95.12 Algoritmos y Programación II

Práctica 8: hashing

Notas preliminares

- Esta práctica presenta una estructura de datos útil para implementar, por ejemplo, conjuntos o diccionarios: la tabla de hash.
- Los ejercicios marcados con el símbolo ♣ constituyen un subconjunto mínimo de ejercitación. No obstante, recomendamos fuertemente realizar todos los ejercicios.

Ejercicio 1 ♣

Para cada implementación de tablas de hash listada a continuación, indicar cómo resulta la tabla de hash al insertar, en el orden dado, las claves

$$\langle \text{uno}, \text{dos}, \text{tres}, \text{cuatro}, \text{cinco}, \text{seis}, \text{siete}, \text{ocho}, \text{nueve}, \text{diez}, \text{once}, \text{doce} \rangle$$

Suponer que la tabla, inicialmente vacía, tiene tamaño $m = 16$. Utilizar los siguientes hashes para cada clave:

k	$h(k)$
uno	7470
dos	26627384
tres	1976494643
cuatro	1149459685
cinco	25857009
seis	469560
siete	30035060
ocho	27740724
nueve	29248806
diez	416312
once	29032165
doce	1706774245

(a) Direccionamiento cerrado (i.e., listas enlazadas en cada bucket).

(b) Direccionamiento abierto con sondeo lineal (*linear probing*).

(c) Direccionamiento abierto con sondeo cuadrático (*quadratic probing*).

(d) Direccionamiento abierto con hashing doble.Para este caso, utilizar la siguiente función de hash:

$$h_i(k) = (h(k) + i\,h'(k)) \bmod m$$

donde $h'(k) = 1 + (h(k) \bmod (m-1))$

Ejercicio 2

Para cada tabla del ejercicio anterior, encontrar una expresión matemática para representar la cantidad total de memoria utilizada por una tabla de tamaño m conteniendo n claves insertadas.

Ejercicio 3

Sea $h : U \to \{1, \ldots, m\}$ una función de hash y sea T una tabla de hash con direccionamiento cerrado de tamaño m. Probar que, si $|U| > nm$, existe un conjunto $K \subseteq U$ de tamaño n tal que $h(k_1) = h(k_2)$ para cualquier $k_1, k_2 \in K$, lo cual pone en evidencia el peor caso de $\Theta(n)$ al realizar una búsqueda en T.

Ejercicio 4

Los nuevos códigos postales argentinos tienen la forma *cddddccc*, donde *c* indica un caracter A,...,Z y *d* indica un dígito $0, \ldots, 9$. Por ejemplo, C1424CWN es el código postal que representa a la calle Saraza a la altura 1024 en la Ciudad Autónoma de Buenos Aires. Encontrar una función de hash apropiada para los códigos postales argentinos.

Ejercicio 5 ♣

(a) Considerar una tabla de hash con direccionamiento cerrado de tamaño m y conteniendo n claves. La performance de la tabla decrece a medida que el factor de carga $\alpha = n/m$ aumenta. Para mantener $\alpha < 1$, puede duplicarse el tamaño del arreglo de la tabla cuando $n = m$. Sin embargo, para lograr esto, será necesario rehashear todos los elementos. Explicar por qué es necesario el rehasheo.

(b) Mostrar que, utilizando la estrategia sugerida, el costo promedio para insertar un elemento en la tabla sigue siendo $O(1)$.

Ejercicio 6

(a) Dar una secuencia de m claves para llenar una tabla de hash con direccionamiento abierto y sondeo lineal en el menor tiempo posible. Encontrar una cota asintótica ajustada para el mínimo tiempo requerido para llenar la tabla.

(b) Dar una secuencia de m claves para llenar una tabla de hash con direccionamiento abierto y sondeo lineal en el mayor tiempo posible. Encontrar una cota asintótica ajustada para el mínimo tiempo requerido para llenar la tabla. Pensar cómo responder este mismo ítem si se utilizara sondeo cuadrático.

Ejercicio 7

Suponer que, en lugar de implementar direccionamiento cerrado utilizando listas enlazadas, se decide innovar y sustituir las listas por árboles binarios de búsqueda.

(a) ¿Cuál es, en tal caso, la complejidad temporal de peor caso para las operaciones de inserción, borrado y búsqueda?

(b) ¿Cuál es la complejidad temporal de caso promedio para dichas operaciones?

(c) Repetir las preguntas anteriores si se utilizan árboles AVL.

(d) Repetir las dos primeras preguntas si se utilizan listas ordenadas.

Ejercicio 8 ♣

Considerar los conjuntos de enteros $S = \{s_1, \ldots, s_n\}$ y $T = \{t_1, \ldots, t_m\}$.

(a) Proponer un algoritmo que utilice una tabla de hash para determinar si $S \subseteq T$. ¿Cuál es la complejidad temporal de caso promedio de este algoritmo?

(b) Mostrar que es posible determinar si $S = T$ en tiempo promedio $O(n + m)$.

Ejercicio 9 ♣

Diseñar un algoritmo para solucionar el problema del elemento mayoritario (recordar su enunciado recurriendo a la práctica 5) utilizando tablas de hash. Calcular la complejidad temporal de caso promedio y, a partir de esto, sacar conclusiones de cuán eficiente es respecto de los otros algoritmos ya estudiados.

Ejercicio 10 ♣

Considerar una tabla de hash T con direccionamiento cerrado en la que cada bucket se implementa con listas enlazadas y tal que el rehasheo de claves se da en una de cada dos inserciones. Si la función de hash utilizada por T distribuye uniformemente los valores, ¿es cierto que realizar n inserciones consecutivas en esta tabla toma tiempo $O(n)$?

Ejercicio 11

Sea T una tabla de hash con direccionamiento cerrado implementada de la siguiente manera:

- Su cantidad inicial de buckets es m.
- Utiliza una función de hash h computable en $\Theta(1)$ y que distribuye los valores uniformemente.
- Define un umbral u para el factor de carga λ tal que, cada vez que $\lambda > u$, el tamaño de T se multiplica por una constante fija k y la tabla se rehashea.
- El umbral está definido en el siguiente valor:

$$u = \frac{1}{m\,k^m}$$

Encontrar una expresión asintótica ajustada para representar el costo total de efectuar m inserciones consecutivas en T.

Ejercicio 12 ♣

(a) Proponer la interfaz de una clase C++ para representar tablas de hash con direccionamiento cerrado, detallando claramente las operaciones y métodos provistos. Contextualizar apropiadamente la clase mencionando estructuras de datos auxiliares utilizadas, interfaces requeridas, invariantes, etc.
Aclaración: no es necesario dar la implementación de ningún método.

(b) Supongamos que cierta aplicación que hace uso de la tabla de hash sugerida en el ítem anterior encuentra que, por motivos desconocidos, varias instancias registran elementos hasheados erróneamente (i.e., almacenados en posiciones de la tabla que no se corresponden con los valores calculados por la función de hash). Implementar en C++ una función `reparar_tabla` que, dada una tabla de hash T, reposicione en T todo elemento mal hasheado colocándolo en la ubicación que realmente le corresponde.

Ejercicio 13

En las implementaciones de tablas de hash tradicionales, el orden de inserción de los elementos suele perderse. Esto se pone de manifiesto, por ejemplo, al solicitar la lista de elementos insertados o bien al iterar sobre ellos. Para ilustrar este hecho, el código que sigue (escrito en el lenguaje de programación Python) obtiene las claves de un diccionario determinado:

```
>>> d = dict()
>>> d['e. honda'] = 11
>>> d['goro'] = 22
>>> d['king kong'] = 109
>>> d.keys()
['goro', 'king kong', 'e. honda']
```

Python provee, por otro lado, diccionarios *ordenados* (`OrderedDicts`) que, precisamente, mantienen el orden de inserción en la tabla:

```
>>> from collections import OrderedDict
>>> od = OrderedDict()
```

```
>>> od['e. honda'] = 11
>>> od['goro'] = 22
>>> od['king kong'] = 109
>>> od.keys()
['e. honda', 'goro', 'king kong']
```

Proponer una estructura apropiada para implementar tablas de hash ordenadas, sujeta además a las siguientes restricciones:

- La complejidad de las operaciones de inserción, borrado y búsqueda debe ser exactamente igual que en las tablas de hash tradicionales (i.e., $O(1)$ promedio)
- La complejidad del algoritmo para obtener los elementos insertados debe ser $\Theta(n)$, siendo n la cantidad de dichos elementos.

Minimizar tanto como sea posible la complejidad espacial (i.e., la memoria adicional a la tabla que la estructura utiliza).

Ejercicio 14 ♣

Las *cuckoo hash tables* son tablas de hash de direccionamiento abierto en donde el mecanismo de resolución de colisiones consiste en disponer de dos tablas y dos funciones de hash distintas que permitan encontrar posiciones alternativas para los elementos a almancenar. La idea, entonces, es trabajar con dos arreglos T_1 y T_2 de n posiciones cada uno, y dos funciones de hash h_1 y h_2, cada una vinculada a uno de los arreglos. Al insertar un nuevo elemento x, se siguen los siguientes pasos:

- Si $T_1[h_1(x)]$ está libre, poner a x ahí y terminar.
- De lo contrario, sea x' el elemento en $T_1[h_1(x)]$. Poner a x en lugar de x' en dicha posición.
- Ahora, resta recolocar a x' en la tabla. Esto lo logramos ubicándolo en $T_2[h_2(x')]$, y repitiendo el proceso para el eventual elemento x'' previamente alojado allí (i.e., recurrir a $T_1[h_1(x'')]$ y así sucesivamente).
- Puede ocurrir que este proceso no termine, como lo muestra la figura de más abajo (x desplaza a y en T_1, que a su vez desplaza a z en T_2, que a su vez desplaza a u en T_1, que a su vez desplaza a v en T_2, que su vez desplaza a y en T_1).
- En tal caso, se define un número máximo k de iteraciones. Cuando la cantidad de reubicaciones en la tabla supera este valor, se generan dos nuevas funciones de hash h_1' y h_2', se rehashean todos los elementos en la tabla con estas dos funciones y se vuelve a reintentar la inserción. Este concepto de rehasheo no necesariamente involucra redimensionar las tablas sino recalcular los hashes de cada elemento con las nuevas funciones.

(a) Proponer la interfaz de una clase C++ para representar cuckoo hash tables, detallando claramente las operaciones y métodos provistos. Contextualizar apropiadamente la clase mencionando estructuras de datos auxiliares utilizadas, interfaces requeridas, invariantes, etc.
Aclaración: no es necesario dar la implementación de ningún método.

(b) A partir de lo anterior, implementar en C++ el algoritmo de inserción delineado más arriba.

(c) Comparar el tiempo de búsqueda en peor caso de una cuckoo hash table frente al tiempo de búsqueda en peor caso de una tabla de hash tradicional. ¿Cuál resulta más eficiente?

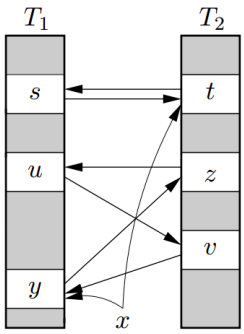

Figura 1: ciclo en una cuckoo hash table