Facultad Politécnica - Departamento de Informática – Carrera: Ingeniería Informática
Algoritmos y Estructura de Datos III - Primer Examen Parcial – 2023/1er Periodo. Sección TQ/TR

Prof. Cristian Cappo/ Prof. Luis Moré Jueves, *14/09/2023* Puntos: 58 + 2 (orden, claridad, pulcritud) = **60**

EXAMEN SIN MATERIAL. APAGUE SU CELULAR. La hoja de examen se devuelve. La interpretación de los temas forma parte de la evaluación. TENGA A MANO HOJAS DE EXAMEN, CALCULADORA, LAPIZ, BORRADOR Y BOLÍGRAFO. Duración 150 minutos.

Nombre y Apellido: _____ Nro. CIC:_____ Sección:___

Posibles soluciones

Tema 1 (15p)
Por cada expresión indique si es *Verdadero* o *Falso,* fundamentando la respuesta.

a) $n^3 \in O(n^4)$

Si. porque $n^3 \leq n^4 \, para \, n \geq 1$. $f(n) \in O(g(n))$ significa que hay una constante c tal que $f(n) \leq cg(n)$. Entonces, una función estrictamente menor está en O de una mayor.

b) $n^3 \in \theta(n^4)$

No. Esto no es cierto, porque para que dos funciones sean Θ una de otra, tienen que crecer aproximadamente con el mismo ratio. En particular, como $n^3 < cn^4$ para cualquier $n > 1/c$, no se puede dar que $n^3 \geq cn^4$ para alguna $c > 0$, para alguna n lo suficientemente grande.

c) $2^{2n} \in O(2^n)$

No, $2^{2n} = 2^n 2^n > c2^n$ para cualquier $n > log\, c$. Entonces 2^{2n} no es menor a $c2^n$ para cualquier $c > 0$ y una n lo suficientemente grande.
b

d) $\sum_{i=1}^{log\, n} 4^i \in \theta(n^2)$

Si. $\sum_{i=1}^{logn} 4^i = (4^{log(n+1)} - 12)/3) = 4/3n^2 - 4 \in \Theta(n^2)$, usando la fórmula de la suma de series geométricas.

e) $\sum_{i=1}^{n} f(i) = f(1) + f(2) + .. f(n) \in \theta(f(n))$, siendo *f* una función creciente positiva.

No, sea $f(n) = n$. Entonces, $\sum_{i=1}^{n} f(i) = \sum_{i=1}^{n} i = n(n-1)/2 \notin \Theta(n)$

Criterio de corrección por cada ítem: Indicó correctamente si fue Verdadero o Falso (1p). Fundamenta correctamente (2p)

Tema 2 (13p)

Diseñe una estructura de datos en Java que soporte agregar elementos comparables (o borrarlos) de dos listas desordenadas con posibilidad de tener duplicados y verificar si algún elemento aparece en ambas listas. Los elementos son iguales de acuerdo a la interface *Comparable*.

La especificación y el funcionamiento se muestra abajo:

```
Especificación de la Estructura de dato
public class BiLista<E..> // Definición
-------------------------------------------
BiLista()                // Crea la ED

void agregarLista1(E x)  // Agrega x a la Lista 1
void agregarLista2(E x)  // Agrega x a la Lista 2

void borrarLista1(E x )  // Borra x de la Lista 1
void borrarLista2(E x)   // Borra x de la Lista 2

boolean existeDupLista() // Retorna true si existe
el
                         // mismo elemento
                         // en ambas listas
```

```
// Ejemplo de uso
BiLista<Integer> bl = new BiLista<Integer>();// [] []

bl.agregarLista1(1);     // [1] []
bl.agregarLista1(45);    // [1,45] []
bl.agregarLista2(56);    // [1,45] [56]
bl.agregarLista2(56);    // [1,45] [56,56]
bl.existeDupLista();     // false
bl.agregarLista2(45);    // [1,45] [56,56,45]
bl.existeDupLista();     // true
bl.borrarLista2(56);     // [1,45] [56,45]
bl.borrarLista1(45);     // [1] [56,45]
bl.existeDupLista();     // false
```

La operación *existeDupLista* debe realizarse en tiempo constante *O(1)* en el peor caso mientras que las operaciones de *agregarLista#* y *borrarLista#* debe hacerse en *O(log n)* a lo sumo en el peor caso. Puede utilizar cualquiera de las estructuras de datos vistas en clase (listas, pilas, colas, BST, AVL o tablas de dispersión) y asumir que ya están implementadas en su forma estándar. **No utilizar ninguna colección del API de Java (esto invalidará la solución). Justificar los tiempos de su implementación.**

Criterio de corrección:

	Implementación correcta	**Funciona en el tiempo solicitado**	**Justificación de tiempo**
Estructura de datos	4p	-	-
agregarLista#	1p	1p	1p
borrarLista#	1p	1p	1p
existeDupLista	1p	1p	1p

Tema 3 (10p)

Considere el algoritmo *sonIsomorfos(x,y)* que recibe dos nodos que son raíces de árboles binarios y retorna *Verdadero* si los árboles **Tx** y **Ty** son isomorfos. Dos árboles **Tx** y **Ty** son isomorfos si existe una función *f*, uno a uno sobre el conjunto de nodos de **Tx**, en el conjunto de vértices de **Ty,** tal que los vértices v_i y v_j son adyacentes en **Tx** si $f(v_i)$ y $f(v_j)$ son adyacentes en **Ty**. Así, es posible corresponder cada nodo en **Tx** a cada nodo en **Ty** satisfaciendo la adyacencia, independientemente del contenido de los nodos.

Considere que *tamanho(x)* retorna el tamaño del subárbol con raíz **x**. Asuma que la implementación contiene un dato extra con el tamaño de cada subárbol. El *tamanho(null)* es 0. Así *tamanho(r)* retorna la cantidad de nodos del árbol con raíz ***r***.

```
public static boolean sonIsomorfos( Nodo x, Nodo y ){
      if ( tamanho(x) !=  tamanho(y) )
            return false;
      if ( x == null )
            return true;
      if ( ( sonIsomorfos(x.izq, y.izq) && sonIsomorfos(x.der, y.der))  ||
           ( sonIsomorfos(x.der, y.izq) && sonIsomorfos(x.izq, y.der)) )
            return true;
      return false;
}
```

Se pide:

a) Dar dos ejemplos de árboles binarios isomorfos con al menos 5 nodos (3p)
b) Aplicar el algoritmo dado a uno de los árboles en a) (3p)
c) Asumiendo que *tamanho(x.izq) == tamanho(x.der)* en cada nodo interno **x** calcule el tiempo de ejecución *T(n)* de este algoritmo y la cota O. (4p)

Criterio de corrección:

Parte a	Un ejemplo correcto – 1.5p	Dos ejemplos correctos – 3p.
Parte b	Aplica con errores – 1p	Aplica sin errores -3p

Parte c	Obtiene correctamente la T(n) – 2p	Calcula Correctamente la cota O – 2p

parte c) El algoritmo hace cuatro llamadas recursivas. A partir de los supuestos dados para el árbol balanceado, cada una de las cuatro llamadas se hacen a un árbol de la mitad de tamaño. La parte no recursiva del algoritmo tiene coste constante. Entonces, se tiene la recurrencia $T(n) = 4T(n/2) + O(1)$. Ésto se ajusta al teorema maestro, siendo $a = 4 > 1 = b^k$, y el tiempo total es $O(n^{log_2 4}) = O(n^2)$.

Tema 4 (15p)

Suponga que tiene una lista de **n** números y debe imprimirlos de manera ordenada. Además tiene acceso a un árbol rojinegro que soporta las operaciones de búsqueda, inserción, borrado, mínimo, máximo, sucesor y predecesor en tiempo *O(log n)*. Usted debe imprimir la lista ordenada utilizando las siguientes estrategias:

a) Solo la operación de inserción y un tipo de recorrido en el árbol.
b) Solo las operaciones de mínimo, sucesor e inserción.
c) Solo las operaciones de mínimo, inserción y borrado.

Escriba un algoritmo (puede ser pseudocódigo) que imprima la lista de **n** números de manera ordenada en un tiempo máximo de *O(n log n)* por cada estrategia mencionada. Las operaciones mencionadas ya están disponibles al igual que los recorridos (no hace falta implementar). Fundamente el tiempo de cada algoritmo.

Criterio de corrección:

Parte a	Muestra un ejemplo correcto – 1.5p	Muestra dos ejemplos correctos – 3p.
Parte b	Aplica con errores – 1p	Aplica sin errores – 3p
Parte c	Calcula correctamente la T(n) – 2p	Calcula correctamente la cota O – 2p

Cualquier algoritmo para ordenar elementos usando un BST debe iniciar construyendo el árbol. Esto involucra la inicialización del árbol (básicamente poner el puntero t a NULL), y luego leer/insertar cada uno de los n items en t. Ésto tiene costo O(n log n), porque cada inserción cuesta a lo sumo tiempo O(log n). Curiosamente, solamente construir la estructura de datos es un paso que limita la tasa para cada uno de los algoritmos de ordenamiento.

El primer problema nos permite hacer inserción y recorrido inorden. Podemos construir un árbol de búsqueda insertando los n elementos, y luego hacer un recorrido para acceder a los items de manera ordenada.

El segundo problema permite usar las operaciones de sucesor y mínimo luego de construir el árbol. Podemos comenzar por elemento mínimo, y luego encontrar repetidamente el sucesor para recorrer los elementos de manera ordenada.
El tercer problema no nos permite usar sucesor, pero nos permite borrar. Podemos encontrar repetidamente y borrar el mínimo elemento para recorrer nuevamente los elementos de manera ordenada.

Las soluciones a los tres problemas son:

```
sortA()
    inicializar-arbol(t)
    Mientras (hayan datos a insertar)
    read(x);
    insert(x,t);
recorrido-inorden(t);
```

```
sortB()
    inicializar-arbol(t)
    Mientras (hayan datos a
insertar)
    read(x);
    insert(x,t);
y=Minimo(t);
mientras (y =/= NULL)
    print (y->clave)
    y=Successor(y,t)
```

```
sortC()
    inicializar-arbol(t)
    Mientras (hayan datos a
insertar)
    read(x);
    insert(x,t);
y=Minimo(t);
mientras (y =/= NULL)
    print (y->clave)
    Delete(y,t)
    y=Minimum(t)
```

Tema 5 (5p)

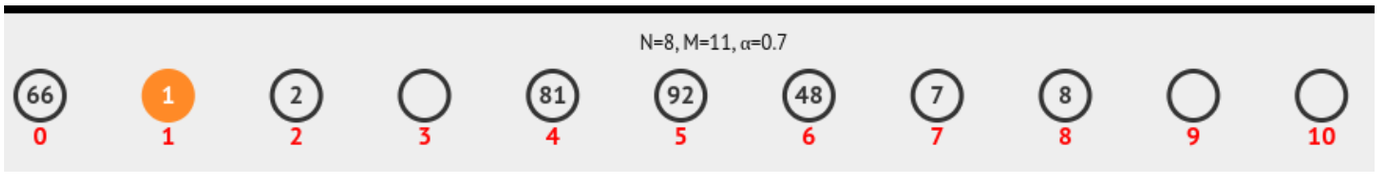
Dada una Tabla de dispersión inicialmente vacía con las siguientes características:

- Resolución de colisiones cerrada con exploración lineal.
- M=11, $\lambda = 0,7$, h(k) = (i + k%M)%M

- La estrategia a utilizar en caso de superar el umbral α es rechazar nuevas inserciones.

Realice el proceso de inserción de los siguientes elementos: 81,92,7,8,66,2,1,48,12,13. Indique los elementos insertados y rechazados.
Calcule T(N) en el mejor y peor caso en términos de inserciones y rechazos en una tabla como la descrita anteriormente. Tenga en cuenta para los fines del cálculo que se buscan insertar N elementos, y que N=M.
Criterio de corrección: Realiza las inserciones correctamente según características 2p - T(N) correcta mejor caso, con fundamentación– 1p. T(N) correcta peor caso, con fundamentación – 2p.

Dadas las condiciones del problema, los elementos que pueden insertarse quedan de la siguiente manera (se puede verificar en visualgo.net):



Insertar 12 implica que $\lambda = 0,8$, por lo que ya debe ser rechazado, así como 13.

Para calcular T(N), se debe tener en cuenta que se insertan N elementos a partir de cero. El mejor caso se daría cuando nunca ocurren colisiones, por lo que la tabla se llenaria de la siguiente forma aproximadamente:

n_1	n_2	n_3	…	…	…	…	…	n_λ		

esto implica: cantidad de inserciones correctas: 0,7N (cada inserción es O(1)) y rechazos 0,3N (cada rechazo es O(1)). entonces T(N)= 0,7N + 0,3N = N.

En el peor caso, todas las inserciones se harían en el mismo slot. Esto se daría cuando los datos vienen de tal manera que h(k) siempre mapea al mismo slot, por lo que se daría algo como esto (asumiendo sin perder generalidad que se intenta insertar siempre en el primer slot):

n_1 (1op)	n_2 (2op)	n_3 $(3op)$	…	…	…	…	…	n_λ(0,7n op)		

La cantidad de operaciones para realizar las inserciones en este caso es 1+2+3+...+0,7N , y 0,3N para rechazos. entonces se tiene:

T(N) = $\frac{0,7N}{2}(1 + 0,7N) + 0,3N$

Para N=10 sería 28 + 3.