IIC2133

I1

23 sept. 2025

Tiempo: 2 horas
Se deben **entregar sólo 3** de las 4 preguntas.

La entrega es a través de Canvas. Al final de la prueba tienen 10 minutos para subir la imagen de la prueba: en formato vertical, cada pregunta por separado. Para esto, se permite el uso de UN dispositivo (teléfono, tablet), el cual debe estar guardado y en silencio durante la prueba.

Torpedo: Una hoja tamaño oficio escrita a mano.

Cada pregunta tiene 6 puntos (+1 punto base). La nota final de la interrogación es el promedio de las 3 preguntas entregadas.

Si necesita salir ir al baño, debe dejar el celular adelante en la mesa con los ayudantes.

Lea todas las preguntas antes de iniciar las respuestas.

1) En clase vimos que la base de *Quicksort* es la buena elección de un pivote para la función *Partition* y usamos un pivote elegido de forma aleatoria. Sin embargo, existen otras maneras de elegirlo.

Una de ellas es reemplazar la función *Partition* por *MedianPartition*. En la función *MedianPartition* el pivote se elige con la función *Median*(A, i, f), que entrega en tiempo O(n) el índice en donde se ubica la mediana del arreglo $A[i..f]$.

a) [1 pt.] Modifique los pseudocódigos de *Quicksort* y *Partition* de modo de adaptarlos a este nuevo escenario (*MedianPartition*) y explique porqué *MedianPartition* es O(n).

b) [2 pts.] Argumente en qué casos el uso de *MedianPartition* podría mejorar el tiempo de ejecución del peor caso de *Quicksort*.

c) [3 pts.] Otra posible mejora de *QuickSort* es cambiar a *InsertionSort* para sub-secuencias pequeñas (≤ 20 elementos). Modifique el pseudocódigo de *Quicksort* para implementar esta mejora.

2) Árboles 2-3

a) [1 pt.] **Inserte** los siguientes elementos en un árbol 2-3, *A*, inicialmente vacío: *X*, *D*, *Y*, *R*, *W*, *M*, *C*, *G*. Represente gráficamente cada paso y las operaciones relevantes para mantener la propiedad de árbol 2-3 luego de cada inserción.

b) [2 pts.] Proponga el pseudo código para un **algoritmo de búsqueda** en un árbol 2-3. El algoritmo recibe como parámetros un puntero al árbol donde buscar, un puntero al padre del árbol dónde buscar, y la llave a buscar. El algoritmo debe retornar punteros al nodo encontrado (vacío si no se encuentra) y a su padre.

c) [3 pts.] Proponga el pseudo código para un **algoritmo de inserción** en un árbol 2-3. El algoritmo recibe como parámetros un puntero al árbol donde insertar, un puntero al padre del árbol dónde insertar, y la llave a insertar. *Nota*: Debe detallar todas las funciones necesarias para la inserción, por ejemplo, *Split* de nodos y aumento de altura del árbol.

3) Al analizar una solución para ordenar un conjunto de datos por dos criterios (llave primaria "apellido" y llave secundaria "nombre"), de modo que el resultado esté ordenado por la llave primaria, y, para igual llave primaria, por la llave segundaria, se propone el siguiente algoritmo:

```
Ordena(A,n)
      QuickSortPri(A, 0, n-1) // Ordena el arreglo A, completo,
                              // por llave primaria
      i ← 0
      mientras i ≤ n - 2
            j ← i + 1
            primaria ← A[i].p
            mientras j ≤ n - 1 y A[j].p == primaria
                  j ← j + 1
            QuickSortSec(A, i, j - 1)     // Ordena A, entre i y j – 1,
                                          // por llave secundaria
            i ← j
```

a) [2 pts.] Justifique que el algoritmo propuesto es **correcto**.

b) [2 pts.] ¿Cuál es la **complejidad** de mejor caso y la **complejidad** de peor caso del algoritmo propuesto?

c) [2 pts.] Considere que existe el algoritmo de ordenamiento *BlockSort*( ) el cual es Estable, Correcto, In-place, con complejidad O(n), O(n log n), O(n log n) en mejor, promedio y peor caso, respectivamente.

Dado lo anterior, **escriba una modificación de** *Ordena*(A, n) que se beneficie de estas características (en particular, de la Estabilidad), asumiendo que Ud. dispone de funciones *BlockSortPri*(A, i, j) y *BlockSortSec*(A, i, j), que ordenan el arreglo A por llave primaria o secundaria, respectivamente. Argumente brevemente porqué su versión es correcta y cómo se compara en complejidad con la versión original.

4) Árboles binarios de búsqueda

a) Dado un **ABB** (no necesariamente balanceado), una forma —distinta de la estudiada en clases— de eliminar una clave cuyo nodo tiene ambos hijos es la siguiente: el nodo es ocupado por su (hijo y) subárbol izquierdo, *Ti*, mientras que su subárbol derecho, *Td*, se convierte en el subárbol derecho del nodo más a la derecha de *Ti*.

i) [1 pt.] Justifica que esta eliminación respeta las propiedades de ABB.

ii) [0.5 pts.] Muestra con ejemplos que esta eliminación puede tanto aumentar como reducir la altura del árbol original (dos ejemplos, uno para cada caso).

b) [1.5 pts.] Considerando que un nodo por sí solo —es decir, un árbol que sólo tiene la raíz— tiene ***profundidad*** 0, dibuja **árboles AVL** de profundidades 2 y 4 tales que *tengan el mínimo número de nodos posible*. Recuerda que un árbol AVL con el mínimo número de nodos está compuesto por una raíz y por dos subárboles AVL, cada uno a su vez con el mínimo número de nodos; comprueba que tus árboles cumplen esta propiedad.

c) Dados dos árboles AVL, T1 y T2, y una clave k, tales que todas las claves de T1 son menores que k y todas la claves de T2 son mayores que k, describe un algoritmo *eficiente* para implementar la unión de T1, k y T2 en un nuevo árbol AVL, T; es decir, T tiene exactamente todas las claves de T1 más todas las claves de T2 y más la clave k. Por "eficiente" entendemos lo siguiente: si el número total de claves involucradas es n, entonces tu algoritmo debe ser significativamente mejor que $O(n \log n)$.

i) [1 pt.] Considera primero el caso en que las alturas de T1 y T2 difieren a lo más en 1; ¿cuál es el algoritmo y cuál es su complejidad en este caso?

ii) [2 pts.] Considera ahora el caso en que la altura de T1 es dos o más que la altura de T2; ¿cuál es el algoritmo y cuál es su complejidad en este caso?