NOMBRE: Diego Fluxá

Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación

IIC2133 — Estructura de Datos y Algoritmos — 2' 2020

Documento de Diseño Tarea 2

1 Utilizando un diccionario

Para resolver este problema nos será de gran utilidad utilizar un diccionario. En este caso, implementaremos un diccionario en forma de arreglo

Primero, este diccionario lo utilizaremos para guardar todo subárbol del árbol original utilizando como clave el resultado de ingresar el subárbol (de altura 2 o mayor por issue 152) en una fórmula de hash y luego en una fórmula de ajuste, y como valor una lista con el ID del nodo o nodos (En caso que ese tipo de subárbol se repita dentro del árbol) raíz, una representación como número (u_int64_t) a partir del formato binario de dicho subárbol y la altura del subárbol (Para poder diferenciar subárboles que pueden tener la misma representación como número por que alguno comienzo con 1 o más 0s). Además, es necesario mencionar que podría darse que 2 o más tipos de árboles posean la misma llave, por lo que, en dicho caso, se creará una lista ligada con cada una de estos tipos de árbol.

Para poblar la tabla recorreremos el árbol de manera incremental, comenzando por la raíz, hasheando el primer árbol de altura 2 y luego usar este valor para calcular el de altura 3 y así sucesivamente. Cuando se termine con los subárboles que contengan a la raiz, se procederá a hacer el mismo proceso con los arboles hijos de la raíz y así sucesivamente hasta ingresar todos los nódos a excepción del último nivel (Dado que no pueden poseer subárboles de largo mayor o igual a 2).

Luego, para determinar si un árbol se encuentra dentro del árbol original, solo se necesitará aplicar la misma función de hash y de ajuste a este nuevo árbol para luego, con este resultado, revisar si existe dicha clave en el diccionario. Luego de esto, en caso de no existir, se determinará que el árbol nuevo no existe dentro del original, y en el caso de que si exista la clave, se procederá a revisar si la representación del árbol y la altura son equivalentes a las guardada como valor. Si lo fuera, se devuelven todos los IDs (guardados en el valor) de los nodos raíz donde se encuentra dicho árbol en el original. En caso de que no coincidan, se avanzará en la lista ligada de dicha llave buscando si la representación del árbol nuevo y su altura coinciden con las del elemento de la lista, devolviendo los IDs de las repeticiones en caso de que coincida con un elemento y, si se acaban los elementos y no se encontró una coincidencia, se devolverá que no se encuentra el árbol nuevo dentro del original.

El diccionario que implementaremos será en forma de arreglo. Para esto, sabemos que debemos establecer un largo inicial del arreglo. Por esta razón, nosotros usaremos como largo inicial la cantidad máxima de subárboles de altura mayor o igual a 2 que se pueden encontrar en un árbol de altura h, de manera que no tendremos que realizar resizing en ningún momento pues nunca se superará este número de árboles guardados.

El número de subárboles posibles de altura mayor o igual a 2 de un árbol de altura h vendrá dado por:

Árboles altura 2 + Árboles altura 3 + ... + Árboles altura h

Aquí podremos observar que existirá 1 árbol de altura 2 para cada nodo del árbol original a excepción de los nodos hojas. Por lo tanto habrán $(2^h - 1) - 2^{h-1}$ subárboles de altura 2.

Esto se repetirá para los arboles de altura h^* con $h^* \leq h$, donde no se contabilizarán no habrán subárboles para los últimos $h^* + 1$ niveles. Por lo tanto habrán $(2^h - 1) - \sum_{j=1}^{h^*-1} 2^{h-j}$ subárboles de altura h^*.

Entonces, el número total de subárboles de alturo mayor o igual a 2 corresponderá a:

$$\sum_{i=2}^{h}((2^h - 1) - \sum_{j=1}^{i-1} 2^{i-j})$$

Y, al desarrollar esto obtendremos que:

$$\text{Número de subárboles} = (2^h - 1) - h$$

Es decir, el número de subárboles de altura mayor o igual a 2 será equivalente al número de nodos del árbol menos su altura.

En conclusión, utilizar un diccionario nos evitará tener que recorrer todo el árbol original en busca de las repeticiones del árbol nuevo, si no que simplemente se tendrá que ingresar al valor de la clave obtenida al aplicar la función de hash y de ajuste al segundo árbol. De esta manera, se obtendrán las ocurrencias del árbol nuevo en el original en un solo paso la mayoría de las veces pues, en caso de existir una colisión (2 o más tipos de árboles dan la misma clave), se tendrá que recorrer una lista ligada (dado que se utilizará encadenamiento) para ver si existen ocurrencias de este segundo árbol, pero aún así estas listas deberían ser de un largo considerablemente menor al total de subárboles, pues se utilizará una función de hash que distribuya uniformemente.

2 Función de hash

La función de hash que utilizaremos corresponderá a una función de hash incremental donde se comenzará por la raíz del árbol que se desea hashear, de la cual obtendremos su color (1 o 0).

Luego, se avanzará por niveles obteniendo un número que lo represente al concatenar todos los colores (en forma de 1s o 0s). Posteriormente, se hará una función bitwise XOR ($\wedge$) entre la concatenación y el valor de hash del subárbol anterior. Entonces, se comenzará realizando un XOR con el valor de la raíz y la representación del nivel 2, luego se realizará un XOR entre el resultado y el nivel 3 y así sucesivamente hasta terminar los niveles del árbol.

3 Análisis de distribución función de hash

Nuestra función de hash distribuirá uniformemente.

Esto se debe a que sabemos que los valores de los nodos entregados están distribuidos de manera uniforme y también sabemos que el operador bitwise XOR distribuye uniformemente si los valores a los cuales se está aplicando fueron distribuidos uniformemente.

Entonces, dado que nuestra función utiliza datos que distribuyen uniformemente y solo el operador bitwise XOR, la función de hash definida distribuye uniformemente.

4 Justificación elección de parámetros de la tabla

- Incremental: La función de hash es incremental debido a que, para calcular el hash de un árbol de altura h con $(h \geq 2)$ se utiliza el hash del árbol con la misma raíz que el original pero de altura $h-1$. Esto nos será de especial utilidad a la hora de poblar la tabla de hash pues podremos agregar un árbol y luego usar su valor de hash para calcular el siguiente subárbol, de manera que evitaremos calcular dos veces el hash del primer árbol.

- Resize: Para nuestra tabla de hash no se definirá una función resize debido a que al crearla se utilizará el número máximo de subárboles posibles a agregarse para designar el largo de la tabla. De esta manera, nos podemos asegurar de que nunca necesite crecer para abarcar más subárboles. Es importante mencionar que esta tabla probablemente nunca se llenará, pues existe la posibilidad de que se presenten colisiones y también puede repetirse el mismo tipo de subárbol dentro del árbol original por lo que no se ocupará un espacio. Esto se cumple especialmente para árboles originales grandes debido a que, por ejemplo, el número de tipos de subárboles de altura 2 son solo 8 (000, 001, 010, 011, 100, 101, 110, 111) mientras que existirán más de 8 subárboles de altura 3, por lo que será inevitable que alguno se repita. Sin embargo, al definir el largo de la tabla como lo hemos hecho, nunca necesitaremos agrandarla, tanto para árboles grandes como pequeños (en árboles pequeños es más probable que se llene la tabla pero nunca necesitará crecer).

- Función de ajuste: Para la función de ajuste se utilizará una función simple y vista en clases que corresponde al método de la división. Es decir:

$$h(k) = k \mod m$$

  Donde el m utilizado corresponde al largo de la tabla, es decir, el número máximo de subárboles, que corresponde a $(2^h - 1) - h$ donde h es la altura del árbol original.
  Como vimos en clases, al aplicarse esta función de ajuste a valores distribuidos de manera uniforme, se mantendrá dicha distribución. Luego, dado que ya demostramos que nuestra función de Hash distribuye uniforme, nos será útil esta función de ajuste, pues continuará distribuyendo uniformemente.

- Encadenamiento: Para nuestra tabla de hash, como se ha mencionado anteriormente, se utilizará encadenamiento, debido a preferencia personal y para lograr evitar tener que hacer resizing de la tabla. Esto se debe a que el encadenamiento acepta un factor de carga $\lambda \approx 1$, mientras que en direccionamiento abierto, un factor de carga $\lambda 0.5$ resulta en inserciones y búsquedas muy lentas.

- Factor de carga: Se puede definir el factor de carga como $\lambda = \frac{n}{m}$ donde n corresponde al número de datos almacenados y m al numero de casillas de la tabla. De todas maneras, dado que utilizaremos encadenamiento y que nuestro m decidido inicialmente corresponde al número máximo teórico de subárboles posibles, no nos será de interés el factor de carga pues nunca superará el límite del encadenamiento, y, por ende, nunca se tendrá que realizar resizing.

- Sondeo: Dado que se utilizará encadenamiento en lugar de direccionamiento, no nos será necesario definir un método de sondeo.