacía para insertar el elemento. Hay diferentes maneras de lograr esto.
    a. **Lineal**. Se busca la siguiente posición disponible, módulo, el tamaño de la tabla. Este método puede degenerar muy rápido porque en el peor de los casos nos tardamos n veces, es decir, es de complejidad O(n). Lo que hace es que genera grupos de datos contiguos y entonces la tabla empieza a perder cierta uniformidad.



    b. **Cuadrática**. Se busca la siguiente posición de acuerdo a una función cuadrática. Si pensamos en una función como $f(i) = i^2$, donde i es el intento a verificar. Vaya, en el intento 1 checamos 1 posición abajo y 1 arriba, en el 2 checamos 4 abajo y 4 arriba, en el 3 checamos 9 abajo y 9 arriba; y así sucesivamente. Cada vez hacemos más saltos de acuerdo a una función cuadrática.



   c. **Rehasheo**. Se busca en la posición y si esa función está llena, se aplica otra función de hash sobre eso. Así se puede ir distribuyendo de una mejor manera. Se aplican múltiples funciones de hash encadenadas.

- **Encadenamiento**. Se almacenan listas ligadas (Linked List) en las casillas del arreglo, no solo un dato por cada casilla. Esto hace que tengamos la ventaja de que cuando insertemos el arreglo, tengamos una complejidad de *O(1)*, puesto a que aunque nos encontremos con una colisión, no se tendrá que buscar en otra casilla para insertar. Claro que se podría pensar "¿pero qué no podríamos tener una lista ligada con todos los elementos en ella?" Pero eso no podría pasar del todo, ya que por medio de la función de hash y el manejo de la expansión de la tabla, los elementos se van a distribuir más uniformemente para evitar el problema de tener una lista ligada con *n* elementos en ella.

  La idea es que, como queremos que funcione rápido, vamos a requerir que nuestra función de hash, en conjunto con el tamaño de la tabla que tenemos, genere colisiones uniformes (que no todas caigan en el mismo lugar para no tener listas demasiado largas, sino que más o menos se tengan listas del mismo tamaño) y que también no haya muchas de ellas [esto se va a controlar de alguna manera con el tamaño de la tabla]. Esto último se logra una vez que haya suficientes datos guardados en la estructura, que el siguiente dato implique un aumento en la capacidad para que las listas sean más chicas.

***Bonus** rápido.*

¿Cuál es la complejidad de insertar? Podríamos esperar que *O(1)*, pero ahora con las funciones de hash y los métodos de resolución de colisiones podemos tardarnos más porque de alguna manera vamos a tener que recorrer la tabla. En promedio buscamos que sea constante, pero también tenemos que pensar que tiene que ver el tamaño de la tabla y el número de datos que ya están insertados dentro de ella. Vaya, tiene que haber suficientes espacios vacíos para que tengamos una complejidad bastante razonable.

*¿Qué **no** podemos hacer con la tabla de hash?*

- Pues no podemos encontrar el máximo ni el mínimo.
- No podemos ordenar los datos.

Bloom filters.

Esta es una estructura de datos súper compacta en la que se utiliza una Tabla de Hash booleana; además es probabilística porque nos da un resultado correcto con cierta probabilidad [en serio, en todas las estructuras anteriores podríamos decir seguros si un dato se encuentra en ella o si no; aquí ya no estamos completamente seguros porque existen falsos positivos]. Lo que se hace es que teniendo un dato *k* y una función de hash *h(x)*, cuando se hace la operación *h(k) = c*, con *c* el índice correspondiente en la tabla de hash, esa casilla se pone como *true* (o 1 si trabajamos de manera "binaria"). El problema de esto es el hecho de que no está guardando los datos como tal [porque en una tabla de hash normal, encadenamos las cosas y ya está si hay una colisión], sino que se guarda un simple *true* o un *false* aún si ahí colisionaron más de un solo dato. Para manejar las colisiones en este caso, se cuenta con una segunda función de hash independiente para dirigirlo a otra casilla. Si la tabla es lo suficientemente grande, con este método se están tratando de "reducir" las colisiones o la probabilidad de las mismas. Lo interesante es que no solamente se tienen que tener dos funciones de hash, sino que se pueden tener *n* funciones de hash independientes. ¿Para qué hacemos eso? Pues si dos datos colisionan después de haber aplicado la función de hash, tal vez es menos probable que colisionen en las siguientes. Pero hasta eso, el punto de esta estructura no es minimizar las colisiones, entonces ¿cuál es?
La primera motivación es que se pueden representar un montón de datos de una manera muy compacta. Si se llena todo el arreglo de *trues*, eso significa que ya representamos todo el universo de posibles datos en un arreglo booleano. El punto es lo compacto, y claro que tiene sus problemas, pero más adelante veremos qué podemos hacer y qué no.

Entonces ¿**bloom filters**? Sí, una tabla de hash [arreglo] booleano suficientemente grande y vamos a tener *q* funciones de hash independientes [o lo más independientes que se puedan] para *representar* datos en ella.

Insertar.

- Se aplican las *q* funciones de hash sobre el dato [esperamos que nos regresen *q* (o menos) posiciones en el arreglo] y se pone *True* en las casillas o posiciones indicadas. Lo que parece ser extraño de este método es que cada dato se inserta en la misma tabla de hash.

Buscar.

- Se aplican los *q* filtros al dato. Si es que al aplicar alguno de estos *q* filtros, la casilla obtenida tiene un *false*, entonces significa que el dato no está en el arreglo. Si se terminan de aplicar los filtros y no hubo ninguno que contuviera un *false*, entonces el dato sí está en la estructura.

*¿Se puede **borrar**?*

No. Solamente podemos insertar y buscar.

Las molestias.

Después de todo esto, si el lector no está sintiendo molestia por esta estructura, debe decirse que qué curioso, porque al autor de este escrito hasta le causó comezón todo el cuerpo de pensar en todas las dudas y los "pero" existentes al tomar la clase. Vamos a dejarlo en claro: el causante de la incomodidad es no saber qué dato fue el que cambió la casilla de valor. ¿A qué nos referimos? Es sencillo: imaginemos que tenemos un arreglo de 10 casillas y que tenemos *k1* y *k2* como datos de entrada. Supongamos que *k1* "prendió" las casillas 3 y 7, mientras que *k2* "prendió" las casillas 5 y 8. Supongamos que se busca un dato *c* tal que al aplicar los filtros, este busca que las casillas 5 y 7 están prendidas. Aquí nos encontramos con que sí lo están, pero no porque se haya insertado antes el dato *c*, sino porque otros datos prendieron esas casillas. He aquí lo feo.

¿Entonces?

En esta estructura tenemos dos casos:

- Existen **falsos positivos**, es decir, elementos que 'yo' pienso que están, pero que no están. Nunca los insertamos en la estructura, pero pensaremos que sí está. Incluso se pueden calcular cuántos falsos positivos tenemos en la estructura dado el número de filtros, de casillas, etcétera.
- No existen **falsos negativos**. Es por ello que no vamos a borrar, porque eso implicaría que cambiáramos un *true* a un *false* y no queremos no poder que sí insertamos algo en la estructura.

¿Para qué sirve esto?

Para empezar, se debe decir que esto tiene complejidad de *O(1)*, evidentemente. Por ejemplo, Google ocupa esta estructura para saber si una página de internet es 'maliciosa' o no.

Esto abre la puerta a un mundo de algoritmos que son apasionantes: los *randomized algorithms*. Este es el conjunto de estrategias para resolver problemas que son intratables (no tenemos un algoritmo

eficiente para hacerlo), y se ocupa de la probabilidad para atacar el problema y encontrar la solución. Por ejemplo: para saber si un número es primo. Solo piénsalo, ¿no crees que es ineficiente estar dividiendo entre todos los primos anteriores conocidos? Claro, hay reglas de dedo muy sencillas, pero aún así. Es interesante ponerse a investigar.

Diferencia entre SkipList [randomizada] *y Bloom Filters* [probabilística].
La diferencia es que una usa números o secuencias de números aleatorios para su uso y lo que es probabilístico es su desempeño, pues sus resultados son deterministas. En la segunda, el resultado es probabilístico.