Estructuras de Datos y Algoritmos – IIC2133

Pauta I2

1. Árboles de búsqueda balanceados

a) Como vimos en clase, la inserción de un nodo en un árbol rojo-negro se puede dividir en tres casos; en todos los casos, el nodo *x* recién insertado se pinta inicialmente de rojo:

1) el padre *p* de *x* es negro à estamos listos

2) el padre *p* de *x* es rojo, pero el hermano *q* de *p* es negro (si *p* no tiene hermano, lo suponemos negro) à hacemos una o dos rotaciones y le cambiamos el color a dos nodos [**ustedes tienen que saber los detalles de este caso**]

3) el padre *p* de *x* es rojo, y el hermano *q* de *p* también es rojo (es decir, el nodo recién insertado, su padre y su tío son todos rojos) à como el abuelo *r* de *x* necesariamente es negro, cambiamos los colores de *r* y de sus hijos *p* y *q*; ahora *r* es rojo (y sus hijos *p* y *q* son negros), por lo que hay que revisar el color del padre *s* de *r* :

   s es negro à estamos listos

   s es rojo à repetimos el caso 2) o el caso 3), según corresponda, pero ahora con *s* en lugar de *p*, y el hermano *t* de *s* en lugar de *q*

El caso 3) significa que potencialmente podemos llegar de vuelta hasta la raíz del árbol. Una manera de evitar tener que hacer este recorrido "de vuelta" es ir preparando el árbol a medida que vamos bajando (desde la raíz, cuando estamos buscando el punto en que hay que hacer la inserción), de modo de garantizar que *q* no será rojo —algoritmo de inserción *top-down* :

> Al ir bajando, cuando encontramos un nodo *U* que tiene dos hijos rojos, cambiamos los colores de *U* (a rojo) y de sus hijos (a negro); esto producirá un problema solo si el padre *V* de *U* también es rojo, en cuyo caso aplicamos el caso 2) anterior.

Muestra la ejecución de este algoritmo de inserción *top-down* al insertar la clave 22 en el siguiente ár-bol rojo-negro: la raíz tiene la clave 35; sus hijos, las claves 30 (rojo) y 42 (negro); los dos hijos negros de 30 tienen las claves 25 y 33; los dos hijos rojos de 42 tienen las claves 40 y 45; y los dos hijos rojos de 25, tienen las claves 20 y 27.

Respuesta (estos son, más o menos, los pasos importantes)

Al bajar desde la raíz, con clave 35, no encontramos ningún caso especial hasta que llegamos al nodo negro con clave 25, que tiene ambos hijos rojos: 20 y 27. [**0.5 pts**]

Entonces aplicamos la nueva regla: cambiamos los colores de 25 a rojo y de sus dos hijos, 20 y 27, a negros. [**0.5 pts.**]

Como el padre de 25 también es rojo (el nodo con clave 30), aplicamos el caso 2: hacemos una rotación sim-ple a la derecha de la arista 35–30 —con lo que queda 30 como raíz (roja), con hijos 25 y 35 negros— e intercambiamos colores entre padre e hijos, dejando a 30 negro, y a 25 y 35 ambos rojos. [**1.5 pt.**]

Ahora el nodo con clave 25 tiene ambos hijos —con claves 20 y 27— negros; y como 20 es una hoja, inserta-mos allí el nuevo nodo con clave 22, que pintamos de rojo. [**0.5 pts.**]

b) Determina un orden en que hay que insertar las claves 1, 3, 5, 8, 13, 18, 19 y 24 en un árbol 2-3 inicialmente vacío para que el resultado sea un árbol de altura 1, es decir, una raíz y sus hijos.

Respuesta

Un árbol 2-3 con una raíz y sus hijos tiene a lo más 4 nodos y puede almacenar a lo más 8 claves (dos claves por nodo). Como son exactamente 8 claves las que queremos almacenar, éstas tiene que quedar almacena-das de la siguiente manera: la raíz tiene las claves 5 y 18; el hijo izquierdo, 1 y 3; el hijo del medio, 8 y 13; y el hijo derecho, 19 y 24. [**1 pt.**]

Para lograr esta configuración final hay varias posibilidades; aquí vamos a ver una. [**2 pts.**]

Podemos insertar primero las claves 1, 5 y 13, en cualquier orden, con lo cual queda 5 en la raíz, y 1 y 13 como hijos izquierdo y derecho. (En vez de 1 puede ser 3 y en vez de 13 puede ser 8. Por otra parte, en vez de empezar con 1, 5 y 13, es decir, empezar "por la izquierda", podríamos empezar por la derecha con 8-13, 18 y 19-24.) A partir de ahora, podemos insertar 3 en cualquier momento.

Ahora tenemos que conseguir que 18 quede en la raíz, junto con 5. Insertamos 18, que va a acompañar a 13, y a continuación insertamos 24, que va al mismo nodo de 13 y 18; como este nodo tiene ahora tres claves —13, 18 y 24 (lo que no puede ser)— lo separamos en dos nodos con las claves 13 y 24, respectivamente, y subimos la clave 18. Ahora podemos insertar 8 y 19, en cualquier orden.

2. Tablas de *hash*

Explica cómo manejar una tabla de hash si los elementos se almacenan dentro de la tabla (recuerda que puede haber colisiones), y además mantenemos una lista ligada de todos los casilleros vacíos. Su-ponemos que cada casillero de la tabla puede almacenar un *flag* (un bit 0 o 1) y, ya sea, un elemento más un puntero, o bien dos punteros. El objetivo es que todas las operaciones de diccionario (inser-ción, búsqueda y eliminación), así como las operaciones sobre la lista ligada, puedan ser ejecutadas en tiempo esperado O(1). ¿Es necesario que la lista ligada sea doblemente ligada?

Respuesta

Usamos el *flag* para indicar si el casillero tiene un puntero y un elemento (vale 0, cuando forma parte de una lista de elementos insertados que tienen el mismo valor de hash, similar a hashing con encadena-miento), o dos punteros (vale 1, cuando forma parte de la lista **doblemente ligada** de casilleros vacíos). Los punteros son simplemente índices de casilleros en la tabla. Sea L la lista de casilleros vacíos, y sean *prev* y *next* los dos punteros de cada casillero vacío. [**0.5 pts.**]

Sea x el dato que nos interesa y sea k su valor de hash.

Inserción. Si el casillero k está vacío (su *flag* vale 1, por lo que forma parte de L), entonces lo sacamos de L —en tiempo O(1), ya que L es doblemente ligada— y almacenamos ahí x, p.ej., en el campo del puntero *prev*. Además, ponemos el *flag* del casillero en 0 y el puntero *next* en *null*. El casillero k pasa así a ser el primer elemento (y por ahora el único) de una lista de elementos que tienen valor de hash k. [**1 pt.**]

Si, por el contrario, el casillero k está ocupado (su *flag* vale 0), entonces hay dos posibilidades:

- Es el primer elemento de una lista L_k de elementos insertados que tienen valor de hash k ; entonces saca-mos un casillero (p.ej., el primero) de L, almacenamos x en este casillero (en el campo *prev*), y agregamos el casillero a la lista L_k (p.ej., como segundo elemento). Todas estas operaciones toman tiempo O(1). [**0.5 pts.**]

- Es un elemento de una lista de elementos insertados que tienen valor de hash $k' \neq k$; entonces sacamos el casillero k de esta lista (por supuesto, lo reemplazamos por un casillero de L), y lo convertimos en el primer casillero de la lista de elementos que tienen valor de hash k. Nuevamente, todas estas operaciones toman tiempo O(1). [**0.5 pts.**]

Búsqueda. Si el casillero k no tiene a x, entonces seguimos la cadena de punteros *next* que empieza aquí. Al hacer esta búsqueda, conviene guardar el puntero al elemento anterior al que estamos mirando (para el caso de eliminación). Esta operación **no** toma tiempo O(1). Para la búsqueda, vale el mismo argumento visto en clase para hashing con encadenamiento. [**2 pt.**]

Eliminación. Supongamos que al hacer la búsqueda anterior, en la lista L_k que empieza en el casillero k, encontramos a x en el casillero q, que el casillero anterior a q en L_k es el casillero p, y que el casillero siguiente es r. Entonces sacamos el casillero q de L_k, lo ponemos (de vuelta) en L, y actualizamos L_k (haciendo que el casillero p apunte al casillero r). [**1.5 pts**]

3. Búsqueda en profundidad (DFS)

La siguiente en una versión más general del algoritmo visto en clases para recorrer un grafo G a partir de un vértice v:

```
DFS(G, v):
    marcar v
    S ß v              —inicializa un stack de vértices, S, con v
    while S no está vacío :
            w ß S      —extrae el vértice en el top de S y asígnalo a w
            "visitar" w
            for all x tal que (w, x) es una arista de G :
            if x no está marcado:
            marcar x
            S ß x
```

a) Supongamos que *G* es no direccional (las aristas no tienen dirección). Decimos que *G* es **biconec-tado** si no hay ningún vértice que al ser sacado de *G* desconecta el resto del grafo. Por el contrario, si *G* no es biconectado, entonces los vértices que al ser sacados de *G* desconectan el grafo se conocen co-mo **puntos de articulación**.

Describe un algoritmo eficiente para encontrar todos los puntos de articulación en un grafo conectado. Explica cuál es la complejidad de tu algoritmo.

Respuesta.

Para obtener un algoritmo eficiente que resuelve este problema, utilizamos una versión adaptada de DFS que pueda detectar aristas hacia atrás en el grafo no direccional. El principio sobre el que se basa la solución es que luego de hacer DFS en un grafo no dirigido hay dos posibilidades para que un vértice sea punto de articulación:

1. La raíz del árbol DFS generado es un punto de articulación: esto ocurre cuando la raíz tiene más de un hijo, pues significa que la única forma de descubrir todos estos hijos, fue partir una nueva rama de la búsqueda desde la raíz. Si no fuera así y la raíz no es punto de articulación, entonces al sacarla los hijos deberían tener otro camino que los una. *Este será el primer chequeo: verificar si la raíz tiene más de un hijo en el árbol DFS.* **[1 pt. Por caso raíz]**
2. Un vértice que no es raíz es punto de articulación: esto ocurre si para un nodo *u* todos sus descendientes no cuentan con un camino que llegue a un antecesor de *u* en el árbol DFS. Esto se puede verificar en tiempo constante si se agrega un atributo a cada nodo que se define según

   $$u.low = min\{u.disc,\ w.low \text{ para algún } w \text{ descendiente de } u\}$$

   Este atributo guarda el tiempo de descubrimiento de *u* o el de un ancestro alcanzable con alguna arista hacia atrás. Con esta estrategia, los puntos de articulación (no raíz) son aquellos que tienen el atributo *low* menor que el de sus descendientes directos. **[1 pt. Por estrategia general]**

Con esto, el algoritmo DFS que incorpora el atributo *low* a cada vértice seguido de una revisión para cada vértice permite entregar los puntos de articulación de todo el grafo. En resumen:

1. Ejecutar DFS que registra tiempos de descubrimiento y *low* (paso completo en $O(V+E)$)
2. Revisar cada vértice en el árbol DFS generado y comparar *low* con sus descendientes. Agregar a una lista aquellos vértices que no cumplan la condición buscada (paso completo en $O(V+E)$).
3. Revisar cuántos hijos tiene la raíz del árbol y agregarla si tiene más de uno (paso completo en $O(1)$).

Por lo tanto, la complejidad del algoritmo es $O(V+E)$ para un grafo representado con listas de adyacencia.

[1 pt. Por explicación de complejidad]

b) Considera el siguiente grafo *G* direccional (las aristas tienen dirección), representado mediante sus listas de adyacencias:

[0]: 5–1	[1]:	[2]: 0–3	[3]: 5–2	[4]: 3–2	[5]: 4	
[6]: 9–4–0	[7]: 6–8	[8]: 7–9	[9]: 11–10	[10]: 12	[11]: 4–12	[12]: 9

Ejecuta el algoritmo DFS anterior a partir del vértice $v = 0$. Muestra el orden en que los vértices van siendo marcados y el contenido del stack *S* cada vez que cambia.

Respuesta (posible solución).

Partiendo de 0, lo marcamos y lo ponemos en *S*; *S*= [0].

[A partir de ahora, cada paso = **0.5 pts.**]

Al sacar 0 (*S*= [ ]) y asignarlo a *w*, vemos que sus vecinos son 5 y 1; los marcamos y los ponemos en *S*; *S*= [5, 1].
Al sacar 5 (*S*= [1]) y asignarlo a *w*, vemos que su único vecino es 4; lo marcamos y lo ponemos en *S*; *S*= [4, 1].
Al sacar 4 (*S*= [1]) y asignarlo a *w*, vemos que sus vecinos son 3 y 2; los marcamos y los ponemos en *S*; *S*= [3, 2, 1].
Al sacar 3 (*S*= [2, 1]) y asignarlo a *w*, vemos que sus vecinos son 5 y 2, ambos ya marcados; *S*= [2, 1].
Al sacar 2 (*S*= [1]) y asignarlo a *w*, vemos que sus vecinos son 0 y 3, ambos ya marcados; *S*= [1].
Al sacar 1 (*S*= [ ]) y asignarlo a *w*, vemos que no tiene vecinos; *S*= [ ] y terminamos.

4. Dos grafos y se dicen *isomorfos* si existe una función biyectiva tal que si y sólo si . Escribe un algoritmo que determine si dos grafos son isomorfos. Considera que si dos grafos son isomorfos, al eliminar un vértice *x* de G_1 (y las aristas de *x*) y su vértice correspondiente (y aristas) en G_2, los grafos que quedan también son isomorfos.

Respuesta

F = diccionario inicialmente vacío

```
backtracking(Grafo1, Grafo2):
        // Caso base 1 pto
        If (grafo1.nodos = grafo2.nodos = vacio) return true
        // Asignar los nodos de un grafo los nodos del otro 2 pts
        n = grafo1.nodos.pop(0)
        For nodo in grafo2.nodos:
                F(n) = nodo
                // Restriccion 2 pts
                If  (cumple_restriccion(n, nodo))
                        // Hacer bien caso recursivo y UNDO 1pto
                        If  (backtracking(grafo1 sin n, grafo2 sin nodo) return true
                F(n) = null
        Return false

cumple_restricciones(nodo1, nodo2):
        If (|nodo1.vecinos| != |nodo2.vecinos|) return false
        For v in nodo1.vecinos:
                If (v esta en F):
                        If (F(v) esta en nodo2.vecinos):
                                Return false
        Return true
```