I2 2018-1

2a) Hashing

Queremos multiplicar dos números X y Y, y para verificar si el resultado $Z = X \times Y$ es correcto, aplicamos una función de hash h a los tres números y vemos si

$$h(\, h(X) \times h(Y)\,) = h(Z).$$

Si los números difieren, entonces cometimos un error, pero si son iguales, entonces el resultado es (muy probablemente) correcto. h se define como calcular repetidamente la suma de los dígitos, hasta que quede un solo dígito, en cuyo caso 9 cuenta como 0.

P.ej., si $X = 123456$ y $Y = 98765432$, entonces $Z = 12193185172992$, y $h(X) = h(21) = 3$, $h(Y) = h(44) = 8$, $h(Z) = h(60) = 6$, y efectivamente, $h(3 \times 8) = h(24) = 6 = h(Z)$. Notemos que el dígito 9 no influye en el resultado calculado de h; p.ej., $h(49) = h(13) = 4$.

a) Da una expresión matemática para $h(X)$ [**1.5 pts.**]; y **b**) muestra que este método de verificación del resultado de la multiplicación es correcto [**1.5 pts.**].

Respuesta:

a) **X mod 9** (También se aceptan respuestas en donde el alumno calcula la suma de los dígitos con alguna expresión y después puso módulo 9, o expresiones recursivas, funciones compuesta y otras donde se mencione explícitamente que h(9) = 0 o se maneje bien ese caso)

b)

Sabemos que:

$$X = 9 \times Q_1 + R_1$$
$$Y = 9 \times Q_2 + R_2$$
$$X \times Y = Z$$

Y queremos demostrar que:

$$h(h(X) \times h(Y)) = h(Z)$$
$$h(h(9 \times Q_1 + R_1) \times h(9 \times Q_2 + R_2) = h(X \times Y)$$
$$h(h(9 \times Q_1 + R_1) \times h(9 \times Q_2 + R_2) = h((9 \times Q_1 + R_1) \times (9 \times Q_2 + R_2))$$

En la expresión de la izquierda podemos eliminar todos los términos múltiplos de 9 ya que estamos trabajando en mod 9, obteniendo

$$h(R_1 \times R_2) = h((9 \times Q_1 + R_1) \times (9 \times Q_2 + R_2))$$
$$h(R_1 \times R_2) = h(81 \times Q_1 \times Q_2 + 9 \times Q_1 \times R_2 + 9 \times Q_2 \times R_1 + R_1 \times R_2)$$

Nuevamente, eliminamos los términos múltiplos de 9 de la expresión de la derecha ya que estamos trabajando en mod 9, obteniendo

$$h(R_1 \times R_2) = h(9 \times (9 \times Q_1 \times Q_2 + Q_1 \times R_2 + Q_2) + R_1 \times R_2)$$
$$h(R_1 \times R_2) = h(R_1 \times R_2)$$

Notas: Se aceptaran demostraciones menos formales, incluso con palabras, siempre que mencionen que los cocientes se eliminan ya que se está trabajando en mod 9 y quedan los restos. No se aceptarán respuestas que solo mencionan la propiedad sin demostrarla. Si el alumno no uso **X mod 9** en la parte a, se deberá evaluar caso a caso.

2b) Tablas de *hash*

Para cada uno de los siguientes problemas, responde si es posible resolver el problema eficientemente mediante *hashing*. En caso afirmativo, explica claramente cómo; de lo contrario, sugiere otra forma de resolverlo según lo estudiado en el curso.

a) Considera un sistema de respaldo en que toda la información digital de una empresa tiene que ser respaldada, es decir, copiada, cada cierto tiempo. Una propiedad de estos sistemas es que solo una pequeña fracción de toda la información cambia entre un respaldo y el siguiente. Por lo tanto, en cada respaldo, solo es necesario copiar la información que efectivamente ha cambiado. El desafío es, por supuesto, encontrar lo más que se pueda de la información que no ha cambiado. [**1.5 pts.**]

Si es posible resolver mediante hashing. Un ejemplo de implementación eficiente mediante hashing es el uso de una función de hash que inicialmente se haya usado para respaldar toda la información. Estos elementos habrían llegado a algún espacio de la tabla. Al momento de querer respaldar nuevamente la información que cambió, uno puede tomar cada archivo de la información digital y utilizar el hash para identificar si este se modificó en la tabla (por ejemplo, hashear el nombre del archivo con su path y la fecha de modificación, o también hashear el archivo completo) y en caso de que coincidan, no se respalda. En caso de que sean diferentes, se puede generar el nuevo hash a partir de un hash incremental y los pocos cambios generados para luego insertar nuevamente en la tabla (liberando la anterior o complementando con el resto de la información nueva a respaldar).

b) Dada una lista L de números, queremos encontrar el elemento de L que sea el más cercano a un número dado x. [**0.5 pts.**]

No es resolvible por hashing eficientemente. Dado que es una lista sin un orden específico, se debe considerar alguna alternativa que entrega un orden para luego hacer por ejemplo una búsqueda binaria sobre un arreglo o una búsqueda sobre un árbol binario balanceado.

c) Queremos encontrar un string S de largo m en un texto T de largo n. [**1 pt.**]

Se vio en clases. Se puede resolver eficientemente mediante hashing. Usando una función de hash incremental, se toman los primeros m caracteres del texto de largo n para calcular el hash. De esta manera se compara con el hash del string S a buscar. En caso de que no sean iguales los hash, debes eliminar el primer elemento del string y agregar el siguiente del texto (O(1)) para luego volver a realizar el proceso. En caso de que existan colisiones, es mejor revisar caracter a caracter en caso de que sean iguales los hash.

I3 2017-1

2. Hashing universal es una técnica para generar buenas funciones de hash. Dado un universo de claves U, se definen la funciones $g_{a,b}(k) = (ak + b) \bmod p$ y $h_{a,b}(k) = g_{a,b}(k) \bmod m$, donde p es un primo tal que $p > k$, para cada $k \in U$, y $a \in \{1, \ldots, p-1\}$ y $b \in \{0, \ldots, p-1\}$.

   a) Una de las razones porque $h_{a,b}$ es "buena" es porque "evita colisiones antes de módulo m". Esto significa que si $k \neq k'$, entonces $g_{a,b}(k) \neq g_{a,b}(k')$. Demuestre este resultado.

   Respuesta (2 puntos): Esto se puede demostrar por contradicción, asumiremos lo contrario, es decir $k \neq k'$ y $g_{a,b}(k) = g_{a,b}(k')$. Desarrollamos la igualdad:

$$\begin{aligned} g_{a,b}(k) &= g_{a,b}(k') \\ g_{a,b}(k) - g_{a,b}(k') &= 0 \\ (ak + b) \bmod p - (ak' + b) \bmod p &= 0 \end{aligned}$$

   Utilizando las propiedades del módulo y que la resta siempre estará en el rango $\{0, p-1\}$ llegamos a que:

$$\begin{aligned} ((ak + b) - (ak' + b)) \bmod p &= 0 \\ (a(k - k')) \bmod p &= 0 \end{aligned}$$

   Luego, para que se cumpla la igualdad se debe cumplir alguna de las siguientes condiciones:

   - $a \bmod p = 0$, lo cual no es cierto debido a que $a \in \{1, \ldots, p-1\}$ y $b \in \{0, \ldots, p-1\}$.
   - $(k - k') \bmod p = 0$, lo cual no es cierto ya que $p > k$ y por lo tanto la diferencia nunca estará fuera del rango $\{0, p-1\}$. Además como $k \neq k'$ la resta nunca será 0.
   - $a(k - k') = cp$ con c entero. Tampoco es cierto, ya que si lo fuera a o $(k - k')$ serían divisores de p, lo cual sumando el hecho de que $a < p$, y $(k - k') < p$ contradice el hecho de que p es primo.

   Por lo tanto llegamos a una contradicción, ya que es imposible que se cumpla la igualdad.

   Asignación de puntaje:

   - **1 punto** por demostrar que la expresión no es igual a 0 antes del módulo.
   - **1 punto** por demostrar que la expresión no es igual a un múltiplo de p.

b) El teorema de *a*) se puede demostrar sin obligar a que p sea primo, pero imponiendo otras restricciones sobre $g_{a,b}(k)$. Diga cuáles y demuestre su respuesta.

Respuesta (2 puntos): Se mantienen la condiciones anteriores pero agregando la condición de que a sea primo relativo a p, es decir que a no tenga divisores en común con p. La demostración es equivalente a la de *a)*, solo que ahora $a(k-k') = cp$ no puede ser cierto ya que significaría que un factor primo de a está en p.

Asignación de puntaje:

- **2 puntos** por respuesta correcta junto a su demostración.
- **1.5 puntos** por respuesta correcta con errores en la justificación.
- **1 punto** por solo decir que a no sea un divisor de p (Esto no es suficiente, podría pasar que a sea un factor de cp)
- **0.5 puntos** por dar condiciones demasiado restrictivas, por ejemplo restringir que $|a(k-k')|$ sea menor a p.

c) Muestre que si relajamos la restricción "$p > k$, para cada $k \in U$" es posible que $g_{a,b}(k) = g_{a,b}(k')$ incluso cuando $k \neq k'$.

Respuesta (2 puntos):
Ahora se puede dar que $k - k' = cp$ y por lo tanto es posible que $(a(k-k')) \bmod p = 0$. También se puede demostrar con un contraejemplo, un caso sería $a = 1, b = 0, p = 3, k = 1, k' = 4$.

$$g_{a,b}(k) = (1 \cdot 1 + 0) \bmod 3 = 1$$
$$g_{a,b}(k') = (1 \cdot 4 + 0) \bmod 3 = 1$$

La asignación de puntos es binaria, se dan 0 o 2 puntos.

EXAMEN 2017-1

1. Para cada una de las siguientes afirmaciones, diga si es *verdadera* o *falsa*, **siempre justificando** su respuesta.

e) Si se tienen dos tablas de hash, una con direccionamiento abierto y otra cerrado, las dos del mismo tamaño m y el mismo factor de carga α, ambas ocupan la misma cantidad de memoria.

j) Como diccionario, una tabla de hash es más conveniente que un árbol rojo negro en cualquier aplicación. (Sin considerar la dificultad de implementación).

l) Re-hashing, el procedimiento que construye una nueva tabla de hash a partir de otra existente, toma tiempo $O(n)$ en una tabla con colisiones resueltas por encadenamiento de tamaño m que contiene n datos. **Respuesta:** Falso. El tiempo depende del tamaño de la tabla y del número de datos; específicamente es $O(m+n)$.

I2 2016-2

3. Tienes el texto completo de la novela *A Tale of Two Cities*, de Charles Dickens.

> *It was the best of times, it was the worst of times, it was the age of wisdom, it was the age of foolishness …*

Un algoritmo de reconocimiento de texto necesita obtener la lista de palabras distintas en la novela. Escribe un algoritmo que haga esta operación en tiempo O(*n*) promedio, en que *n* es el número total de palabras del texto; y justifica esta complejidad.

Respuesta:

Una solución es hacer una tabla de hash con encadenamiento sobre los posibles strings y en base a eso detectar las palabras iguales:

$TablaHash \leftarrow$ Arreglo tamaño m de listas vacías.
$Resp \leftarrow$ Lista vacía de respuesta.
for $pal \in Palabras$ **do**
  $key \leftarrow hash(pal)\%m$
  if $TablaHash[key] == \emptyset \vee pal \notin TablaHash[key]$ **then**
    $TablaHash[key].add(pal)$
    $Resp.add(pal)$
  end if
end for
return $Resp$

Donde la función de hash es tal que no produce colisiones: se puede transformar el String a un entero fácilmente usando potencias de 128 en ASCII.

Cabe destacar que la tabla si tendrá colisiones, ya que a la función de hash luego se le aplica el módulo del tamaño del arreglo "m".

Finalmente como nuestra función de hash distribuye bien y suponiendo que satisface HUS, el riesgo de colisiones distribuye uniformemente en el caso promedio y por lo tanto la búsqueda en promedio será O(1) si tomamos un "m" lo suficientemente grande. (Se pueden hacer otras funciones o suponer que existe una, siempre y cuando se diga que satisface HUS).

El puntaje se entrega de la siguiente forma:

- 1 pto si es que se hizo un algoritmo correcto pero no de la complejidad pedida
- 3 ptos si es que se hizo una tabla de hash pero sin colisiones *
- 6 ptos por una respuesta correcta con justificación de complejidad, también existen otras soluciones como usar direccionamiento abierto.

*** Es imposible hacer una tabla de hash sin colisiones ya que las palabras posibles son muchas (en teoría infinitas), notamos que además la cantidad de strings posibles crece exponencialmente según el largo, por lo que por ejemplo si hiciéramos una tabla sin colisiones para strings de largo 100 se necesitaría un arreglo de tamaño** $128^{100} = 5.26 * 10^{210}$ **lo cual es gigantesco y no es posible para ningún computador (Un computador normal soporta del orden de** 10^9 **de memoria).**

I1 2015-2

3. En el caso de hashing con direccionamiento abierto, considera que tienes una tabla de tamaño $T = 10$ y las claves A_5, A_2, A_3, B_5, A_9, B_2, B_9, C_2, en que K_i significa que al aplicar la función de hash h a la clave, obtenemos la posición i. Muestra qué ocurre al insertar las claves anteriores, en el orden en que aparecen, en los siguientes casos:

a) Usamos revisión lineal, de modo que la posición intentada es $(h(K) + i)$ modulo T.

b) Usamos revisión cuadrática, de modo que la posición intentada es $h(K)$, $h(K)+1$, $h(K)–1$, $h(K)+4$, $h(K)–4$, $h(K)+9$, ..., $h(K)+(T–1)^2/4$, $h(K)–(T–1)^2/4$, todos divididos módulo T.

	a)	**b)**
0	B_9	B_9
1		B_2
2	A_2	A_2
3	A_3	A_3
4	B_2	
5	A_5	A_5
6	B_5	B_5
7	C_2	
8		C_2
9	A_9	A_9

I1 2015-1

1. *Estructuras de datos básicas.*

a) Describe una **estructura de datos** que funcione en memoria principal para majenar una pequeña biblioteca en nuestro departamento. La biblioteca tiene libros y tiene lectores. Los lectores son los estudiantes, profesores y funcionarios del departamento, todos con nombre y RUT. Los libros tienen título y tienen autor (el nombre de una persona). Tanto los lectores como los libros permanecen fijos durante el semestre (marzo a julio, agosto a diciembre). Al bibliotecario le interesa poder prestar libros a los lectores, recibir de vuelta libros que estaban prestados, y conocer la información habitual eficientemente: ¿Qué lectores hay registrados en la biblioteca? ¿Qué libros tiene la biblioteca? ¿Cuáles de ellos están disponibles y cuáles están prestados? ¿Cuál lector tiene un determinado libro prestado? ¿Cuáles libros están prestados a un determinado lector? Tu estructura puede incluir **arreglos**, **listas ligadas** en varias direcciones, y **tablas de hash**. **Justifica**.

Podemos manejar tanto los libros como los lectores en arreglos, en que cada casillero tiene varios campos; puede haber un tercer arreglo para los autores de los libros. Todos estos arreglos son en realidad tablas de hash; las claves son los RUT's de los lectores, los títulos de los libros, y los nombres de los autores. Los casilleros del arreglo de los autores son punteros a una lista ligada de punteros a cada uno de los libros (casilleros del arreglo de libros) del autor correspondiente. Para saber qué lectores y qué libros hay, simplemente recorremos los arreglos corresppndientes. Además, hay que representar los préstamos.

(Podríamos usar una tabla de doble entrada, es decir, una matriz, en que, p.ej., las columnas son los libros y las filas los lectores. Cada casilla de la matriz correspondería a un boolean, para indicar si el libro está prestado —1— o no —0— a un determinado lector. Si hay 500 lectores y 1,000 libros, la martiz tendría 500,000 casillas; pero a lo más 1,000 de estas casillas podrían tener un 1, cuando todos los libros están prestados.)

Siguiendo la sugerencia del enunciado, representamos los préstamos por listas ligadas a partir de los lectores. Cada elemento de una lista ligada es un puntero a un libro: el libro que está prestado al lector correspondiente. Como un lector puede tener varios libros prestados, ponemos todos esos elementos en una lista doblemente ligada. Además, para saber a qué lector está prestado un determinado libro, cada elemento de las listas ligadas es apuntado "de vuelta" por el libro correspondiente.

I1 2014-1

2. *Hashing*.

a) En el caso de *hashing* con direccionamiento abierto, si la tabla empieza a llenarse, entonces la ejecución de las operaciones de búsqueda e inserción empieza a tomar demasiado tiempo. La solución es construir otra tabla que sea el doble de grande, definir una nueva función de *hash*, y revisar la tabla original completa, calculando el nuevo valor de *hash* para cada elemento e insertándolo en la nueva tabla.

Esta operación, llamada *rehashing*, es cara, pero en la práctica, considerando la frecuencia con que debe ejecutarse, no afecta demasiado al desempeño global de la tabla. ¿Por qué?

Supongamos que decidimos hacer *rehashing* cada vez que la tabla está 50% llena (número de claves $n = m/2$, en que m es el tamaño de la tabla); es decir, tienen que haber ocurrido por lo menos $n = m/2$ operaciones de inserción. ¿Cuánto cuesta hacer *rehashing*? Reservar una nueva tabla de tamaño $2m$ y definir una nueva función de *hash* para esta tabla es O(1); suponiendo que calcular el valor de *hash* de una clave es O(1), entonces recorrer la tabla original y calcular el nuevo valor de *hash* para cada clave almacenada es O(m). Es decir, ejecutamos una operación de costo O(m) después de haber hecho $m/2$ = O(m) operaciones de costo O(1) en promedio cada una. Por lo tanto, en términos de O( ), el costo de *rehashing* no suma al costo ya incurrido de las inserciones en la tabla original.

b) Queremos encontrar la primera ocurrencia de un string $P_1P_2\ldots P_k$ en otro string mucho más largo $A_1A_2\ldots A_n$ ($n > k$). ¿Cómo podemos resolver este problema usando *hashing*? Suponiendo que para un string s empleamos una función de *hash* como la siguiente, ¿qué tan eficiente, en términos de k y n, es esta solución?

```
hashValue = 0
for (i = 0; i < s.length(); i++)
   hashValue = 37*hashValue + s.ascii(i)
hashValue = hashValue % tableSize
```

La idea es calcular el valor de *hash* para el string $P_1P_2\ldots P_k$ y luego calcular el valor de *hash* para cada substring de largo k del string $A_1A_2\ldots A_n$. Si los valores de *hash* son distintos, entonces los strings no pueden ser iguales. Si los valores de *hash* son iguales, entonces comparamos los strings carácter por carácter (ya que hay una pequeña posibilidad de que los strings sean distintos).

En general, este método va a tomar un tiempo proporcional al tiempo que toma calcular el valor de *hash* para un string de k caracteres, digamos $f(k)$, multiplicado por el número de tales strings, $n - k$, y más lo que toma comparar carácter por carácter dos strings de k caracteres; es decir, $(n - k)f(k) + k$. En el caso de la función de *hash* dada, $f(k) = ck$; luego, el método toma tiempo proporcional a $ck(n - k) + k$; esencialmente, O(nk).

Sin embargo, observamos que el valor de *hash* del string $A_iA_{i+1}\ldots A_{i+k-1}$, digamos h_i, puede obtenerse directamente a partir del valor de *hash* del string $A_{i-1}A_i\ldots A_{i+k}$, digamos h_{i-1}, sin tener que calcularlo desde cero: $h_i = h_{i-1} - \text{ascii}(A_{i-1})*37^{k-1} + \text{ascii}(A_{i+k})$. Es decir, sólo el cálculo del valor de *hash* del primer substring $A_1A_2\ldots A_k$ toma tiempo proporcional a k; los $n-k-1$ siguientes son todos constantes. Por lo tanto, el método toma tiempo proporcional a $k + (n-k-1) + k$; esencialmente, O($n+k$).

I1 2013-1

2. En el caso de hashing con encadenamiento, propón una forma de almacenar los elementos dentro de la misma tabla, manteniendo todos los casilleros no usados en una *lista ligada de casilleros disponibles*. Para esto, considera que cada casillero puede almacenar un *boolean* y, ya sea, un elemento más un puntero o dos punteros. Todas las operaciones de diccionario y las que manejan la lista debieran correr en tiempo $O(1)$ en promedio. Específicamente, explica lo siguiente:

a) **[0,5 puntos]** El papel del *boolean*.

El boolean es para saber si la casilla tiene un elemento y un puntero a otro elemento (o *null*), o si tiene dos punteros (a las casillas delante y detrás en la lista **doblemente ligada** de casillas disponibles).

b) **[4 puntos]** ¿Cómo se implementan las operaciones de diccionario: inserción, eliminación y búsqueda?

Llamemos **lista de colisiones** a la lista ligada que se forma al colisionar elementos (similarmente al hashing con encadenamiento) y **lista disponible** a la lista de casillas disponibles.

[2 puntos] Inserción: Se aplica la función de hash al nuevo elemento x; supongamos que da k. Si la casilla k está disponible (su boolean vale *true*), la sacamos de la lista (en tiempo O(1) porque está doblemente ligada) y ponemos ahí x: cambiamos el boolean a *false* y ponemos el puntero en *null*.

Si la casilla k está ocupada (su boolean vale *false*) por un elemento z, hay dos posibilidades: z hace hash a k; o z hace hash a un valor distinto de k (es decir, es parte de otra lista de colisiones). En el primer caso, hay que agregar x en el segundo lugar de la misma lista de z, usando una casilla de la lista disponible. En el segundo caso, hay que mover z a una casilla disponible y poner x en la casilla dejada por z, actualizando apropiadamente los punteros involucrados.

[1,25] Eliminación: Sea k la casilla a la que hace hash el elemento x a eliminar. Si x es el único elemento en la lista de colisiones que empieza en la casilla k, hay que agregar esta casilla a la lista disponible. Si x es el primer elemento, pero no único, en su lista de colisiones, hay que mover el segundo elemento z a la casilla k y agregar la casilla en que estaba z a la lista disponible. Si x no es el primer elemento en su lista de colisiones, hay que agregar la casilla que ocupa x a la lista disponible, actualizando apropiadamente los punteros.

[0,75 puntos] Búsqueda: Hay que revisar la casilla a la cual el elemento x hace hash. Si es una casilla disponible, entonces x no está en la tabla. De lo contrario, si x no está en la casilla, hay que seguir los punteros.

c) **[1,5 puntos]** ¿Por qué las operaciones de diccionario y las que manejan la lista de casilleros disponibles corren en tiempo $O(1)$ en promedio?

Las operaciones de diccionario operan igual que en el caso de hashing con encadenamiento y, como vimos en clase, esas corren en tiempo O(1) en promedio.

Las operaciones sobre la lista de casillas disponibles corren en tiempo O(1) gracias a que la lista es doblemente ligada (si no, el único problema ocurre cuando se saca una casilla de la lista).

I1 2011-2

2. En el caso de hashing con direccionamiento abierto, supón que tienes una tabla de tamaño $T = 10$ y las claves A_5, A_2, A_3, B_5, A_9, B_2, B_9, C_2, en que K_i significa que al aplicar la función de hash h a la clave, obtenemos la posición i. Muestra qué ocurre al insertar las claves anteriores, en el orden en que aparecen, en los siguientes casos:

a) [**3 pts.**] Si usamos revisión lineal, de modo que la posición intentada es $(h(K) + i) \bmod T$.

b) [**3 pts.**] Si usamos revisión cuadrática, de modo que la posición intentada es $h(K)$, $h(K)+1$, $h(K)–1$, $h(K)+4$, $h(K)–4$, $h(K)+9$, ..., $h(K)+(T–1)^2/4$, $h(K)–(T–1)^2/4$.

	a)	**b)**
0	B_9	B_9
1		B_2
2	A_2	A_2
3	A_3	A_3
4	B_2	
5	A_5	A_5
6	B_5	B_5
7	C_2	
8		C_2
9	A_9	A_9

I1 2010-1

1. En el caso de hashing con encadenamiento, propón una forma de almacenar los elementos dentro de la misma tabla, manteniendo todos los casilleros no usados en una lista ligada de casilleros disponibles. Pare esto, supón que cada casillero puede almacenar un boolean y ya sea un elemento más un puntero o dos punteros. Todas las operaciones de diccionario y las que manejan la lista deberían correr en tiempo esperado $O(1)$. Específicamente, explica lo siguiente:

a) [**1** pt] El papel del boolean.

El boolean es para saber si el casillero tiene un elemento (y un puntero a otro elemento o null), o si tiene dos punteros (a los casilleros delante y detrás en la lista ***doblemente ligada*** *de casilleros disponibles).*

b) [**3** pts.] ¿Cómo se implementan las operaciones de diccionario: inserción, eliminación y búsqueda?

Llamemos ***lista de colisiones*** *a la lista ligada que se forma al colisionar elementos (similarmente al hashing con encadenamiento).*

Inserción*: Se aplica la función de hash al elemento; supongamos que da* k. *Si el casillero* k *está disponible (su boolean vale* true*), lo sacamos de la lista (en tiempo O(1) porque está doblemente ligada) y almacenamos ahí el elemento. Para esto, cambiamos el boolean a* false *y ponemos el puntero en* nil.

Si el casillero k *está ocupado (su boolean vale* false*), hay dos posibilidades: es el primer elemento de la lista de colisiones que comienza en el casillero* k*; o es algún otro elemento en alguna lista de colisiones que comienza en otro casillero.*

Eliminación:

Búsqueda:

c) [**2** pts.] ¿Por qué las operaciones de diccionario y las que manejan la lista de casilleros disponibles corren en tiempo esperado $O(1)$?

Las operaciones de diccionario operan igual que en el caso de hashing con encadenamiento y, como vimos en clase, esas corren en tiempo O(1) esperado.

Las operaciones sobre la lista de casilleros disponibles corren en tiempo O(1) gracias a que es doblemente ligada (si no, el único problema ocurre cuando se saca un casillero específico de la lista de casilleros disponibles).