Solución
Primer Parcial de Programación 3
1° de octubre de 2014

Nota previa: los logaritmos utilizados a lo largo de esta solución son en base 2.

Ejercicio 1 (12 puntos)

El problema a resolver es la inserción en un heap. Un heap conceptualmente es un árbol binario casi completo donde se cumple que cada elemento es mayor (menor) que sus hijos; el árbol es completo o tiene incompleto solamente su último nivel. Al insertar se coloca el nuevo nodo de tal forma que en el árbol vaya completando el último nivel de izquierda a derecha.

Al insertar un elemento se utiliza la siguiente estrategia:

- se agrega el nuevo elemento al final
- se verifica la condición de orden del heap (entre un nodo y sus hijos) para el nuevo elemento colocándolo donde le corresponda (es necesario comparar con los elementos que son sus ancestros en el árbol para ubicar su lugar)
- al terminar el algoritmo se devuelve un heap (que cumple todas las condiciones mencionadas)

En la implementación propuesta este árbol se implementa en un arreglo: el lugar `i` representa un nodo del árbol y sus hijos están en las posiciones `2i` y `2i + 1` (recíprocamente el padre del nodo `i` está en `i div 2`). La raíz está en el lugar 1 (el 0 no se utiliza en el caso de `C`). El heap tiene así una marca de último elemento en el arreglo para indicar su final.

a) La operación básica es la comparación entre elementos del heap (`h[i]` $<$ `h [i div 2]`) ya que es una operación imprescindible para mantener la relación entre los elementos.

   Alternativa: el swap puede considerarse de la misma forma ya que es la acción (condicionada a la comparación) que permite mantener la relación entre los elementos. Se acepta contar esta operación como atómica o compuesta de varias asignaciones. Se debían contar sólo la cantidad de operaciones básicas, no era necesario tener en cuenta costos.

b) El mejor caso se da cuando el elemento a insertar es mayor o igual a su padre. Es decir, el elemento se inserta al final del heap y no se mueve de dicha posición.

   El peor caso se da cuando el elemento insertado es menor que la raíz del heap y, por lo tanto, menor que todos los elementos del heap.

c) En el mejor caso solo puede pasar lo siguiente:

$$T_b = 1 \text{ si el largo del heap es mayor a 0}$$

   En el peor caso se debe sumar todas las comparaciones que se debe realizar. Para ese caso se debe comparar con los ancestros de todos los niveles superiores. Por lo tanto la expresión que resume este conteo es:

$$T_w = \sum_{i=1}^{\lfloor log(k) \rfloor} c,$$

siendo c el costo de realizar la operación de comparación y k la cantidad de elementos del heap. Por lo tanto el costo en el peor caso es $\lfloor log(k) \rfloor c$.

d) I) **VERDADERO**

En primer lugar citemos la definición de $O(f)$:

$$O(f) = \{g : N \to R^*/\exists c \in R^+, \exists n_0 \in N, \forall n \in N, n > n_0, g(n) \leq c.f(n)\}$$

Por lo tanto lo que queremos demostrar es que la función constante 1 está incluida en este conjunto. Es decir, demostrar que:

$$1 \in \{g : N \to R^*/\exists c \in R^+, \exists n_0 \in N, \forall n \in N, n > n_0, g(n) \leq c.n\}$$

Tomando c y n_0 con valor 1 podemos ver que se cumple que $\forall n \in N, n > n_0, 1 \leq n$, demostrando que se cumple la afirmación.

II) **FALSO**

Una función está incluida en $\Theta(f)$ si y solo si esa función está incluida en $O(f)$ y $\Omega(f)$. Por la parte anterior sabemos que la función constante está en $O(f)$ por lo que para que esta afirmación se cumpla bastaría con demostrar que la función constante 1 está en $\Omega(f)$.

Citemos la definición de $\Omega(f)$:

$$\Omega(f) = \{g : N \to R^*/\exists c \in R^+, \exists n_0 \in N, \forall n \in N, n > n_0, g(n) \geq c.f(n)\}$$

Por lo tanto deberíamos demostrar que:

$$1 \in \{g : N \to R^*/\exists c \in R^+, \exists n_0 \in N, \forall n \in N, n > n_0, g(n) \geq c.n\}$$

Esto no se cumple ya que no existe n_0 ni c a partir de los cuales se pueda cumplir la condición de desigualdad para la función constante 1.

III) **VERDADERO**

Sea h una función tal que $h \in O(f)$. Por hipótesis sabemos que $f \in O(g)$. Entonces, por propiedad transitiva demostrada en teórico sabemos que $h \in O(g)$.

Asimismo sea k una función tal que $k \in O(g)$. Por hipótesis sabemos que $g \in O(f)$. Entonces, por propiedad transitiva demostrada en teórico sabemos que $k \in O(f)$.

Con las dos demostraciones anteriores hemos demostrado que todos los elementos de un conjunto también están en el otro. Por lo tanto hemos demostrando que $O(f) = O(g)$.

Ejercicio 2 (18 puntos)

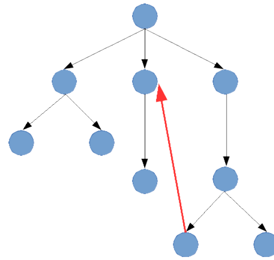
a) I) **FALSO**[1]

El resultado de una recorrida BFS no produce aristas forward.

Supongamos un nodo cualquiera de un grafo. Al momento de procesar sus adyacentes ellos pueden estar procesados o no procesados. En el último caso se generan aristas tree. En el otro las aristas que los relacionan son back o cross; nunca pueden ser forward porque, si las hubiera, estarían a distancia mayor a uno, lo cual es absurdo.

II) **FALSO**

Un contraejemplo posible sería considerar el siguiente grafo, el cual tiene en $A - A_T$ únicamente la arista marcada en rojo:



Se puede ver claramente que la arista roja es cross y apunta a un nodo que se encuentra dos niveles por encima del nodo origen.

III) **VERDADERO**

Si existieran aristas cross en distintos sentidos significa que estarían apuntando a nodos post-procesados y nodos que aún no habrían sido procesados al momento de procesar el nodo origen. Esto no tiene sentido ya que, para el segundo caso, el nodo debería ser procesado por el algoritmo y encontrarse en el subárbol que tiene como raíz el nodo origen.

IV) **FALSO**

En general no se cumple porque en BFS no se cumple el recíproco:

($\Rightarrow$) Si un grafo es acíclico implicaría que al momento de ejecutar tanto el algoritmo BFS como DFS no se podría visitar un nodo ya marcado. Por lo tanto no pueden haber aristas back.

($\Leftarrow$) La generalidad no se cumple ya que no se cumple para BFS. Un contraejemplo con el que podemos ver esto es el grafo definido de la siguiente forma:

$$G = (V, E)$$
$$V = \{a, b, c\}$$
$$E = \{a \rightarrow b, a \rightarrow c, b \rightarrow c, c \rightarrow b\}$$

Si consideramos una recorrida BFS que comienza en el vértice a el árbol que se forma es $\{(a, b), (a, c)\}$ y las aristas del ciclo son $\{(b, c), (c, b)\}$. Dichas aristas son ambas cross, lo que demuestra el contraejemplo.

b) En un paso genérico de la recorrida DFS, procesando un vértice v se visitarán a partir de él los adyacentes w que no estén marcados. De esa forma la arista (v, w) será una arista tree. Los w que estén marcados indicarán que la arista (v, w) estará en $A - A_T$. O sea que será back, forward o

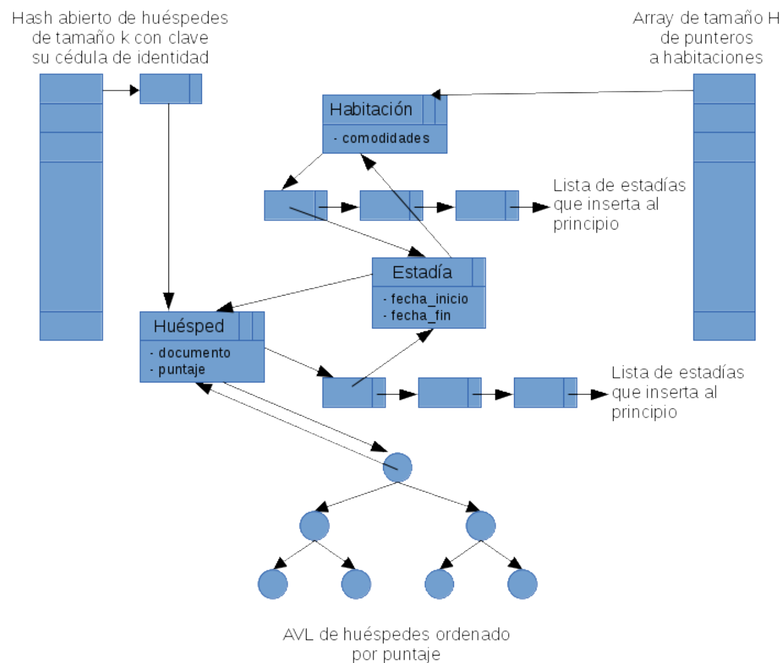
[1] Según la expresión de la proposición en lógica de primer orden se trata de una implicancia donde el antecedente es siempre vacío (no hay aristas forward). Esto hace que la proposición sea verdadera. Se aceptará la respuesta verdadera si la justificación se asemejara a lo anterior.

cross. La forma de diferenciar a las aristas back es usar el hecho de que éstas van de un vértice a un ancestro en el árbol de la recorrida. Los ancestros de v serán terminados de procesar después que este vértice, entonces basta con llevar una marca en postorden (para indicar cuando se termina de procesar el vértice) y chequear junto a que esté visitado si w **NO** tiene asignada una posmarca.

```
ListaArista aristasBack(Grafo g, Vertice v){

    ListaArista retorno;
    visitado[v] = true
    foreach (w in adyacentes(v)) {
        if (visitado[w]) {
            if (!posmarca[w])
                Insertar(retorno,(v,w));
        }
        else
            Concat(retorno,aristasBack(g,w));
    }

    posmarca[v] = true;
    return retorno;

}
```

c) El algoritmo base es el mismo. w debe estar visitado, se debe controlar que se haya empezado a procesar después que v para que sea un descendiente suyo: alcanza con prenumerar los vértices y en ese caso si el valor de pre procesamiento de w es mayor que el de v entonces la arista es forward.

Ejercicio 3 (10 puntos)



NOTA: Debido a los órdenes solicitados en las operaciones, el AVL podía también ser representado como un ABB.

`void imprimirHabitacionesOcupadas(Hotel h)`
Se debe recorrer el arreglo completo de habitaciones (en $O(H)$) y comparar la fecha actual con el intervalo de inicio y fin de la primer estadía de la lista de estadías de la habitación consultada ($O(1)$ porque siempre es la primer estadía por como se insertan). Si la fecha se encuentra en el mismo entonces la habitación se encuentra ocupada y se imprimen los datos. En caso contrario no se imprime nada.

`void estadiasHuesped(Hotel h, int documento)`
Primero se debe encontrar el huésped (como los huéspedes se almacenan en un hash se realiza en $O(1)$) y luego se imprimen todas las estadías de la lista (al ser necesario recorrer toda la lista esto se realiza en $O(e)$).

`void estadiasHabitacion(Hotel h, int numero)`
Primero se accede a la habitación (al estar almacenada en un arreglo se accede por el índice en $O(1)$) y luego se imprimen todas las estadías de la lista (al ser necesario recorrer toda la lista esto se realiza en $O(e)$).

`void altaEstadia(Hotel h, int documento, int habitación, Date inicio, Date fin)`
Se debe buscar la habitación y al huésped (ambas operaciones se hacen en $O(1)$ promedio por acceder a los mismos a través de un array y un hash respectivamente). Luego se crea la estadía agregando las referencias a la habitación y huésped y se agrega al principio de cada una de las listas que almacenan la habitación y huésped (todo esto en $O(1)$).

`void sumarPuntaje(Hotel h, int documento, int puntos)`
Se accede al huésped a través del AVL y se borra la entrada en el AVL y se rebalancea el árbol (todo esto se hace en $O(log(k))$ promedio por propiedades de la estructura). Luego se suma *puntos* al huésped y

se vuelve a insertar en el AVL (esto también se realiza en $O(log(k))$).

`void rankingHuespedes(Hotel h)`
Se recorre el AVL de forma ordenada (depende de cómo se define el orden de los elementos del AVL si la recorrida es en orden u orden inverso) y se imprime el ranking de los huéspedes. Esta operación se realiza en $O(k)$ ya que se deben visitar todos los elementos del AVL.