Programación II

Estructuras de datos avanzadas: Tabla Hash

Universidad Nacional de Rosario.
Facultad de Ciencias Exactas, Ingeniería y Agrimensura.

Introducción

Muchas aplicaciones requieren trabajar con tres operaciones básicas: **insertar**, **buscar**, y **eliminar**.

Ejemplos

- **Agenda telefónica:** el usuario encuentra números de teléfono (que no recuerda porque son muy largos) a partir del nombre de pila de la persona a quien quiere llamar.

  En estos casos el usuario puede añadir nuevos contactos ingresando el nombre con el que lo identificará y luego el número de teléfono a guardar. De un modo similar también puede borrar contactos existentes.

Introducción

Muchas aplicaciones requieren trabajar con tres operaciones básicas: **insertar**, **buscar**, y **eliminar**.

Ejemplos

- **Sistema de caché de un navegador:** A través de este caché, el navegador guarda las respuestas que recibió de determinados pedidos (por ejemplo, *HTTP requests*) a distintos servidores.

  De esta forma, puede reutilizar las respuestas guardadas en pedidos subsiguientes.

  En estos casos, podemos pensar que el caché del navegador guardará la URL de los recursos junto con su contenido (por ejemplo, una determinada imagen). Pasado cierto tiempo, el navegador borrará la entrada en el caché asociada a la URL.

Introducción

Además de las tres operaciones básicas ya mencionadas, estas aplicaciones tienen en común otra cuestión: **la necesidad de utilizar claves y valores**.

Es que justamente, en estos casos, la operación de búsqueda trae inherentemente estos conceptos: se busca algo por su clave (nombre de pila, URL), y se obtiene su valor asociado (el número de teléfono, la imagen).

Estas aplicaciones son ejemplos de lo que en ciencias de la computación se conoce como **diccionarios**.

Introducción

Un **diccionario** es un *conjunto dinámico* que admite las tres operaciones ya mencionadas: insertar (`insert`), buscar (`search`) y eliminar (`delete`), y en donde cada elemento del diccionario contiene una clave, la cual permite identificarlo.

Veremos, entonces, cómo podemos implementar diccionarios.

Tabla de Direccionamiento Directo

Veamos el siguiente ejemplo para entender un poco de qué se trata este tipo de implementación.

Ejemplo:

Supongamos que una empresa de turismo nacional decide hacer descuentos en ciertos destinos dependiendo del mes.

Por ejemplo, en enero los descuentos son en paquetes para viajar a Bariloche, en febrero en paquetes para viajar a Ushuaia, en marzo en paquetes para viajar a Córdoba, etc.

Quien esté a cargo del sistema informático de la empresa podría decidir guardar esta información en un diccionario, en donde las claves son número naturales representando el mes correspondiente y, los valores, la información asociada al descuento de dicho mes.

Tabla de Direccionamiento Directo

En general, cuando se trabaja con aplicaciones que necesitan un conjunto dinámico en el cual cada elemento posee una clave dentro de un universo

$$U = \{0, 1, \cdots, m-1\}$$

donde m no es muy grande y las claves son todas diferentes, para representar al diccionario podemos utilizar una *tabla de direccionamiento directo* (`DirectAccessTable`)

$$T[0, 1, \cdots, m-1]$$

en la cual cada posición, o *slot*, corresponde a una clave en el universo U.

Tabla de Direccionamiento Directo

Una implementación en Python de esta tabla utilizando listas podría ser la siguiente:

```
class DirectAccessTable():
    def __init__(self, capacity):
        self.m = capacity
        self.T = [None] * self.m

    def insert(self, key, element):
        self.T[key] = element

    def search(self, key):
        return self.T[key]

    def delete(self, key):
        self.T[key] = None
```

Tabla de Direccionamiento Directo

Volviendo sobre el ejemplo, podríamos representar el sistema de descuentos de la empresa de turismo mediante una tabla de direccionamiento directo de 12 elementos, como se muestra a continuación.

```
discounts = DirectAccessTable(12)
discounts.insert(0, "Bariloche")
discounts.insert(1, "Ushuaia")
discounts.insert(2, "Cordoba")
# insertar los descuentos para el
# resto de los meses
```

Tabla de Direccionamiento Directo

La siguiente figura ilustra el enfoque de las tablas de direccionamiento directo de modo general.

La posición k contiene un elemento del conjunto con clave k. Si el conjunto no contiene ningún elemento con clave k, entonces $T[k]$ está vacío, lo cual lo representamos indicando que contiene `None`.

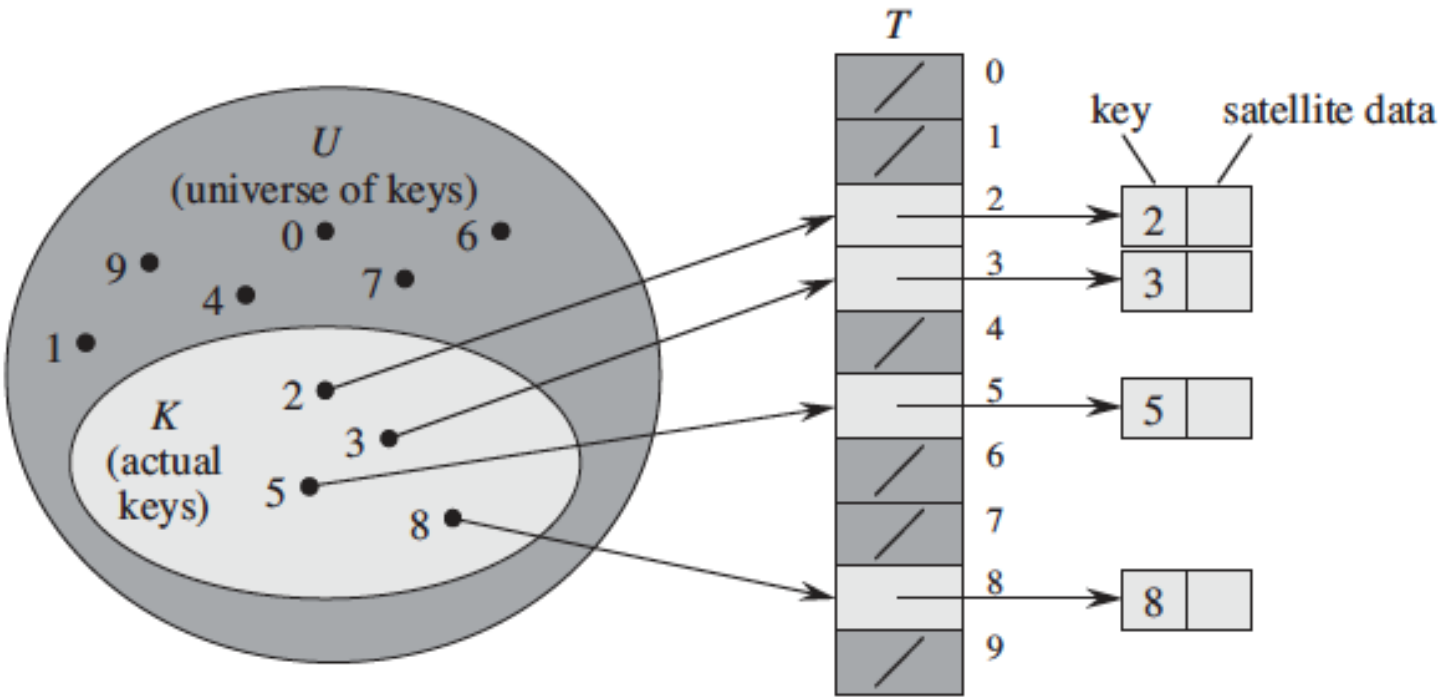


Tabla de Direccionamiento Directo

En este tipo de implementación, cada una de las operaciones de diccionario se ejecutan en tiempo constante.

En cuanto al espacio de memoria ocupado es del orden del tamaño del universo U.

Ahora bien, ¿qué pasaría si en lugar de trabajar con un sistema de descuentos por mes, nos fuéramos a otro campo y, por ejemplo, quisiéramos indexar los artículos de Wikipedia?

Si consideráramos el título de un artículo la clave mediante la cual se recupera el contenido del mismo, ¿cuál sería el universo de las claves? ¿Qué porcentaje de ese universo correspondería realmente a títulos válidos de artículos existentes?

¿Qué implementación usamos en este caso? Analicemos

Podemos notar que:

- cuando el tamaño del universo U de claves es grande
- cuando cantidad de elementos del diccionario es mucho más pequeño que el tamaño de U

No resulta conveniente implementar el diccionario mediante una tabla de direccionamiento directo.

Veremos a continuación otra implementación posible.

¿Qué implementación usamos en este caso? Analicemos

Volviendo al ejemplo de Wikipedia, podríamos querer, por ejemplo, indexar los artículos sobre computación (en español) mediante su título.

Si consideramos que un título está definido por valores alfanuméricos de hasta n caracteres de longitud, el conjunto K contendría los títulos de los artículos de computación existentes, mientras que el universo U de claves contendría todas las combinaciones posibles.

Suponiendo que existen cinco mil artículos de Wikipedia sobre computación que queremos indexar, sería razonable, por ejemplo, asignar claves únicamente en el rango $[0, 5000]$.

Para esto necesitaríamos una función capaz de transformar un valor cualquiera en un número natural en dicho rango. De esta forma, reduciríamos la cantidad de memoria necesaria a un orden similar a la cantidad de claves que realmente tendrá el diccionario. Esta es la idea detrás una **tabla hash**.

Tabla Hash (Hash Table)

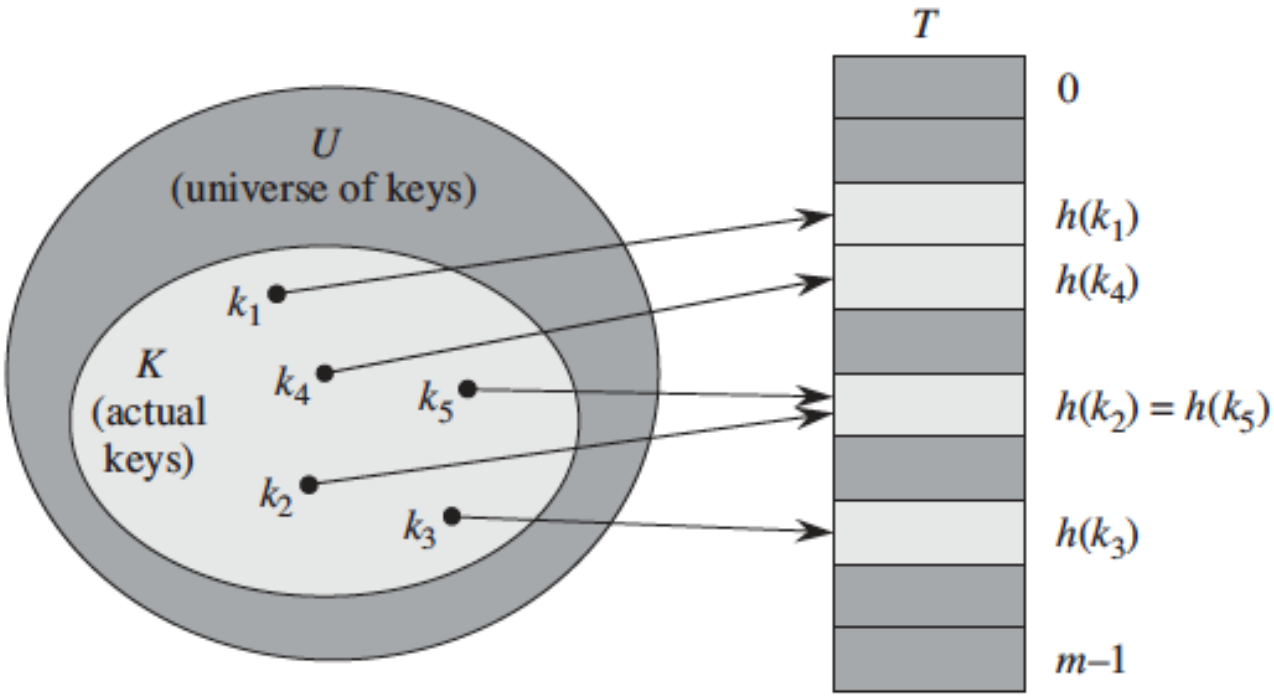
Para implementar una **tabla hash** debemos contar con dos puntos fundamentales:

- Por un lado, debemos definir una tabla de tamaño m, $T[0, 1, \cdots, m-1]$, donde m es la máxima cantidad de claves que soportará. Se espera que m sea menor que el tamaño del universo de claves U (o sea, $m < |U|$).
- Por otro lado, también es necesario definir una función $h : U \to \{0, \cdots, m-1\}$, la cual denominamos *función hash*. h calcula para cada clave k la posición o *slot* de la tabla que le corresponde.

En una tabla hash, el elemento con clave k se almacena en $T[h(k)]$.

Tabla Hash (Hash Table)

La siguiente figura ilustra este enfoque:



Podemos detectar un problema en la figura: dos claves, en este caso k_2 y k_5 mapean a la misma posición de la tabla, ya que $h(k_2) = h(k_5)$. A esta situación se la denomina **colisión**.

Veremos técnicas efectivas para resolver colisiones.

Tabla Hash (Hash Table)

A continuación se presenta una primera versión de la clase `HashTable` en Python.

```
class HashTable():
    def __init__(self, capacity, hashFunction):
        self.m = capacity
        self.h = hashFunction
        self.T = [None] * self.m

    def insert(self, key, element):
        pass

    def search(self, key):
        pass

    def delete(self, key):
```

Funciones Hash

Una buena función hash:

- Debe poder calcularse en tiempo constante
- Satisfacer la hipótesis de hashing uniforme: es equiprobable que una clave dada tenga cualquier valor hash entre 0 y $m - 1$

- Algunas funciones hash asumen que el universo de las claves es el conjunto $\mathbb{N}$ de números naturales
- Por lo tanto, si las claves no son números naturales debemos encontrar la forma de interpretarlas como si lo fueran
- Dada una cadena c, podemos transformar la cadena usando la siguiente función:

$$stringtonat(c) = \Sigma_{i=0}^{length(c)-1} c_i \cdot B^i$$

donde B es la base del conjunto de caracteres.

El método del resto propone la siguiente función hash:

$$h(k) = k \ mod \ m$$

- Debemos tener cuidado con el valor de m
- Funciona mal con valores $m = 2^p$, $h(k)$ estará dado por los primeros p bits de k, sin tener en cuenta los bits de orden superior.
- Funciona bien con m primos

Una función hash alternativa es la que propone el método de la multiplicación:

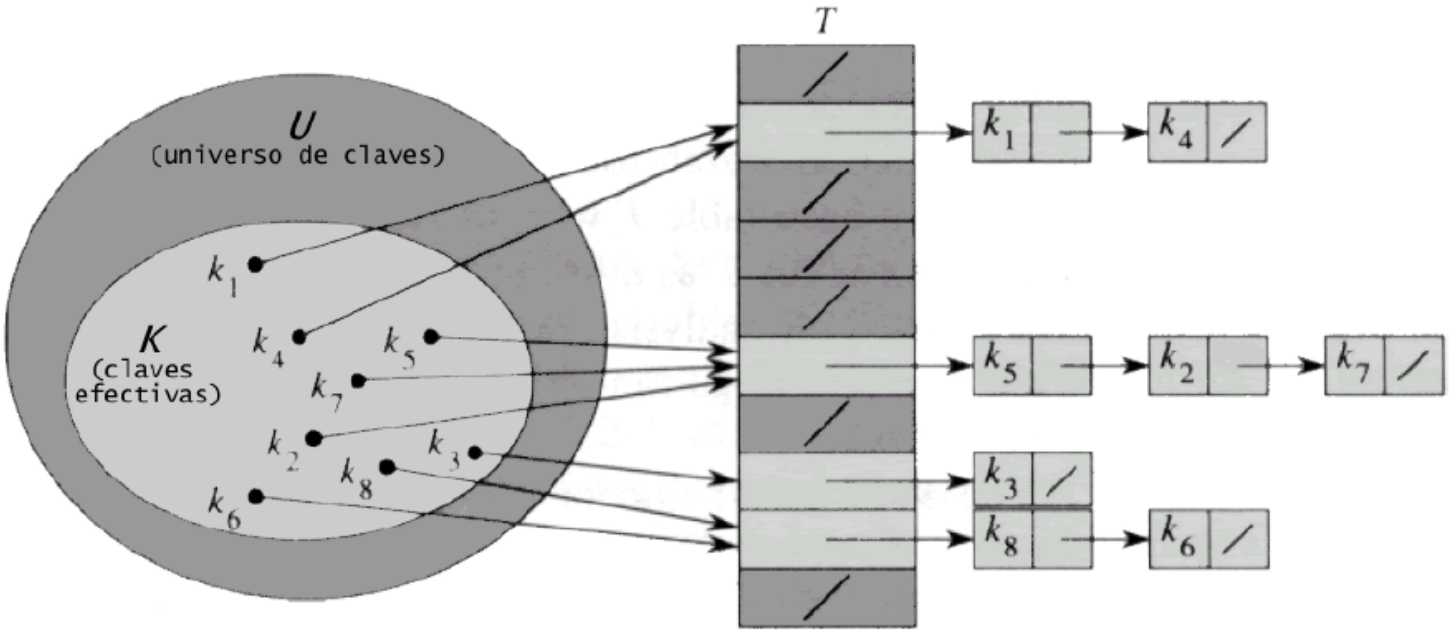
$$h(k) = \lfloor m \cdot (k \cdot A - \lfloor k \cdot A \rfloor) \rfloor$$

donde A es alguna constante en el rango $(0, 1)$.

- Existen otras funciones hash en la literatura
- Tarea para el alumno:

  Investigue sobre otras funciones hash que podamos usar

Cada slot $T[j]$ en la tabla hash contiene una estructura de datos auxiliar que guardara todas las claves cuyo valor hash es j. Una elección común es utilizar una lista enlazada, aunque puede ser cualquier otra estructura.



Por ejemplo, en la figura $h(k_2) = h(k_5) = h(k_7)$

Una inserción toma siempre tiempo constante.

La búsqueda y el borrado en el peor caso toman tiempo proporcional al tamaño de la lista.

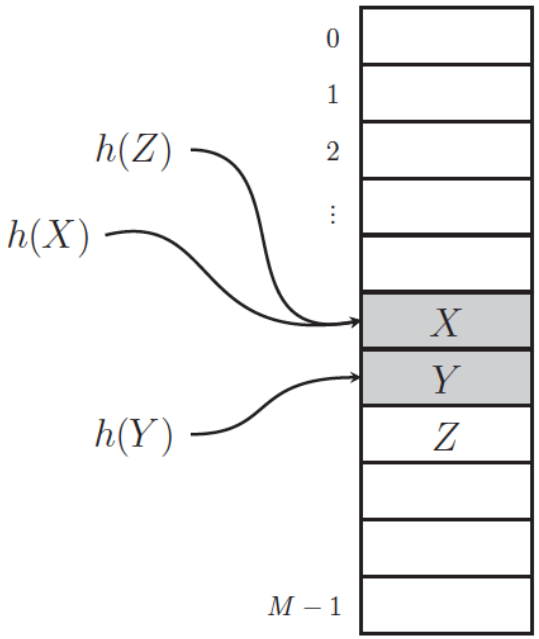
Podemos pensar los elementos de las listas enlazadas de tipo `Node2`

```
class Node2:
    def __init__(self, key=None, element=None,\
    next=None):
        self.key = key
        self.element = element
        self.next = next
```

Y pensar en implementar una **Tabla Hash encadenada** (HashTableChaining) en Python de la siguiente manera:

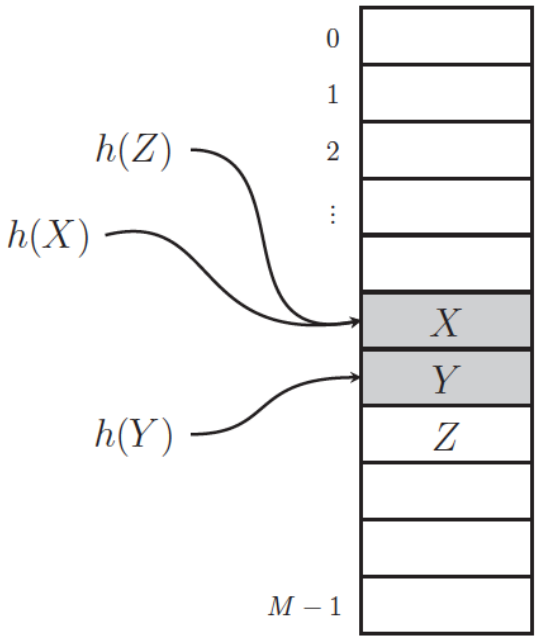
```
class HashTableChaining(HashTable):
    def insert(self, key, element):
        node = Node2(key, element)
        index = self.h(key)
        node.next = self.T[index]
        self.T[index] = node
    def search(self, key):
        # search for an element with key key
        # in list T[h(key)] - complete this code
    def delete(self, key):
        # delete element with key key
        # from list T[h(key)] - complete this code
```

Ciertas aplicaciones, para ahorrar prefieren almacenar todos los elementos directamente en la propia tabla. Esto se puede lograr mediante el método llamado **direccionamiento abierto**.



La función h devuelve un valor entre 0 y $m-1$ que se puede utilizar como un índice en la tabla hash T. Si $T(h(X))$ se encuentra vacío, el elemento de clave X se almacenará en T en el índice $h(X)$.

Ciertas aplicaciones, para ahorrar prefieren almacenar todos los elementos directamente en la propia tabla. Esto se puede lograr mediante el método llamado **direccionamiento abierto**.



En caso de que h asigne el mismo lugar a otro elemento con clave Z, $h(Z) = h(X)$, el direccionamiento abierto ubica al elemento con clave Z, por ej., en la posición libre más cercana.

Si en la búsqueda de una posición vacía se alcanza la parte inferior de la tabla, se considera a la tabla como cíclica y se continúa la búsqueda desde la posición 0.

Esto se implementa fácilmente calculando la siguiente posición de la tabla a la posición i de la siguiente manera:

$$(i + 1)\ mod\ m.$$

Cuando la tabla comience a llenarse, las celdas ocupadas tenderán a formar *clusters* (grupos).

Cuanto mayor sea un cluster, mayor será la probabilidad de un nuevo elemento que caiga dentro del mismo, por lo cual será mayor la probabilidad de que dicho cluster se vuelva aún más grande: esto contradice la hipótesis la distribución uniforme de las claves.

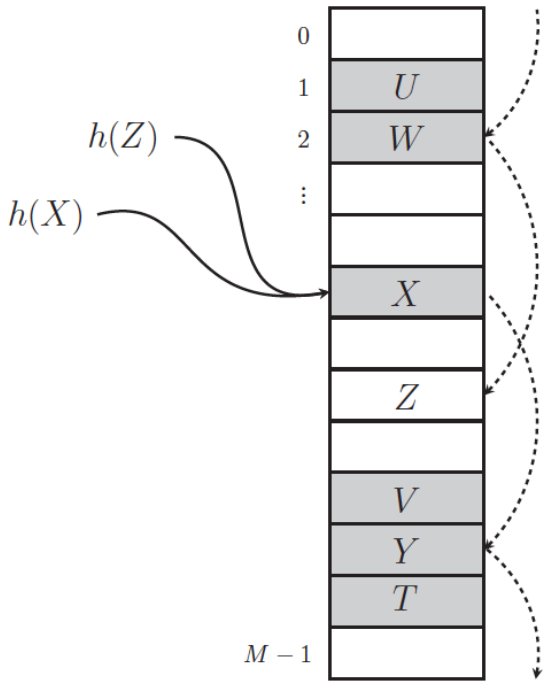
Una mejor idea parecería ser la de buscar una celda vacía en saltos de tamaño k, para algún número fijo $k > 1$.

Si k se elige de manera tal que el máximo común divisor entre k y m, $mcd(k, m)$, sea 1, entonces esta política de salto nos llevará de regreso al punto de partida sólo después de haber visitado todas las posiciones de la tabla.

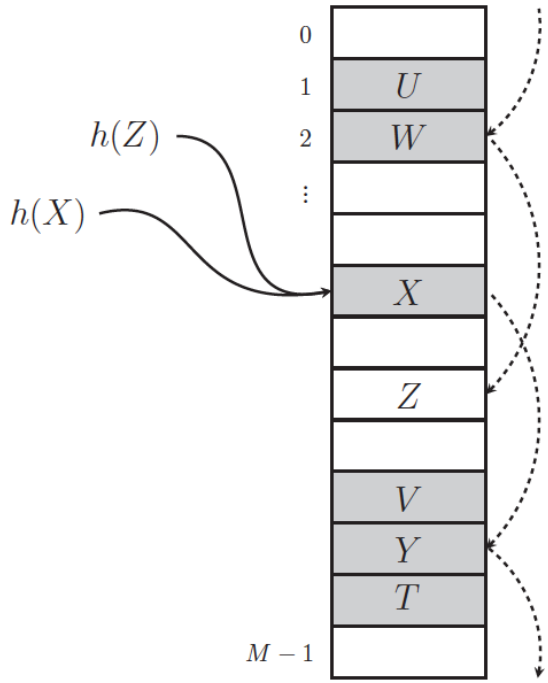
Este es un incentivo para elegir m como número primo, ya que cualquier tamaño de salto k logra el objetivo.

Colisiones

Resolución de Colisiones por Direccionamiento Abierto



En el ejemplo de la Figura los valores usados son $m = 13$ y $k = 5$. Como $h(Z) = 5$ y dicha posición está ocupada por otro elemento de clave X, el índice actual se incrementa repetidamente en k.



La secuencia de posiciones generada, llamada secuencia de sondeo, se representan a través de las flechas, y corresponden a las posiciones 5, 10, 15 $mod\ m = 2$, y 7. Este es la primera ubicación, por lo que el elemento con clave Z será almacenado allí.

Lo cierto es que este último cambio es sólo cosmético: seguirá habiendo clusters como antes, solo podrían ser más difíciles de detectar.

Para evitar estos clusters, se puede reemplazar el salto de tamaño fijo k utilizado en el caso de colisión, por otro salto cuyo tamaño dependerá también de la clave a insertar. Esto sugiere usar **dos funciones hash independientes** $h_1(X)$ y $h_2(X)$, por eso el **método se llama doble hash**.

En un primer intento se intentará almacenar el elemento con clave X en la dirección:

$$l \leftarrow h_1(X)$$

si dicha posición se encuentra ocupada, el tamaño de salto se define por:

$$k \leftarrow h_2(X)$$

de manera tal que $1 \leq k \leq m$.
La secuencia de sondeo para almacenar el elemento de clave X será:

$$(l + k) \ mod \ m, (l + 2k) \ mod \ m, (l + 3k) \ mod \ m, \ldots$$

hasta encontrar una posición vacía.

¿Cuál sería una buena opción para esta segunda función hash h_2?

Una sugerencia sería usar la siguiente función h_2:

$$h_2(X) = 1 + X \ mod \ m'$$

donde m' es el mayor primo menor que m.

Factor de Carga y Re-hashing

El **factor de carga** α es una estadística crítica de una tabla hash y se define de la siguiente manera:

$$\alpha = \frac{n}{m}$$

donde:

- n corresponde a la cantidad de elementos guardados en la tabla
- m corresponda al tamaño de la tabla.

La "performance" de la tabla hash se deteriora en relación con el factor de carga α. Por lo tanto, se realiza un **re-hash de una tabla hash** o se redimensiona si el factor de carga α se aproxima a 1. Una tabla también se redimensiona si el factor de carga cae por debajo de cierta cifra.

Las cifras aceptables del factor de carga α se encuentran en entre 0,6 y 0,75.

Investigue cómo se realiza el re-hashing de una tabla.

Conclusiones

- Hemos implementado diccionarios usando tablas de direccionamiento directo y tablas hash
- Para las tablas hash es importante conseguir funciones hash adecuadas para el espacio de claves con el que vamos a trabajar
- Debe elegirse un método para la resolución de colisiones
- Debe mantenerse un factor de carga aceptable para mantener la "performance" de una tabla hash

¿PREGUNTAS?

Referencias

Apunte de Cátedra

Elaborados por el staff docente.

Será subido al campus virtual de la materia.

A. Downey et al, 2002.

How to Think Like a Computer Scientist. Learning with Python.

Capítulos 12 a 16

R. Johnsonbaugh, 2005

Matemáticas Discretas, 6ta Edición, Pearson Educación, México.

Capítulo 9