Laboratorio de Estructuras de Datos Avanzadas
Practica #3, Opción A
Andrés Gómez de Silva Garza

Resumen:
En esta práctica se van a implementar tablas de hash en Python y en Java. Las implementaciones deben tomar en cuenta todos los principios del software de calidad: eficiencia, robustez, generalidad, modularidad, legibilidad, etc. Los aspectos relevantes acerca del funcionamiento de las tablas de hash se presentan abajo de manera abstracta (es decir, independiente del lenguaje de implementación). Uno de los puntos importantes a considerar cuando se utilizan las tablas de hash, mencionado en dicha discusión teórica, es el manejo de colisiones. Para esta práctica se tendrá que utilizar el método conocido como "encadenamiento separado", también explicado abajo, para el manejo de colisiones. Otro punto importante a considerar es la función de hash que se va a utilizar. Cuál es la más apropiada depende de las características de los datos que se van a almacenar en la tabla de hash. Después de tener implementadas las tablas de hash se debe diseñar y realizar una serie de experimentos que permita medir tiempos de ejecución (y cualquier otro parámetro relevante) para: (a) obtener un entendimiento detallado del funcionamiento de las tablas de hash bajo la estrategia de encadenamiento separado para el manejo de colisiones, y (b) comparar las implementaciones en Python y Java entre sí. Algunos parámetros que deben variar (deben estar contemplados dentro del diseño de la serie de experimentos) para obtener suficientes datos para poder llegar a conclusiones generales son: (a) la cantidad de datos con los que se alimente la tabla de hash, *n*, en relación con el tamaño de la misma, *m* (*n* mucho mayor a *m* vs. *n* aproximadamente igual a *m* vs. *n* mucho menor que *m*), (b) el tipo de operación a realizar sobre la tabla de hash (inserción de datos vs. búsqueda de datos vs. eliminación de datos), y (c) cualquier otro que se les ocurra que permita explorar algún aspecto de las tablas de hash y/o el método de encadenamiento separado para el manejo de colisiones. Dado que las tablas de hash están diseñadas para almacenar grandes cantidades de información tal que el acceso a dicha información sea eficiente, busquen en Internet para ver si encuentran alguna fuente de grandes cantidades de datos (acerca de cualquier tema que les interese: medicina, deportes, transportes, astronomía, finanzas, entretenimiento, etc.) accesibles de forma gratuita, y utilicen dichos datos para poblar sus tablas de hash y realizar los experimentos (tendrán que descargarlos en uno o varios archivos y encontrar la manera de leer la información de dichos archivos desde los archivos en el lenguaje de programación respectivo a la hora de crear las tablas de hash).

Contexto teórico:
Las tablas de hash están diseñadas para el almacenamiento de cantidades muy grandes de información "compleja" (con estructura/detalles internos). Suponemos que cada dato tiene un valor clave (valor llave) asociado y que el universo de valores clave válidos es U.

Las tablas de hash permiten resolver uno o varios de los siguientes posibles problemas que suelen ocurrir en este tipo de situaciones (grandes cantidades de información compleja):

- El tamaño de U es muy grande (es decir, hay una cantidad muy grande de valores clave posibles). Se desperdiciaría mucha memoria (por tener muchas casillas vacías) si se usa un simple arreglo para almacenar los datos en caso de que el tamaño de U sea muy grande

pero la cantidad de valores que realmente se va a tener que almacenar es menor al tamaño de U, una situación muy común. Ejemplo: en teoría cualquier combinación de 16 dígitos podría ser un número de tarjeta de crédito válido, pero en la práctica no se han asignado todas esas combinaciones (y algunas están reservadas para propósitos internos de cada banco).

- Valores clave que no son números enteros y por lo tanto no se pueden usar directamente como índices para acceder a las casillas de un arreglo. La forma de "arreglar" este problema es encontrar un número entero equivalente (representativo) para cada uno de los valores clave originales y luego usar dichos números enteros como índices del arreglo. Ejemplo: el RFC (Registro Federal de Contribuyentes) en México es una clave que contiene dígitos y letras combinados, por lo que no se puede usar directamente como índice en un arreglo.
- Valores clave que no son únicos (es decir, hay varios datos, o varios posibles datos, asociados con cada valor clave). Ejemplo: entre los datos almacenados por una aerolínea está la información acerca de los vuelos que ha operado, está operando y tiene pensado operar; generalmente una aerolínea no suele usar el mismo número de vuelo para varios vuelos el mismo día, pero sí se repiten de un día al siguiente, por lo que, si el número de vuelo se toma como valor clave, habría muchos datos (uno por cada día en el que se operó u operará un vuelo con ese número) asociados con el mismo valor clave.

Las tablas de hash son como arreglos (es decir, son estructuras de datos lineales de acceso directo), pero en lugar de que el valor clave sirva directamente como índice, se tiene que utilizar una función de hash (función de mapeo) *h* para calcular el índice basándose en el valor clave.

Si *h* es de complejidad algorítmica constante entonces las operaciones de búsqueda, inserción y eliminación también son O(1) en general (aunque pueden llegar a ser menos eficientes si *h* y/o la tabla están mal diseñadas).

Algunos parámetros descriptivos importantes:

- *n*: la cantidad de datos que se desea almacenar usando la tabla.
- *m*: la cantidad de casillas (es decir, el tamaño) de la tabla (generalmente mucho menor que el tamaño de U).
- $\alpha=n/m$: el factor de carga de la tabla (generalmente se escoge *m* para que α nunca sea mayor a 0.8, pero no si se utiliza encadenamiento separado para el manejo de colisiones).
- N: la cantidad total de posibles valores clave (igual al tamaño de U).

Dos tipos de función de hash que se utilizan mucho son los siguientes. Aquí vamos a suponer que *k* es un valor clave entero, pero si los valores clave dentro de algún conjunto de datos no son enteros, habrá primero que nada que encontrar un valor entero equivalente antes de poder utilizar las siguientes funciones de hash:

- El método de la división: Si *k* es el valor clave de algún dato, entonces $h(k)=k \bmod m$. Si k_1 y k_2 son valores clave adyacentes, entonces $h(k_1)$ y $h(k_2)$ probablemente también sean adyacentes (aunque no siempre lo serán). Empíricamente se ha observado que una buena elección para el valor de *m* en este caso (suponiendo que *m* no esté dado, sino que tenemos la posibilidad de elegirlo) es algún número primo que no esté cerca de alguna potencia de 2.

- El método de la multiplicación: Si *k* es el valor clave de algún dato y A es una constante tal que 0<A<1, entonces h(k)=piso(m*(k*A-piso(k*A))). Si k_1 y k_2 son valores clave adyacentes, entonces h(k_1) y h(k_2) probablemente no sean adyacentes en este caso. En este método la elección de *m* no es tan importante como en el anterior, pues no tiene un impacto tan grande sobre la eficiencia. Se ha observado empíricamente que una buena elección para el valor de A es A=0.61803399 (=1/relación áurea, proporción áurea).

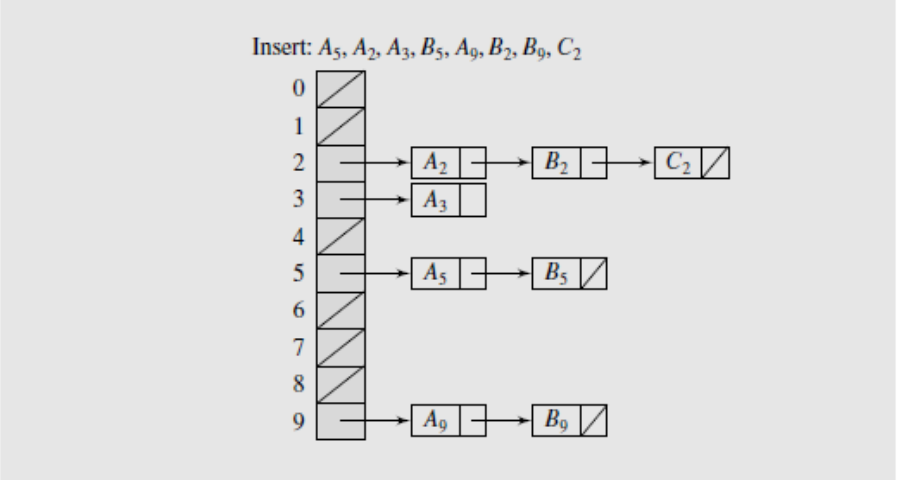
Propiedades deseables de las funciones de hash:

- Que la probabilidad de obtener cualquier posible valor de hash sea aproximadamente la misma que la de obtener cualquier otro valor de hash (es decir, la probabilidad debe ser aproximadamente 1/m). En inglés esto se describe como "simple uniform hashing".
- Que el tiempo requerido para calcular h(k) sea O(1).

Pueden ocurrir "colisiones" (cuando varios valores distintos de *k* pueden arrojar el mismo valor para h(k)). Se han propuesto varias estrategias para el manejo de colisiones ("estrategias de resolución de colisiones"). Aquí se presentará sólo una: encadenamiento separado. La figura ilustrativa de dicha estrategia de resolución de colisiones fue tomada del libro de Drozdek (ver bibliografía abajo) y supone que la operación que se está realizando es la inserción de datos en la tabla y que los subíndices de los datos a insertar indican el valor de hash de dichos datos (es decir, el índice en el que dichos datos deberían almacenarse, si es que se puede):

- Encadenamiento separado ("separate chaining" en inglés): En esta estrategia se hace que cada casilla *c* de la tabla de hash apunte a una lista dinámica de nodos enlazados. Los nodos de dicha lista se usan para guardar todos los datos que se querían almacenar "en la tabla" y que llevaron a la obtención del mismo valor de h(k), donde este valor se usa como índice de la casilla *c*. Si como ejemplo suponemos que *n* sea aproximadamente igual a 1000 y elegimos m=301, entonces en promedio tendremos $\alpha=n/m=3$ (es decir, tendremos aproximadamente 3 nodos en la lista enlazada asociada con cada casilla de la tabla de 301 casillas). Como el acceso a una casilla específica es igual de rápido que el acceso a cualquier otra casilla (una vez conocido su índice) y se logra en un solo paso computacional, después de lo cual el acceso al dato dentro de la lista enlazada correspondiente tardará cuando mucho otros tres pasos computacionales, entonces el tiempo de acceso total a un dato específico será O(1+3)=O(4)=O(1). El acceso es de complejidad algorítmica constante porque no depende de cuántos datos se almacenen en total en la tabla. Visualmente (página 528 de Drozdek):

FIGURE 10.5 In chaining, colliding keys are put on the same linked list.



Si se utiliza la estrategia de encadenamiento separado para la resolución de colisiones, el valor de α=n/m puede ser mayor a 1 y la cantidad de memoria que utiliza la tabla (junto con sus listas enlazadas asociadas) puede crecer y disminuir dinámicamente (gracias a que las listas enlazadas son estructuras de datos dinámicas). En otras estrategias no puede ocurrir que α=n/m sea mayor a 1, y hay que monitorear el valor de α conforme crece la tabla para tratar de evitar que rebase un valor de entre 0.7 y 0.8 (aproximadamente) para que no se degrade el desempeño de los algoritmos de acceso a los datos. En el momento en el que se rebase dicho umbral se podría decidir aumentarle el tamaño a la tabla (hacerle un "expand capacity"), pero esto implica hacer una "pausa" en el comportamiento normal de la tabla mientras se haga el ajuste (dado que las tablas, debido a que en el fondo son arreglos, no son dinámicas). Se han propuesto varias técnicas para hacer esto, pero van más allá del alcance del laboratorio.

¿Cómo comparan las implementaciones de las tablas de hash en Python y en Java en relación a: tiempos de desarrollo, cantidad de líneas de código, dificultad de diseño/implementación, tiempos de ejecución de las operaciones típicas y los demás parámetros que se les ocurra medir? ¿Cómo compara el desempeño de sus implementaciones de las tablas de hash en relación a lo que dice la teoría que debe ser dicho desempeño?

Bibliografía:

• Drozdek, Adam, *Data Structures and Algorithms in Java (2nd ed.)*, Thomson (Cengage) Learning, 2005.