Diccionarios y árboles de búsqueda balanceados

Introducción

Previamente hablamos de tipos de dato abstractos y de cómo hay distintas formas de implementarlos usando diferentes estructuras de datos. A grandes rasgos, el tipo de dato abstracto es lo que queremos almacenar y las operaciones que queremos que tenga. La estructura de datos es el cómo hacer esto.

En esta ocasión hablaremos de los `Diccionarios`, que es un tipo de dato abstracto que nos permite almacenar información y consultarla. Veremos brevemente algunas ventajas y desventajas de implementar a los diccionarios con listas enlazadas o arreglos. Después, veremos que hay una implementación más «pareja» usando árboles.

Diccionarios

Un `Diccionario` es un tipo de dato abstracto que guarda parejas del estilo `{llave: valor}`, en donde no hay dos `llave` iguales (aunque quizás sí se repite el `valor`). Es justo como un diccionario en la vida real, en donde guardamos parejas `{palabra: definición}`. Sin embargo, los diccionarios con los que trataremos serán mucho más generales, pues en realidad hay pocas operaciones que queremos que tengan. Para nuestros fines, basta que en un `Diccionario` se puedan realizar las siguientes operaciones:

- `Buscar(k)` - Dada la `llave k`, queremos encontrar su `valor`, si es que este existe.
- `Insertar(k,x)` - Agrega al diccionario la pareja `{k:x}`, si es que no hay ninguna pareja con `llave=k`.
- `Reasignar(k,x)` - Remplaza en el diccionario la pareja `{k:valor}` (si es que existe) con la pareja `{k:x}`
- `Eliminar(k)` - Elimina del diccionario la pareja `{k:valor}`, si es que existe.

En algunas implementaciones también se incluyen las siguientes operaciones:

- `Mínimo` - Se regresa la pareja `{llave:valor}` con el mínimo valor posible de `llave`.
- `Máximo` - Se regresa la pareja `{llave:valor}` con el máximo valor posible de `llave`.

Implementación con listas enlazadas o arreglos

Es posible implementar un `Diccionario` con una variante de las listas enlazadas llamada **lista de asociación**. La idea es exactamente la misma que una lista enlazada, pero ahora cada nodo guarda una pareja del estilo `(llave,valor)` y una referencia al siguiente nodo.

La operación de `Buscar` se puede hacer en tiempo $O(n)$ recorriendo todos los elementos para ver si la llave coincide con alguna. La de `Insertar` se puede hacer en tiempo $O(1)$ si llevamos registro del último nodo (cuidando no insertar llaves repetidas). La de `Reasignar` y `Eliminar` se pueden hacer en tiempo $O(n)$, buscando primero la llave que se debe alterar y luego cambiando su valor en tiempo $O(1)$.

También es posible implementar un `Diccionario` usando arreglos o arreglos dinámicos. Si suponemos que nuestras llaves son valores enteros, podemos guardar el `valor` de la pareja `{llave:valor}` en la entrada `A[llave]` del arreglo. Mientras no tengamos que almacenar un valor con esta llave, podemos poner un valor «de a mentiras» que nos indique que esa llave no tiene valor asignado.

Es posible hacer las operaciones de `Diccionario` muy rápido usando arreglos. Las consultas en un arreglo se hacen en tiempo $O(1)$ y esto mismo permite reasignar y eliminar en tiempo constante. Aquí una desventaja es que potencialmente debemos reservar espacio $O(M)$ en donde M es el conjunto de todas las posibles llaves que existen.

Por estas razones, típicamente los `Diccionarios` se implementan con otras estructuras de datos.

Tablas hash

La idea de las **tablas hash** es combinar las ventajas de los tiempos $O(1)$ de las operaciones de `Diccionario` implementadas con arreglos, pero reduciendo el universo de posibles llaves mediante una **función hash**.

A grandes rasgos, una función hash es una función $f:A\to B$ en donde A es un conjunto grande de donde usualmente tomaremos pocos elementos (aunque no sabemos cuáles) y B es un conjunto chico. Dado que A es usualmente mucho más grande que B, esta función f es para nada inyectiva. Cuando dos elementos de A que nos interesan se van al mismo elemento de B, decimos que hay una **colisión**. Es importante saber qué hacer con colisiones y qué tanto afectan a la aplicación que nos interesa.

Las funciones hash pueden tener ciertas propiedades deseables como las siguientes:

- Ser **uniformes**: Que los valores de A se repartan equitativamente en los de B.
- Ser **eficientes**: Que se puedan calcular muy fácilmente, para que al usarlas como subrutinas, no tomen mucho tiempo.
- Que sean **deterministas** es decir, que su valor no dependa de algo aleatorio.
- Que sean **univerales**, lo cual quiere decir que «parezcan aleatorias».

Árboles binarios de búsqueda auto-balanceables

Otra implementación común de diccionarios es mediante el uso de árboles binarios de búsqueda auto-balanceables. Veamos qué quiere decir esto poco a poco.

Cuando hablemos de **árboles binarios**, nos referimos a una estructura de datos en donde hay un **nodo raíz** y además cada nodo tiene dos posibles **hijos**: un **hijo izquierdo** y un **hijo derecho**. Además de esto, cada nodo puede guardar otro tipo de información.

En un **árbol binario de búsqueda** cada nodo guarda una llave `k` y un valor `x`, y además se cumple que para cualquier vértice la llave de su hijo izquierdo (si existe) es menor que la del vértice, y la de su hijo derecho (si existe) es mayor que la del vértice. En un árbol de búsqueda binario se puede realizar el algoritmo de búsqueda binaria que hemos platicado para encontrar un nodo con llave k (si existe) en tiempo a lo más h, donde h es la altura del árbol.

Como queremos que h sea lo más pequeño posible, nos conviene que los vértices queden «repartidos equitativamente» en cada paso. Esto nos lleva a la noción de un **árbol binario balanceado** en donde para cada vértice los subárboles que tiene a la derecha y a la izquierda difieren en a lo más 1 en altura.

Es posible implementar una estructura de datos llamada **árbol binario de búsqueda auto-balanceado** en donde siempre se preserva que vamos almacenando registros de manera que se cumple todo lo siguiente:

- El árbol siempre es binario balanceado.
- Se puede buscar una llave en tiempo $O(\log n)$.
- Se puede insertar, eliminar o reasignar un nodo con cierta llave en tiempo $O(\log n)$.
- Se puede calcular el mínimo o el máximo en tiempo $O(\log n)$.

Tarea moral

Los siguientes problemas te ayudarán a practicar lo visto en esta entrada. Para resolverlos, necesitarás usar herramientas matemáticas, computacionales o ambas.

1. Explica cómo implementarías las operaciones de `Diccionario` cuando son listas enlazadas, y cuando son arreglos.
2. Demuestra que un árbol binario de altura h puede tener como mucho 2^h-1 vértices. Explica por qué esto dice que un árbol binario con n vértices tiene altura de tamaño $\Omega(\log n)$.
3. A continuación se enuncian algunas funciones hash. Investiga qué hacen, qué tan rápido se calculan, cuáles son sus ventajas,

desventajas y aplicaciones.

- MD5
- SHA-1
- SHA-256
- bcrypt

4. En un árbol de búsqueda balanceado se pide que para todo nodo la altura de su subárbol izquierdo y su subárbol derecho no difieran en más de \(1\). Esto permite hacer búsquedas binarias en tiempo \(O(n\log n)\). ¿Qué sucede si pedimos que dichas alturas no difieran en más de \(5\)? ¿El tiempo sigue siguendo \(O(n\log n)\)?
5. Investiga cómo se realizan las operaciones de agregar, modificar y eliminar nodos en los árboles binarios de búsqueda auto-balanceables. Realiza una implementación en Python con programación orientada a objetos.

Anterior
Estructuras de datos vs. tipos de dato abstractos

Siguiente
Ordenar eficientemente

Por Leonardo Ignacio Martínez Sandoval