**3.3 – Diccionarios**

El tipo de dato *diccionario* permite el acceso a los elementos de datos a través del contenido.

Las operaciones principales de los diccionarios son:

* Search(D, k) – Dada una clave de búsqueda k, devuelve un puntero al elemento en el diccionario D cuyo valor de clave es k, si existe.
* Insert(D, x) – Dado un elemento de datos x, lo agrega al conjunto en el diccionario D.
* Delete(D, x) – Dado un puntero a un elemento de datos dado x en el diccionario D, lo elimina de D.
* Max(D) o Min(D) – Devuelve el elemento con la mayor (o menor) clave de D. Esto permite que el diccionario funcione como una cola de prioridad.
* Predecessor(D, k) o Successor(D, k) – Devuelve el elemento de D cuya clave es inmediatamente anterior (o posterior) de k de manera ordenada. Estos nos permiten iterar a través de los elementos de la estructura.

Muchas de las tareas de procesamiento más comunes pueden ser abordadas con diccionarios. Por ejemplo, suponga que queremos eliminar todos los nombres duplicados de una lista de correo, e imprimir los resultados en orden. Inicialice un diccionario vacío D, cuya clave de búsqueda será el nombre del registro. Ahora recorra la lista del correo, y por cada elemento busque si esta en D. Si no lo está, insértelo en D. Una vez terminado, debemos extraer los nombres restantes del diccionario. Comenzando desde el primer elemento (Min(D)) y llamando repetidamente a ***Successor*** hasta obtener el máximo de D (Max(D)), con esto, recorrimos todos los elementos de una manera ordenada.

Problema: ¿Cuáles son los tiempos de ejecución asintóticos en el peor de los casos para cada una de las siete operaciones fundamentales del diccionario (buscar, insertar, eliminar, sucesor, predecesor, mínimo y máximo) cuando la estructura de datos se implementa como:

1. Un vector desordenado,
2. Un vector ordenado.?

Solución:

Tabla

Descripción generada automáticamente

* Buscar fue implementado probando la clave de búsqueda k con cada elemento de una matriz desordenada. Por lo tanto, la búsqueda toma un tiempo lineal en el peor de los casos, que es cuando la clave k no se encuentra en la matriz.
* Insertar fue implementado incrementando una variable n y luego copiando en el elemento x en la n-ésima celda de la matriz, A[n]. La mayor parte de la matriz esta intacta, por lo que esta operación lleva un tiempo constante.
* La eliminación es algo más complicada, por eso el \* en la tabla. La definición establece que se nos da un puntero x al elemento a eliminar, por lo que no necesitamos perder tiempo buscando el elemento. Pero quitando el elemento x de la matriz A deja un hueco que debe ser llenado. Podríamos llenar el agujero moviendo cada uno de los elementos A[x+1] a A[n] una posición hacia arriba, pero esto requiere O(n) tiempo cuando se elimina el primer elemento. La siguiente idea es mejor: solo escribe sobre A[x] con A[n], y decrementa n. Esto solo toma tiempo constante.
* La definición de las operaciones transversales, Predecesor y Sucesor, se refieren al elemento que aparece antes/después de x en orden. Así, la respuesta no es simplemente A[x-1] o A[x+1], porque en una matriz no ordenada el elemento predecesor (o sucesor) físico no es necesariamente su predecesor (o sucesor) lógico. En cambio, el predecesor de A[x] es el mayor elemento menor que A[x]. Similarmente, el sucesor de A[x] es el menor elemento mayor que A[x]. Ambos requieren un barrido a través de los n elementos de A para determinar el ganador.
* Para las operaciones Min y Max, se definen de manera similar con respecto al vector ordenado. Para un vector desordenado, se requiere un barrido lineal para obtener el mayor o el menor.

Problema: ¿Cuáles son los tiempos de ejecución asintóticos en el peor de los casos para cada una de las siete operaciones fundamentales del diccionario cuando la estructura de datos se implementa como:

1. Una lista no ordenada de enlace único (singly-linked unsorted list),
2. Una lista no ordenada doblemente enlazada (doubly-linked unsorted list),
3. Una lista ordenada de enlace único (singly-linked sorted list),
4. Una lista ordenada doblemente enlazada (doubly-linked sorted list).

Solución: Deben considerarse dos aspectos diferentes al evaluar estas implementaciones: listas de enlace simple frente a listas de enlaces dobles y orden ordenado frente a no ordenado.

Tabla

Descripción generada automáticamente

* Inserción / Eliminación – La complicación aquí es la eliminación de la lista de un solo enlace. La definición de la operación eliminar establece que se nos da un puntero X para el elemento a eliminar. Pero lo que realmente necesitamos es un puntero al elemento apuntando a X en la lista, porque ese es el nodo que necesita ser cambiado. No podemos hacer nada sin esta lista predecesora, y por eso debemos gastar tiempo lineal buscándolo en una lista de enlaces individuales. Las listas doblemente enlazadas evitan este problema, ya que podemos recuperar inmediatamente la lista predecesora de x. La eliminación es más rápida para listas ordenadas doblemente enlazadas que para matrices ordenadas, porque dividir el elemento eliminado de la lista es más eficiente que completar el agujero moviendo los elementos de la matriz. El problema del puntero predecesor de nuevo complica la eliminación de listas ordenadas de enlace único.
* Buscar – Ordenar proporciona menos beneficios para las listas enlazadas que para las matrices. La búsqueda binaria ya no es posible porque no podemos acceder al elemento en la mitad sin atravesar todos los elementos anteriores. Lo que proporcionan las listas ordenadas es terminación rápida de búsquedas fallidas, porque si no hemos encontrado a ***Abbot***  por el momento que estamos en ***Costello*** podemos deducir que él no existe. Seguir buscando, toma un tiempo lineal en el peor caso.
* Operaciones transversales: el problema del puntero predecesor vuelve a complicar implementando el predecesor. El sucesor lógico es equivalente al nodo sucesor para ambos tipos de listas ordenadas y, por lo tanto, puede implementarse en tiempo constante.
* Máximo: el elemento máximo se encuentra al final de la lista, lo que normalmente requiere O(n) tiempo para llegar a las listas de enlaces simples o dobles. Sin embargo, podemos mantener un puntero separado a la cola de la lista, pero tenemos que pagar los costos de mantenimiento de este puntero en cada inserción y eliminación. El puntero de cola se puede actualizar en tiempo constante en listas doblemente enlazadas: en verificación de inserción si último -> siguiente aún es igual a null, y en la eliminación seteamos ultimo para apuntar a la lista predecesora de ultimo si se elimina el último elemento.

No tenemos una forma eficiente de encontrar este predecesor para listas de enlaces simples. Así que ¿Por qué podemos implementar el máximo en O(1) en listas de enlaces simples? El truco es cargar el costo a cada borrado, que ya tomaba tiempo lineal. Agregando un barrido lineal adicional para actualizar el puntero no daña la asintótica complejidad de eliminar, mientras nos gana Máximo en tiempo constante como recompensa para un pensamiento claro.

**Capítulo 20: Tablas Hash**

La tabla hash se utiliza para implementar un conjunto en tiempo constante por cada operación.

La tabla hash soporta la extracción o borrado de cualquier elemento nominado. Queremos poder soportar las operaciones básicas en un tiempo constante, al igual que para la pila y la cola. Puesto que los accesos están mucho menos restringidos, este soporte parece casi un objetivo imposible de alcanzar. Es decir, con toda seguridad, cuando el tamaño del conjunto se incremente, las búsquedas en el mismo deberán requerir más tiempo. Sin embargo, veremos que esto no es así.

Suponga que todos los elementos con los que estamos tratando son enteros pequeños no negativos, comprendidos entre 0 y 65.535. Podemos utilizar una matriz simple para implementar cada operación de la forma siguiente. En primer lugar, inicializamos una matriz **a** indexada entre 0 y 65.535 con todo 0s. Para realizar **insert(i)**, ejecutamos **a[i]++**. Observe que **a[i]** representa el número de veces que **i** ha sido insertado. Para realizar **find(i)**, verificamos que **a[i]** no sea 0. Pada realizar **remove(i)**, nos aseguramos de que **a[i]** sea positivo y luego ejecutamos **a[i]--**. El tiempo requerido por cada operación es obviamente constante; incluso el coste adicional de inicialización de la matriz requiere una cantidad constante de trabajo (65.535 asignaciones).

Hay dos problemas con esta solución:

En primer lugar, suponga que tuviéramos enteros de 32 bits, en lugar de 16 bits. Entonces, la matriz **a** debería contener 4.000 millones de elementos, lo que resulta complicado desde el punto de vista práctico. En segundo lugar, si los elementos no son enteros, sino cadenas de caracteres (o incluso algo más genérico), no pueden utilizarse para indexar una matriz. El segundo problema no es, en realidad, un problema en absoluto. Al igual que un número 1234 es una colección de dígitos 1, 2, 3 y 4, la cadena de caracteres “junk” es una colección de caracteres “j”, “u”, “n” y “k”. Observe que el número 1234 es simplemente 1 . 103 + 2 . 102 + 3 . 101 + 4 . 100. Puesto que un carácter es básicamente un entero de pequeño tamaño, podemos interpretar una cadena de caracteres como un entero. Una posible representación sería “j” . 1283 + “u” . 1282 + “n” . 1281 + “k” . 1280.

El problema con esta estrategia es que la representación en forma de entero que acabamos de describir nos proporciona números enteros de un tamaño enorme. La representación para “junk” nos da 224.229.227, y las cadenas de mayor longitud generan representaciones todavía mayores. ¿Cómo evitamos utilizar una matriz de un tamaño absurdamente grande?

Para hacerlo, empleamos una función que asigna los números de gran tamaño (o las cadenas de caracteres interpretadas como números) a números más pequeños y manejables. Una función que establezca una correspondencia entre un elemento y un índice de pequeño tamaño se conoce con el nombre de función hash.

**\*Una función hash convierte el elemento en un entero que sea adecuado para indexar una matriz en la que se almacene el elemento. Si la función hash fuera biunívoca podríamos acceder al elemento empleando su índice matricial\***

Si **x** es un entero arbitrario (no negativo), entonces **x % tableSize** genera un número comprendido entre 0 y tableSize – 1 que es adecuado para indexar en una matriz de tamaño tableSize. Si **s** es una cadena de caracteres, podemos convertir **s** en un entero de gran tamaño **x** utilizando el método sugerido anteriormente y luego aplicar el operador mod para obtener un índice adecuado. Así, si tableSize es 10.000, “junk” se indexaría como 9.227.

El uso de la función hash introduce una complicación: puede que haya dos o más elementos distintos a los que les corresponda la misma posición, provocando una ***colisión***. Esta situación no se puede evitar nunca, porque hay muchos más elementos que posiciones disponibles. Sin embargo, existen muchos métodos para resolver rápidamente una colisión (sondeo lineal, sondeo cuadrático y encadenamiento separado).

**Función hash**

Calcular la función hash para cadenas de caracteres presenta una sutil complicación: la conversión de la String **s** a **x** genera un entero que casi con seguridad será mayor que los valores que la máquina permite almacenar, porque 1284 = 228. Este tamaño de entero es solo 8 veces menor que el valor int más grande permitido. En consecuencia, no podemos esperar obtener el valor de la función hash calculando una potencia de 128. En lugar de ello, utilizamos la siguiente observación.

Un polinomio génerico

A3X3 + A2X2 + A1X1 + A0X0 20.1

Puede evaluarse como

(((A3)X + A2)X + A1)X + A0 20.2

Observe que en la Ecuación 20.2 evitamos calcular el polinomio directamente, lo cual resulta conveniente por tres razones distintas. En primer lugar, evita tener un resultado intermedio de gran tamaño, el cual como ya hemos visto, provocaría un desbordamiento. En segundo lugar, el cálculo de esa ecuación implica únicamente tres multiplicaciones y tres sumas; un polinomio de grado N se calcula en N multiplicaciones y sumas. Estas operaciones son menos costosas que el cálculo de la ecuación 20.1. En tercer lugar, el cálculo se hace de izquierda a derecha (A3 se corresponde con “j”, A2 se corresponde con “u”, etc. Y X es 128).

Sin embargo, sigue existiendo un problema de desbordamiento. El resultado del cálculo sigue siendo el mismo y es probable que sea demasiado grande. Pero aquí hay que recordar que solo necesitamos el resultado mod tableSize. Aplicando el operador % después de cada multiplicación (o suma), podemos cerciorarnos de que los resultados intermedios sigan siendo pequeños.



Una desventaja de esta función hash es que el cálculo del operador mod es costosa, pero como el desbordamiento está permitido (y sus resultados son coherentes en una plataforma determinada), podemos acelerar un poco más la función hash realizando una única operación mod inmediatamente antes de volver. Lamentablemente, la multiplicación repetida por 128 tendería a desplazar los primeros caracteres hacia la izquierda, fuera de la respuesta. Para aliviar este problema, multiplicamos por 37 en lugar de por 128, lo que ralentiza el desplazamiento de los primeros caracteres.

El resultado se muestra en la Figura 20.2, no se trata necesariamente de la mejor de las funciones posibles. Asimismo, en algunas aplicaciones (por ejemplo, si estamos utilizando cadenas de caracteres largas), nos gustaría mejorarla. Sin embargo, hablando en términos generales, la función es bastante buena. Observe que el desbordamiento podría introducir números negativos. Por ello, si la operación mod genera un valor negativo, lo hacemos positivo (líneas 15 y 16). Observe también que el resultado obtenido al permitir el desbordamiento y realizar una operación mod al final no coincide con el que se obtiene al realizar la operación mod después de cada paso. Por tanto, hemos modificado ligeramente la función hash.

Aunque la velocidad es un aspecto importante a la hora de diseñar una función hash, también queremos asegurarnos de que distribuye las claves de manera equitativa. En consecuencia, no debemos llevar demasiado lejos nuestras optimizaciones. Un ejemplo sería la función hash mostrada en la Figura 2.3, que simplemente suma los caracteres que forman la clave y devuelve el resultado mod tableSize. Difícilmente encontraríamos una función más simple. La función es fácil de implementar y calcula un valor hash muy rápidamente. Sin embargo, si tableSize es grande, la función no distribuye las claves demasiado bien.

Por ejemplo suponga que tableSize es 10.000. Suponga también que todas las claves tienen una longitud de 8 caracteres o menos. Puesto que un char ASCII es un entero entre 0 y 127, la función hash solo puede asumir valores comprendidos entre 0 y 1.016 (127 x 8)). Esta restricción no permite, obviamente, una distribución equitativa. Cualquier velocidad que hayamos ganado al acelerar el cálculo de la función hash se verá más que compensado por el esfuerzo necesario para resolver un número de colisiones mayor que el esperado. Finalmente, observe que 0 es un posible resultado de la función hash, por lo que las tablas hash se indexan comenzando en 0.

**\*La tabla va de 0 a tableSize -1\***

**\*La función hash debe ser simple de calcular, pero también debe distribuir las claves de manera equitativa. Si hay demasiadas colisiones, el rendimiento de la tabla hash se verá enormemente afectado\***

**Texto

Descripción generada automáticamente**

**Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente**

**Sondeo Lineal**

**\*En el sondeo lineal, las colisiones se resuelven explorando secuencialmente una matriz (con encadenamiento circular entre el final y el principio), hasta encontrar una celda vacía\***

Ahora que disponemos de una función hash, tenemos que decidir que hacer cuando se produce una colisión. Específicamente, si al aplicar la función hash a **x** obtenemos una posición que ya está ocupada, ¿dónde colocamos ese elemento? La estrategia más simple posible es la del sondeo lineal, que consiste en buscar secuencialmente en la matriz hasta encontrar una celda vacía. La búsqueda salta circularmente desde la última posición hasta la primera en caso necesario. La Figura 20.5 muestra el resultado de insertar las claves 89, 18, 49, 58 y 9 en una tabla hash, cuando se utiliza un sondeo lineal. En el ejemplo estamos asumiendo que disponemos de una función hash que devuelve la clave X mod el tamaño de la tabla.

La primera colisión se produce al insertar 49; el 49 se coloca en la siguiente celda disponible -en concreto, en la celda 0, que está vacía. Después, 58 colisiona con 18, 89 y 49 antes de encontrar una celda vacía, tres posiciones más allá, en la posición 1. La colisión para el elemento 9 se resuelve de forma similar. Mientras que la tabla sea lo suficientemente grande, siempre se podrá encontrar una celda vacía. Sin embargo, el tiempo necesario para encontrar una celda vacía puede llegar a ser muy largo. Por ejemplo, si solo queda una celda vacía en la tabla, puede que tengamos que realizar una búsqueda de tabla completa para encontrarlo. En promedio, cabría esperar tener que explorar la mitad de la tabla para encontrarlo, lo cual está bastante lejos del tiempo constante por acceso que estábamos intentando obtener. Pero si la tabla está relativamente vacía, las inserciones no deberían ser demasiado costosas.

Texto, Carta

Descripción generada automáticamente

Tabla

Descripción generada automáticamente

**\*El algoritmo find sigue la misma secuencia de sondeo que el algoritmo insert\***

El algoritmo find simplemente sigue la misma técnica que el algoritmo insert. Si llega a una celda vacía, querrá decir que el elemento que estamos buscando no se ha encontrado; en caso contrario, terminará por encontrar la correspondencia deseada. Por ejemplo, para encontrar 58, comenzamos por la celda 8 (tal como nos ha indicado la función hash). Vemos allí un elemento, pero no es el que estamos buscando, así que probamos con la celda 9. De nuevo, tenemos un elemento, pero no es el deseado, así que buscamos en la celda 0 y luego en la celda 1, hasta encontrar una correspondencia. Una operación find para el valor 19 implicaría probar las celdas 9, 0, 1 y 2 antes de encontrar una posición vacía en la celda 3. Eso querrá decir que no se ha encontrado el valor 19.

No se puede realizar un borrado estándar porque, como sucede con los árboles de búsqueda binaria, cada elemento de la tabla hash no solo se representa a sí mismo, sino que también conecta entre sí otros elementos, sirviendo como posible celda de almacenamiento durante la resolución de colisiones. Por tanto si eliminamos 89 de la tabla hash, casi todas las restantes operaciones find fallarán. En consecuencia, lo que hacemos es implementar un mecanismo de borrado perezoso, que consiste en marcar los elementos como borrados en lugar de eliminarlos físicamente de la tabla. Esta información se almacena en un miembro de datos adicional. Cada uno de los elementos de la tabla podrá estar activo o borrado.

**Sondeo cuadrático**

El sondeo cuadrático es un método de resolución de colisiones que elimina el problema del agrupamiento primario que presenta el sondeo lineal. Lo consigue examinando determinadas celdas situadas a una cierta distancia del punto de sondeo original. Su nombre deriva del uso de la fórmula F(i) = i2 para resolver las colisiones. Específicamente, si la función hash se evalúa como H y una búsqueda en la celda H no permite finalizar la operación, probamos sucesivamente con las celdas H + 12, H + 22, H + 32, …, H + i2 (con encadenamiento circular entre el final y el principio de la tabla). Esta estrategia difiere de la estrategia de sondeo lineal consistente en buscar en H + 1, H + 2, H + 3, …, H + i.

La figura 20.7 muestra la tabla que resulta cuando se emplea el sondeo cuadrático en lugar del sondeo lineal para la secuencia de inserción mostrada en la Figura 20.5. Cuando 49 colisiona con 89, la primera alternativa que se intenta está situada a una celda de distancia. Esta celda está vacía, por lo que se coloca allí el 49. A continuación, el 58 colisiona en la posición 8. Se prueba con la celda situada en la posición 9 (que está a una celda de distancia), pero se produce otra colisión. Se encuentra una celda vacante en la siguiente celda que se prueba, que está a 22 = 4 posiciones de distancia, con respecto a la posición hash original. De ese modo, el 58 se coloca en la celda 2. Lo mismo sucede con el 9.

Tabla

Descripción generada automáticamente

**\*Recuerde que los puntos de sondeo posteriores están a un número cuadrático de posiciones de distancia con respecto al punto de sondeo original\***

Observe que las posiciones alternativas para los elementos a los que les corresponde un resultado hash de 8 y las posiciones alternativas para los elementos a los que les corresponde un resultado hash de 9 no coincide. La larga secuencia de sondeo para insertar 58 no tuvo ningún efecto sobre la inserción subsiguiente de 9, lo que contrasta con lo que sucedía en el caso del sondeo lineal.

* En el sondeo lineal, cada sondeo prueba con una celda diferente. ¿Está garantizado en el sondeo cuadrático que, cuando se pruebe celda, no la hayamos probado ya durante el curso del acceso actual? ¿Está garantizado en el sondeo cuadrático que, cuando estemos insertando X y la tabla no esté llena, terminaremos insertando X?
* El sondeo lineal se puede implementar fácilmente. El sondeo cuadrático parece requerir operaciones de multiplicación y de módulo. ¿Esta aparente complejidad añadida hace que el sondeo cuadrático resulte poco práctico?
* ¿Qué sucede (tanto en el sondeo lineal como en el cuadrático) si el factor de carga se hace demasiado alto? ¿Podemos expandir dinámicamente la tabla, como suele hacerse con otras estructuras de datos basadas en matrices?

Afortunadamente, las noticias son relativamente buenas en todos los casos citados. Si el tamaño de la tabla es primo y el factor de carga nunca es mayor de 0,5, siempre podremos insertar un nuevo elemento X y ninguna celda será sondeada dos veces durante un mismo acceso. Sin embargo, para que estas garantías se cumplan, tenemos que asegurarnos de que el tamaño de la tabla sea un número primo.

**\*Si el tamaño de la tabla es primo y el factor de carga no es mayor que 0,5, todos los sondeos se realizarán en posiciones diferentes y siempre se podrá insertar un elemento\***

**20.5 Hash con encadenamiento separado**

Una alternativa bastante popular y muy eficiente en términos de espacio al sondeo cuadrático es el hash con encadenamiento separado, en el que se mantiene una matriz de listas enlazadas. Para una matriz de listas enlazadas, L0, L1, … , Lm – 1, la función hash nos dice en que lista hay que insertar un elemento X y luego, durante una operación find, que lista contiene a X. La idea es que, aunque la búsqueda en una lista enlazada es una operación lineal, si las listas son suficientemente cortas, el tiempo de búsqueda será muy rápido. En particular, suponga que el factor de carga, NM, es gamma y que no está acotado por 1,0. En ese caso, la lista típica tendrá una longitud gamma, haciendo que el numero esperado de sondeos para una inserción o una búsqueda que no tenga éxito sea gamma, y que el numero esperado de sondeos para una búsqueda que tenga éxito sea 1 + 1/2gamma. La razón es que una búsqueda con éxito deberá producirse en una lista no vacía y que, en dicha lista, cabe esperar que tengamos que recorrer la mitad de la misma. El coste relativo con éxito frente a una búsqueda que no lo tenga es inusual, en el sentido que, si gamma < 2, la búsqueda con éxito es más costosa que la que no lo tiene. Sin embargo, esta condición tiene bastante sentido, porque muchas búsquedas sin éxito van a encontrarse con una lista enlazada vacía.

Un factor de carga típico es 1,0; los factores de carga menores no mejoran el rendimiento de manera significativa, y además requieren malgastar espacio adicional. El atractivo del mecanismo de encadenamiento separado es que el rendimiento no se ve afectado por un incremento moderado del factor de carga; por tanto, puede evitarse el rehashing. Para aquellos lenguajes que no permiten la expansión dinámica de matrices, este aspecto resulta importante. Además, el numero esperado de sondeos en una búsqueda es inferior que en el sondeo cuadrático particularmente para las búsquedas que no tenga éxito.

Podemos implementar el mecanismo de encadenamiento separado utilizando nuestras clases existentes de listas enlazadas. Sin embargo, como el nodo de cabecera representa un gasto de espacio y no es realmente necesario, si el espacio fuera un problema podríamos elegir no reutilizar componentes, implementado en su lugar una lista simple de tipo pila. El esfuerzo de codificación es bastante pequeño. Además, el coste en términos de espacio es esencialmente de una referencia por nodo, más una referencia adicional por cada lista; por ejemplo, cuando el factor de carga es 1.0 tendremos dos referencias por cada elemento. Esta característica podría tener su importancia en otros lenguajes de programación si el tamaño de un elemento fuera grande.

En ese caso, tendríamos los mismos compromisos que en el caso de la implementación de las pilas con matrices o con listas enlazadas. La API de Colecciones de Java utiliza el mecanismo de encadenamiento separado para tablas hash con un factor de carga predeterminado de 0.75.

**Aplicaciones de las tablas hash**

Las aplicaciones de las tablas hash son muy numerosas. Los compiladores utilizan tablas hash para llevar la cuenta de las variables declaradas en el código fuente. La estructura de datos correspondiente se llama tabla de símbolos. Las tablas hash son la aplicación ideal de este problema específico, porque solo se realizan operaciones insert y find. Los identificadores son normalmente cortos, por lo que la función hash se puede calcular rápidamente. En esta aplicación, la mayoría de las búsquedas tienen éxito.

Otro uso común de las tablas hash es en los programas de juegos. A medida que el programa explora a través de las diferentes líneas de juego, lleva la cuenta de las posiciones con las que ya se ha encontrado, calculando una función hash basada en la posición (y almacenando su movimiento a partir de esa posición). Si vuelve a aparecer la misma posición, usualmente debido a una simple transposición de movimientos, el programa puede ahorrarse una costosa serie de recálculos. Esta característica general de todos los programas de juegos se denomina tabla de transposición.

Un tercer uso de las tablas hash es en los comprobadores ortográficos en línea. Si es importante la detección de errores ortográficos (en lugar de la corrección de esos errores), se puede calcular el valor hash de cada una de las palabras de un diccionario completo y comprobar las palabras en un tiempo constante. Las tablas hash están muy adaptadas a esta aplicación, porque las palabras no tienen por qué estar alfabetizadas.

**NOTAS:**

Las tablas hash se pueden utilizar para implementar las operaciones insert y find en un tiempo promedio constante. Es especialmente importante prestar atención a detalles tales como el factor de carga a la hora de utilizar las tablas hash; en caso contrario, las cotas de tiempo constante dejan de tener sentido. También es importante seleccionar la función hash con cuidado cuando la clave no sea un valor entero o una cadena de caracteres corta. Hay que elegir una función fácilmente calculable y que distribuya equitativamente los valores.

Para la técnica de encadenamiento separado, el facto de carga es normalmente próximo a 1, aunque el rendimiento no se degrada significativamente a menor que el factor de carga sea muy grande. En el caso del sondeo cuadrático, el tamaño de la tabla debe ser un número primo y el factor de carga no debe sobrepasar el valor 0,5. Debe utilizarse el rehashing para el sondeo cuadrático con el fin de permitir que la tabla crezca y mantener así el factor de carga correcto. Esta técnica cobra su importancia en caso de que el espacio escasee y no sea posible declarar simplemente una tabla hash de tamaño enorme.

Api de colecciones de Java

6.2 Patrón Iterador

La API de colecciones hace uso del patrón iterador.

Considere el problema de imprimir los elementos de una colección.

Normalmente, la colección será una matriz, por lo que si asumimos que el objeto v es una matriz, podemos imprimir fácilmente su contenido con un código como el siguiente:

for (int i = 0; i < v.length ; i++)

System.out.printIn( v[ i ] );

En este bucle “i” es un objeto iterador, porque es el que se emplea para controlar la iteración. Sin embargo, utilizar el entero i de esta manera restringe el diseño: solo podemos almacenar la colección en una estructura de tipo matricial. Una alternativa más flexible sería diseñar una clase iteradora que encapsule una posición dentro de una colección. Esta clase proporciona métodos para recorrer la colección.

La clave está en usar interfaces, de forma que el tipo del contenedor sea indiferente. Lo que nos permite utilizar distintos diseños de iterador. Si sustituimos int i por iteradorType itr, entonces el bucle anterior nos quedaría

for( itr = v.first( ); itr.isValid( ); itr.advance( ))

System.out.printIn( itr.getData( ));

Donde first nos da el primer elemento de la colección, isValid nos dice si una posición es válida, advance hace avanzar el iterador y getData nos da los datos correspondientes a la posición actual.

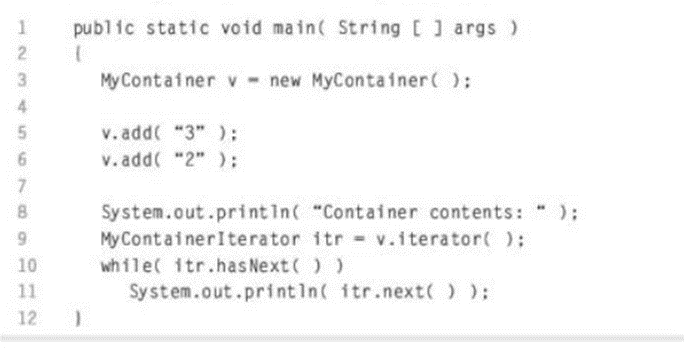
Esto sugiere utilizar una clase iteradora que contenga métodos similares a isValid, advance, getData, etc.

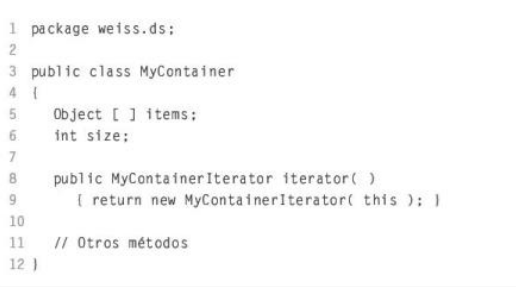
6.2.1 Diseñor básico de un iterador

El primer diseño de iterador utiliza solo tres métodos. Se exige a la clase contenedora que proporcione un método iterator. El cual devuelve un método apropiado para la colección. La clase iteradora solo tiene dos métodos: *hasNext* y *next*. *hasNext* devuelve true si la iteración no ha terminado aún, *next* devuelve el siguiente elemento de la colección (y en el proceso hace avanzar la posición actual).

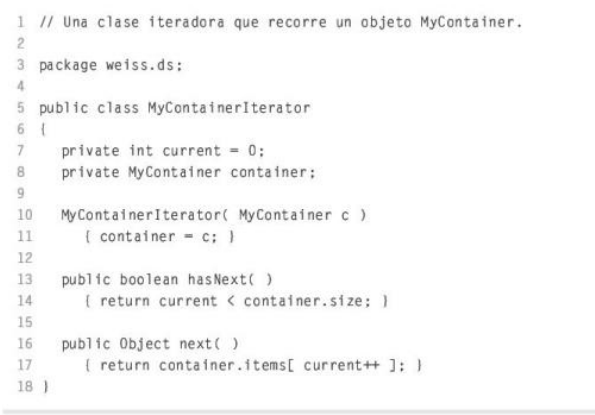
Para ilustrar la implementación de este diseño, vamos a esbozar la clase de colección utilizada y a proporcionar una clase iteradora; estas clases serán MyContainer y MyContainerIterator respectivamente.

El método iterator de la clase MyContainer simplemente devuelve un nuevo iterador; observe que el iterador debe tener información acerca del contenedor en el que está iterando. Por ello, el iterador se construye con una referencia de MyContainer.





La siguiente imagen muestra MyContainerIterator. El iterador mantiene una variable (current) que representa la posición actual dentro del contenedor, así como una referencia al propio contenedor. La implementación del constructor y de los dos métodos es bastante simple. El constructor inicializa la referencia al contenedor. La implementación del constructor y de los dos métodos es bastante simple. El constructor inicializa la referencia al contenedor, *hasNext* compara simplemente la posición actual con el tamaño del contenedor, n*ext* utiliza la posición actual para indexar la matriz (y luego hace avanzar la posición actual).



Una limitación en el diseño de este iterador es la interfaz relativamente limitada. Por ejemplo es imposible hacer que el iterador vuelva al principio y también el método *next* acopla el acceso a un elemento y a la operación de avance del Iterador. Este diseño de *next, hasNext es lo que se utiliza en la API de colecciones de Java.*

*Aunque esto constituye una mejora significativa, si cambiamos una colección basada en matriz a alguna otra cosa, tendremos que modificar todas las declaraciones del iterador.*

Unidad 6

Diccionarios --> permite el acceso a elementos de datos por contenido.

Operaciones principales:

|  |  |  |
| --- | --- | --- |
|  | Array desordenado | Array ordenado |
| Search (D,k) | O(n) | O(logn) |
| Insert (D,x) | O(1) | O(n) |
| Delete (D,x) | O(1) | O(n) |
| Successor (D,x) | O(n) | O(1) |
| Predecessor (D,x) | O(n) | O(1) |
| Minimum (D) | O(n) | O(1) |
| Maximum (D) | O(n) | O(1) |

n --> número de elementos del array

En un array desordenado todas sus operaciones son O(n) menos insertar (porque se implementa incrementando n y copiando x en D[n]) y eliminar (porque escribe sobre D[x] con D[n] y decrementa n). Mientras que en un array ordenado insertar debe recorrer todos los items y eliminar también.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Operaciones: | Simplemente enlazada desordenada | Doblemente enlazada desordenada | Simplemente enlazada ordenada | Doblemente enlazada ordenada |
| Search (D,k) | O(n) | O(n) | O(n) | O(n) |
| Insert (D,x) | O(1) | O(1) | O(n) | O(n) |
| Delete (D,x) | O(n) | O(1) | O(n) | O(1) |
| Successor (D,x) | O(n) | O(n) | O(1) | O(1) |
| Predecessor (D,x) | O(n) | O(n) | O(n) | O(1) |
| Minimum (D) | O(n) | O(n) | O(1) | O(1) |
| Maximum (D) | O(n) | O(n) | O(1) | O(1) |

Tablas Hash (hashing) --> inserción, borrados y busqueda en O(1).

Muy suficiente aunque a expensas de perdida de ordenacion de los elementos, lo que hace que findMin y findMax no puedan ser constantes y la impresión de una tabla ordenada.

Una funcion que establezca una correspondencia entre un elemento y un indice de tamaño pequeño se conoce como funcion hash. Entonces si x es un entero arbitrario (no negativo) entonces x % tableSize genera un numero entre 0 y tableSize-1 que es perfecto para indexar en una matriz de tamaño tableSize.

En caso de que hayan 2 o mas elementos a los que se les corresponda la misma posicion, provoca una colision, lo que es imposible de evitar. Existen 3 metodos para resolver colisiones, son: sondeo lineal, sondeo cuadratico y encadenamiento separado.

Funcion Hash aceptable

hash(string key, int tableSize)

COM

hasval = 0

For(i=0, i < key.length(); i++)

hashval = (hashval \* 128 + key.charAt(i)) % tableSize

Devolver hashval

FIN

La clase String cuenta con un metodo hashCode (optimizado) (gracias al almacenamiento en cache, se puede volver a generar el hashCode sin tener que pagar el costo de calculo, gracias a que los Strings son inmutables)

Sondeo lineal (estrategia más simple) --> consiste en buscar secuencialmente en la matriz hasta encontrar una celda vacía. La búsqueda puede ser muy larga.

El factor de carga de una tabla hash con sondeo es la fracción de la tabla que esta llena. Factor de carga varía entre 0 (vacía) y 1 (llena).

inserción:

Si asumimos que los sondeos son independientes, el promedio de celdas examinadas durante una inserción utilizando sondeo lineal es 1/(1-factor de carga).

El promedio de celdas examinadas en una inserción utilizando sondeo lineal es aproximadamente (1 + 1/(1 – factor de carga) ^2) / 2, si no asumimos que los sondeos son independientes.

Busqueda:

El promedio de celdas examinadas en una busqueda que no tenga éxito utilizando un sondeo lineal es aproximadamente (1 + 1/ (1 – factor de carga) ^2) /2. Y en una busqueda con éxito es aproxiamdamente (1+1/(1-factor de carga))/2.

Sondeo cuadratico --> elimina el problema del agrupamiento primario que presenta el sondeo lineal. Lo consigue examinando las celdas que estan situadas a una distancia de 1, 4, 9 … del punto de sondeo original. (H+1^2, H+2^2, … , H+i^2).

Si el tamaño de la tabla es primo y el factor de carga no es mayor a 0.5 (medio vacio), en la insercion ninguna celda sera sondeada 2 veces, nunca fallara la insercion. En cambio si la tabla esta mas que medio llena, aunque sea por 1 posicion, la insercion podria fallar.

Rehashing –> es cuando se debe crear otra tabla hash, entonces se debe examinar cada elemento, calcular su nuevo hash e insertarlo en la nueva tabla hash.

Agrupamiento secundario –> provocar sondeos en mismas celdas alternativas.

Doble hash –> tecnina para el agrupamiento secundario, se utiliza una segunda funcion hash para dirigir la resolucion de colisiones. (El resultado de la segunda funcion nunca debe ser 0).

Hash con encadenamiento separado --> se mantiene una matriz de listas enlazadas. Es menos sensible a los factores de carga altos, por lo tanto, se evita el rehashing. El numero esperado de sondeos en una búsqueda es menor que el cuadrático, particularmente para las búsquedas sin éxito.

Borrado perezoso --> consiste en marcar los componentes como borrados en lugar de eliminarlos físicamente de la tabla hash. Es necesario a la hora de sondear.

Factor de carga --> número de elementos dividido el tamaño de la tabla. Representa la fracción de la tabla que está llena.

HashSet (implementa interfaz set). Difiere con el TreeSet en que no se puede usar para enumerar elementos por orden ni tampoco para obtener mayor y menor (los elementos del HashSet no tienen que ser comparables). HashSet es menos potente que TreeSet, pero mejor rendimiento.

Si se itera el HashSet, el orden no está definido. HashSet no acepta duplicados.

Una colección es un objeto que agrupa múltiples elementos en una sola unidad.

Un mapa se convierte en una collections de las siguientes formas:

* KeySet }
* Values } Collections views --> única forma de iterar sobre un mapa
* entrySet }

Análisis de los métodos de hashing

* Inserción y recuperación del peor caso muy malo
* Probabilidad de encontrar una posición libre la primera vez: (N-k)/N

Hashing desventajas

* Tamaño fijo de la tabla. En caso de que se conozca el tamaño del conjunto, para lograr un buen rendimiento se dimensiona la tabla 10% más.
* Si además de insertar y buscar, también es necesario eliminar, estas estructuras son muy ineficientes

Interfaz Set

HashSet: una tabla hash

* La más eficiente
* Recorrido sin orden

TreeSet: arbol de tipo rojo-negro

* Menos eficiente
* Recorridos ordenados por elementos

LinkedHasSet: tabla de hash más lista enlazada

* Coste mayor que HashSet
* Recorridos ordenados por insercion

Interfaz Map para tablas Hash

* No permite claves duplicadas
* Implementaciones: HashMap, TreeMap, LinkedHashMap, EnumMap