Solución Primer Parcial de Programación 3 (30/09/2010)

Instituto de Computación, Facultad de Ingeniería

Ejercicio 1 (13 puntos)

a) (**3 puntos**) Describa brevemente como se podría resolver el problema.

Dadas las restricciones de un grafo bipartito completo se realiza una recorrida BFS que determina y verifica la posibilidad de definir las particiones.

Se considera un vértice inicial donde todos sus adyacentes deberían determinar una partición. El inicial y cualquiera no adyacente a éste, deberían determinar la otra. De este modo queda definida la cardinalidad de las particiones si fuese bipartito completo.

Luego se realiza la verificación recorriendo en BFS, donde cada vértice deberá cumplir que:

- Todos sus adyacentes pertenecen a la otra partición.
  (Cada adyacente se clasifica de ese modo en caso que aún no lo este, o en caso contrario, se verifica que cumpla)
- La cantidad de adyacentes coincide con el cardinal de la partición a la cual pertenecen.

De este modo se contempla el caso en que el grafo tenga más de una componente conexa ya que la cardinalidad de adyacentes de un vértice no coincidirá con alguna de las particiones.

b) (**10 puntos**) Implemente un algoritmo en C* que lo resuelva según a).

```
#include "stdafx.h"
#include "stdio.h"
#include <iostream>

using namespace std;

bool esBipartitoCompleto(Grafo g){
      int N = cantNodosGrafo(g);
      int procesado[N] = {0};
      int nivel[N] = {0};
      int n = 0; // Nodo de comienzo (puede ser cualquiera)
      nivel[n] = 0;

      // Determinación de cardinalidad de particiones: A y B.
      Lista ady = adyacentesNodoGrafo(g,n);
      int cantB = 0;
      while (!ListaEsVacia(ady)){
            listaSacar(ady);
            cantB++;
      }
      int cantA = N - cantB;
```

```
	// Recorrida BFS de verificación de bipartición y completitud
	Lista procesar = CrearLista();
	ListaInsertarUltimo(procesar,n);
	while (!ListaEsVacia(procesar)) {
		n = listaPrimero(procesar); listaSacar(procesar);
		ady = adyacentesNodoGrafo(g,n);
		procesado[n] = 1;
		int cantAdy = 0;
		while (!ListaEsVacia(ady)) { // Control de adyacentes
			int a = listaPrimero(ady); listaSacar(ady);
			cantAdy++;
			if (!procesado[a] && !listaPerteneceElem(procesar,a)){
				ListaInsertarUltimo(procesar,a);
				nivel[a] = (nivel[n]+1)%2;
			}
			else
				if (nivel[a] == nivel[n]) { // Control de
adyacentes en la misma partición

	listaDestruir(procesar);listaDestruir(ady);
					return false;
				}
		}
		// Verifica que G sea conexo y exista adyacencia con todos
los nodos de la otra partición (completo).
		if ((nivel[n]==0 && cantAdy!=cantB) || (nivel[n]==1 &&
cantAdy!=cantA))
			listaDestruir(procesar);listaDestruir(ady);
			return false;
	}
	return true;
}

int _tmain(int argc, _TCHAR* argv[])
{
	int a = 0;
	esBipartitoCompleto();
	return 0;
}
```

Ejercicio 2 (13 puntos)
a) (11 puntos)

Nota: pueden existir otras soluciones alternativas a la propuesta que también cumplan los órdenes de las operaciones.

Las operaciones restringen la estructura en la siguiente forma:

1. void imprimirInformacionMedico(string codigoMedico, Estructura E)

Esta operación debe ser resuelta en O(1) caso promedio, por lo que es necesario mantener un hash abierto de médicos, por código de médico, de tamaño M.
Caso promedio:

1. Encontrar el médico en el caso promedio es O(1) (a través del hash de médicos).
2. Imprimir la especialidad del médico es orden constante, ya que se accede a la información del médico a través de la referencia que se almacena en cada nodo del hash.

Por lo tanto, el orden de la operación en el caso promedio es O(1).

2. void imprimirInformacionPaciente(string ciPaciente, Estructura E)

Resulta necesario mantener un hash abierto de pacientes, por cédula, de tamaño P, para resolver la operación en el orden pedido.

Caso promedio:

a) Encontrar al paciente en el caso promedio es O(1) (a través del hash de pacientes).
b) Imprimir los datos del paciente es orden constante, ya que se accede a la información del paciente a través de la referencia que se almacena en cada nodo del hash.

Por lo tanto, el orden de la operación en el caso promedio es O(1).

3. *void imprimirMotivoConsulta(int día, int mes, string codigoMedico, string ciPaciente, Estructura E)*

Para resolver esta operación se utilizará una matriz de 12x31 de fechas (una fila por cada mes, y una columna por cada día). Cada celda de la matriz tendrá un puntero a un hash abierto de tamaño M, por código de médico. Cada entrada de este hash deberá tener un puntero a otro hash abierto de consultas de tamaño P (por cédula de paciente).

Caso promedio:

i. Encontrar la celda de la matriz correspondiente al día *día* y mes *mes* es orden constante.
   - Encontrar el valor correspondiente al médico *codigoMedico* en el hash es O(1) caso promedio.

   Encontrar la consulta en el hash por cédula de paciente es O(1) caso promedio.
- Imprimir el motivo de la consulta es orden constante, ya que se accede a la información de la consulta a través de la referencia que se almacena en cada

nodo del hash.

Por lo tanto, el orden de la operación en el caso promedio es O(1) promedio.

4. void imprimirPacientesAAtenderPorUnMedico(int día, int mes, string codigoMedico, Estructura E)

Esta operación debe ser resuelta en O(A) promedio, donde A es la cantidad de pacientes a atender en un día y un mes por un médico. Para ello, se deberá mantener en la información del médico en el nodo del hash referenciado a través de una celda de la matriz de fechas, una lista de consultas ordenada por orden de atención (se agrega al final).
Caso promedio:

- Encontrar la celda de la matriz correspondiente al día *día* y mes *mes* es orden constante .
- Encontrar el valor correspondiente al médico *codigoMedico* en el hash es O(1) caso promedio.
- Recorrer la lista de largo A de pacientes a atender por ese médico es O(A) caso promedio.
- Imprimir el nombre de cada paciente es orden constante, ya que se accede a la información del paciente a través de la referencia que se almacena en cada nodo de la lista.

Por lo tanto, la operación tiene O(A) caso promedio.

5. void imprimirMedicoConMasPacienteAAtender(int día, int mes, Estructura E)

Para resolver esta operación en O(1) peor caso, en la celda de la matriz de consultas se deberá mantener un puntero al médico con más consultas de ese día del mes, como también la cantidad de pacientes a atender. Además, en el hash (por código) apuntado por la celda, es necesario mantener la cantidad de pacientes que tiene para atender cada médico.

Peor caso:

- Encontrar la celda de la matriz correspondiente al día _día_ y mes _mes_ es orden constante.
- Imprimir el código y especialidad del médico con más consultas en esa fecha es orden constante, ya que se accede al médico a través de la referencia que se almacena en la celda de la matriz.

Por lo tanto, el orden de la operación en el peor caso es O(1).

6. *void reservarDiaParaUnMedico(int día, int mes, string codigoMedico, string ciPaciente, string motivoConsulta, Estructura &E)*

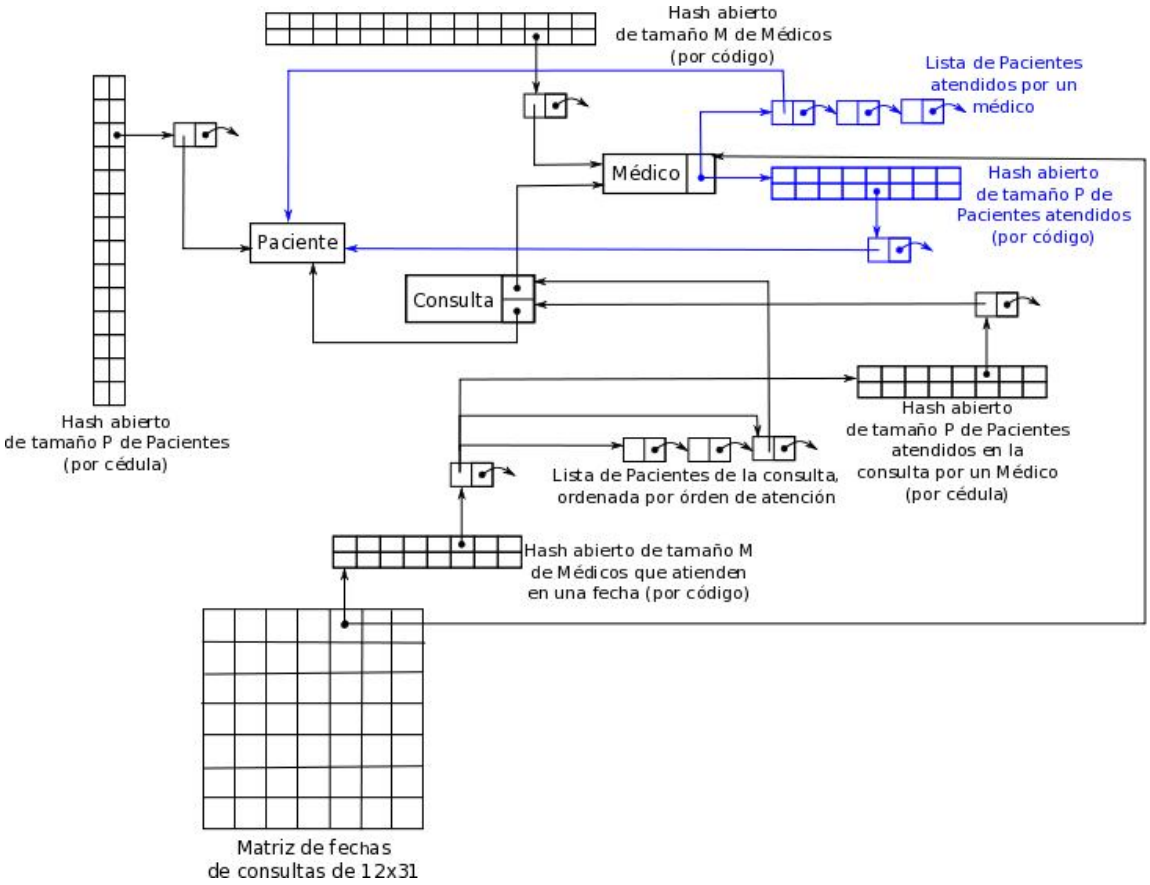
Caso promedio:

- Obtener la información del médico en el hash de médicos, dado el código, es O(1) caso promedio.
- Obtener la información del paciente en el hash de pacientes, dada su cédula es O(1) caso promedio.
- Encontrar la celda de la matriz correspondiente al día *día* y mes *mes* es O(1) orden constante.

- Encontrar el valor correspondiente al médico *codigoMedico* en el hash apuntado por esa celda es O(1) caso promedio.
- Si el médico no tiene consultas ese día, se agrega al hash, lo cual es O(1) caso promedio.
- Agregar al final de la lista de consultas del médico una nueva consulta con *motivoConsulta*, es orden constante a través del puntero al final de la lista.
- Agregar la nueva consulta al hash por cédula de paciente es O(1) caso promedio.
- Incrementar la cantidad de pacientes del médico para ese día en el hash de médicos por fecha es orden constante.
- Se compara este nuevo valor con el del médico que tiene más pacientes para este día. Si la cantidad de este médico es mayor estricto que la del máximo, se cambia el puntero al médico y la cantidad de pacientes de la celda. Todo esto es O(1) caso promedio.

Por lo tanto, esta operación es O(1) caso promedio.

La estructura propuesta se presenta en la siguiente figura (la parte azul corresponde a agregados para la parte b):



b) (2 puntos)

void imprimirPacientesDeUnMedico (string codigoMedico, Estructura E)

Para resolver esta operación, se debe agregar un hash de pacientes a la información del médico (pacientes atendidos alguna vez en el año por el médico) y una lista con inserción al principio con los pacientes que al menos una vez fueron atendidos por el médico. La lista no contiene elementos repetidos.

Peor caso:

- Buscar el médico en el hash de médicos de tamaño M es O(M) peor caso.
- Recorrer la lista de pacientes de este médico, imprimiendo el nombre de cada paciente es O(P) peor caso.

Por lo tanto, la operación en el peor caso es O(M+P).

La única operación que se ve afectada es la 6:

void reservarDiaParaUnMedico(int día, int mes, string codigoMedico, string ciPaciente, string motivoConsulta, Estructura &E)

Al agregar una nueva reserva, es necesario buscar al paciente en el hash del médico,

- si el paciente no pertenece al hash, entonces se agrega al mismo y a su vez se agrega a la lista de pacientes.
- si el paciente pertenece al hash, entonces no se hace nada.

Esto es O(1) caso promedio, por lo que el orden original de la operación no cambia.

Ejercicio 3 (14 puntos)

Parte 1) (7 puntos)

a) (3 puntos)

```
1 for (int i = 1; i <= n; i++){
2
3  for (int j = 1; j <= i * i; j++){
4
5        if (j % i == 0) {
6              bool valor = random();
7              if (valor) {
8                    calcular();
9              }
10       }
11 }
12}
```

La única línea del código a considerar para el costo es la 8.

Mejor Caso:
Como la probabilidad de que valor sea falso es 2/3, es posible que siempre sea falso y por lo tanto, que no se ejecute nunca la línea 8. Quedando un costo: $T_B(n) = 0$

Razonamiento para los casos Peor y Medio:
En la línea 1 se tiene una iteración en donde varía la variable *i*. Respecto al for de la línea 3, este va desde 1 hasta i^2. Puede observarse que hay exactamente *i* números múltiplos de *i* en ese intervalo. Por lo que se concluye que la línea 5 es verdadera exactamente *i* veces.

Peor Caso:
Como la probabilidad de que *valor* sea verdadero es distinta de cero y las invocaciones son independientes, es posible que siempre sea verdadero y por lo tanto, siempre se ejecute la línea 8. Son *i* veces, y cada una de estas tiene costo 1.

$$T_W(n) = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Caso Medio:
Una vez que la línea 5 es evaluada en verdadero, la probabilidad de que se ejecute la línea 8 es 1/3.

$$T_A(n) = \sum_{i=1}^{n} i\frac{1}{3} = \frac{n(n+1)}{2}\frac{1}{3}$$

b) (2 puntos)

i) La afirmación es falsa. Propiedades utilizadas:

- Si lím {n→+∞} (f(n) / g(n)) = 0 entonces f(n) ∈ O(g(n)) pero g(n)∉O(f(n));
- *f(n) ∈ O(g(n))* sii *g(n)* ∈ $\Omega(f(n))$
- $\Theta(f) = O(f) \cap \Omega(f)$

$$\lim_{n\to+\infty} \frac{n!}{(2n)!} = \lim_{n\to+\infty} \frac{n!}{(2n)(2n-1)...(n+1)n!} = \lim_{n\to+\infty} \frac{1}{(2n)(2n-1)...(n+1)} = 0$$

ii) La afirmación es verdadera. Propiedades utilizadas:

-Si lím {n→+∞} (f(n) / g(n)) ∈ R+ entonces f(n) ∈ O(g(n)) y g(n) ∈ O(f(n))

- *f(n) ∈ O(g(n))* sii *g(n)* ∈ $\Omega(f(n))$
- $\Theta(f) = O(f) \cap \Omega(f)$

$$\lim_{n\to+\infty} \frac{n!+3n^2}{n!} = \lim_{n\to+\infty} \frac{n!}{n!} + \frac{3n^2}{n!} = 1 + 0 = 1$$

c) (2 puntos)

La afirmación es falsa. Recordamos la definición de O

$$f(n) \in O(g(n)) \text{ sii } (\exists K \in R^+)(\exists n_0 \in N)(\forall n \in N)\left(n \geq n_0 \rightarrow f(n) \leq K * g(n)\right)$$

Contraejemplo:

Si $f(n) = n$ se cumple que $f \in O(n)$, con $n_0 = 0, K = 1$. Ahora se va a probar que $(n^2 + n) \notin O(n)$.

Suponiendo por el absurdo que $(n^2 + n) \in O(n)$ se va a llegar a una contradicción.

$(n^2 + n) \in O(n)$

Sii (definición O)

$(\exists K \in R^+)(\exists n_0 \in N)(\forall n \in N)\left(n \geq n_0 \rightarrow n^2 + n \leq K * n\right)$

Sii (operando)

$(\exists K \in R^+)(\exists n_0 \in N)(\forall n \in N)\left(n \geq n_0 \rightarrow n \ \leq (K-1)\right)$

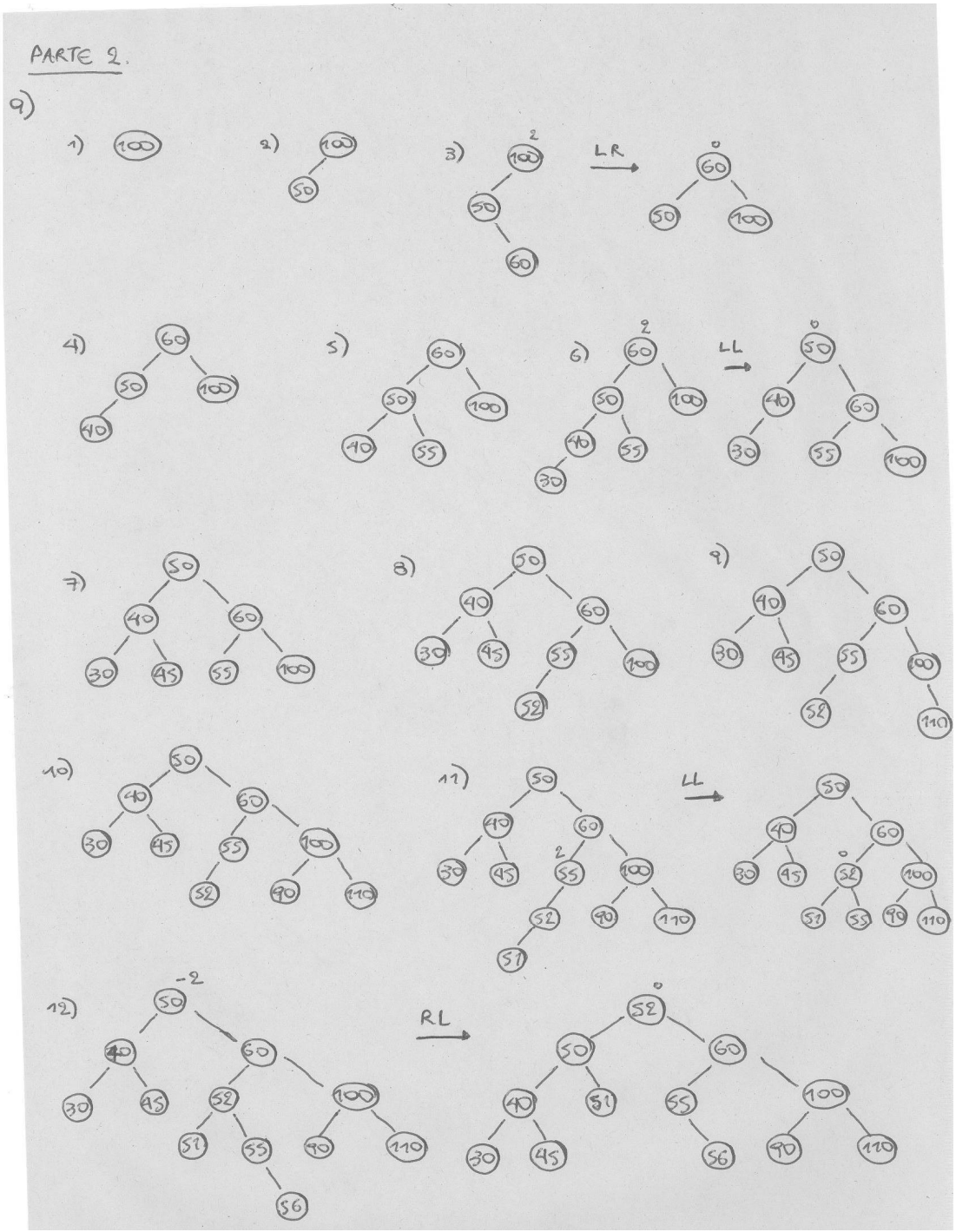
La última proposición es falsa debido a que afirma que los naturales están acotados superiormente. Por lo tanto, fue erróneo suponer que $(n^2 + n) \in O(n)$. Luego, se concluye que $(n^2 + n) \notin O(n)$. Finalmente se llega que es erróneo afirmar que $(f^2 + n) \in O(2n)$.

Nota: muchos estudiantes intentaron probar que para toda *f* genérica que satisfaga $f \in O(n)$ es falsa la afirmación. El problema de ello es que algunas funciones *f* lo cumplen como $f = n^{1/2}$ y otras no como la que se utilizo como contra ejemplo. Como

sucede lo anterior es imposible realizar una demostración sin instanciar *f* bajo ciertas restricciones.

Parte 2) (7 puntos)

a) (4 puntos)



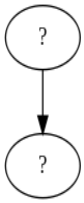
b) (3 puntos)

El concepto de “peor árbol AVL de altura *h”* se refiere al AVL que para la altura *h* dada tenga la menor cantidad de nodos *n* posible.
Razonando de forma inversa, dada una cantidad de *n* elementos almacenados en un árbol binario de búsqueda cualquiera, lo que se busca es que este tenga la menor altura posible. O sea que el costo de búsqueda sea óptimo para el peor caso.

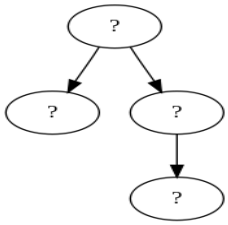
A continuación se muestra una construcción para un sub conjunto de los peores AVL de altura *h.*

h=0

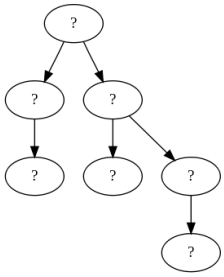
h=1



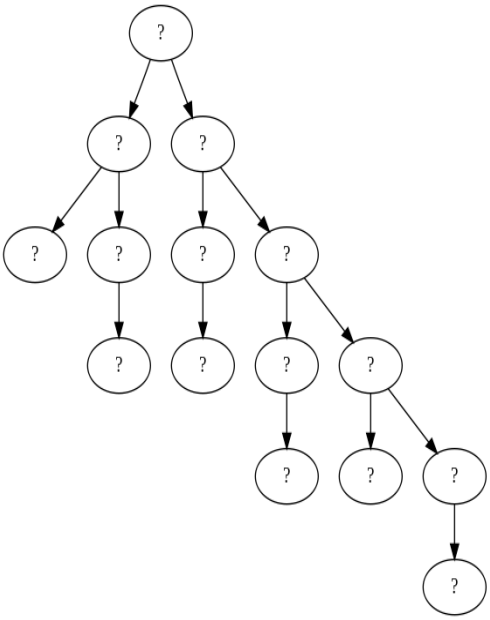
h=2



h=3



h=4



h=5