Estructuras de datos y algoritmos I-2012

Interrogación n° 2

Instrucciones generales

- Ingresa tu nombre en todas las hojas de respuesta
- Entrega 1 hoja por pregunta, independiente si no respondiste la pregunta
- Lee cuidadosamente cada pregunta
- Dispones de 120 minutos

Pregunta 1 – Ordenación (20 pts)

a) (8 pts) Escribe en seudo código el algoritmo de ordenación quicksort.

```
/** 5 pts */
void quicksort(int* arr, int from, int to) {
        if (to – from <= 0) return;
        int index_pivote = pivotear(arr,from,to);
        quicksort(arr,from,index_pivote -1);
        quicksort(arr,index_pivote+1,to);
}
// aquí el pivoteo es clave. Usaremos from como pivote
/** 3 pts */
int pivotear((int* arr,int from,int to) {
        int index_pivote = from;
        int valor_pivote = arr[index_pivote];
        for (int i=from+1; i<=to; i++){
                if (arr[i] < valor_pivote){
                        index_pivote++;
                        swap(arr,i,index_pivote);
                }
        }
        swap(arr,from,index_pivote);
        return index_pivote;
}
```

```
/* También podia hacerse con listas. */
/** 5 pts */
void quicksort(List& values) {
        if (values.size() <= 1) return;
        List menores;
        List mayores;
        int pivote = pivotear(values,menores, mayores);
        quicksort(menores);
        quicksort(mayores);
        values.clear(); // vacia la lista
        values.addAll(menores);
        values.add(pivote);
        values.add(mayores);
}
/** 3 pts */
int pivotear(List& values, List& menores, List& mayores) {
        Iterator it = values.iterator();
        int pivote = it.next(); //sabemos que hay al menos 1
        while (it.hasNext()) {
                int val = it.next();
                if (val <= pivote) menores.add(val);
                else mayores.add(val);
        }
        return pivote;
}
```

b) (6 pts) Aplica quicksort al arreglo [9, 5, 3, 1, 6, 21, 13, 11, 20, 25, 23], mostrando el estado del arreglo antes de elegir un pivote, marcando el pivote y mostrando el resultado de usar dicho pivote.

/* Puntaje: 0-4-6. 6 = Perfecto, 4 = Algún error menor, 0 = otros casos.*/

[9, 5, 3, 1, 6, 21, 13, 11, 20, 25, 23]

[9, 5, 3, 1, 6, 21, 13, 11, 20, 25, 23]

[5, 3, 1, 6, 9, 21, 13, 11, 20, 25, 23]

[5, 3, 1, 6, ~~9~~, 21, 13, 11, 20, 25, 23]

[3, 1, 5, 6, ~~9~~, 13, 11, 20, 21, 25, 23]

[3, 1, ~~5~~, ~~6~~, ~~9~~, 13, 11, 20, ~~21~~, 25, 23]

[1, 3, ~~5~~, ~~6~~, ~~9~~, 11, 13, 20, ~~21~~, 23, 25]

[~~1~~, ~~3~~, ~~5~~, ~~6~~, ~~9~~, ~~11~~, ~~13~~, ~~20~~, ~~21~~, ~~23~~, ~~25~~]

[1, 3, 5, 6, 9, 11, 13, 20, 21, 23, 25]

c) (6 pts) Identifica el algoritmo de ordenación utilizado en cada uno de los siguientes casos:

Caso 1

```
[25, 8, 5, 1, 7,20]
[ 1,25, 8, 5, 7,20]
[ 1, 5,25, 8, 7,20]
[ 1, 5, 7,25, 8,20]
[ 1, 5, 7, 8,25,20]
[ 1, 5, 7, 8,20,25]
```

Selection sort (2 pts)

Caso 2

```
(parte 1)
[ 8, 3,14, 5, 4, 6]
[ 8, 3,14, 5, 4, 6]
[ 8, 3,14, 5, 4, 6]
[ 8, 3,14, 5, 4, 6]
(parte 2)
[ 3, 8,14, 4, 5, 6]
[ 3, 8,14, 4, 5, 6]
[ 3, 4, 5, 6, 8,14]
```

Merge sort (2 pts)

Caso 3

```
(parte 1)
[ 2,13, 5,25, 7,21]
[ 2,13, 5,25, 7,21]
[13, 2, 5,25, 7,21]
[13, 2, 5,25, 7,21]
[25,13, 5, 2, 7,21]
[25,13, 5, 2, 7,21]
[25,13,21, 2, 7, 5]
(parte 2)
[21,13, 5, 2, 7,25]
[13, 7, 5, 2,21,25]
[ 7, 2, 5,13,21,25]
[ 5, 2, 7,13,21,25]
[ 2, 5, 7,13,21,25]
[ 2, 5, 7,13,21,25]
```

Heap sort (2 pts)

Pregunta 2 – Árboles (20 pts)

Considera la siguiente representación de árbol (estracto):

```
class Tree {
    public:
        double value();
        List<Tree>* children();
    private:
        double nodeValue;
        List<Tree>* childrenList;
}
```

Construye la clase TreeStatistics, que tiene los siguientes dos métodos:

```
double average(const Tree& tree);
double total(const Tree& tree);
```

los cuales, respectivamente, entregan el promedio de los valores del árbol y el total de los valores del árbol.

```
class TreeStatistics {
public:
    /** 10 pts */
    double average(const Tree& tree) {
        return total(tree) / count(tree);
    }
    /** 10 pts */
    double total(const Tree& tree) {
        double subtotal = tree.value();
        for (Iterator<Tree> it = tree.children().iterator(); it.hasNext(); ) {
            subtotal += total( it.next() );
        }
        return subtotal;
    }
```

```
private:
        int count(const Tree& tree) {
                int counted = 1;
                for (Iterator<Tree> it = tree.children().iterator(); it.hasNext(); ) {
                        counted += count( it.next() );
                }
                return counted;
        }
}
```

Pregunta 3 – Árboles de búsqueda binaria (20 pts)

a) (9 pts) El algoritmo de inserción de un árbol rojo negro podría dividirse en tres etapas.

Describe muy brevemente estas etapas (en qué consiste y cual es su objetivo o función).

1 .- Descender hasta el lugar de inserción, aplicando cambio de color cada vez que hay dos hermanos rojos, y arreglando con rotación simple o roble. Su objetivo es llegar al lugar donde se insertará el nodo, evitando que el padre del nodo sea rojo y tenga un hermano rojo. (3 pts)

2.- Insertar un nodo rojo. (3 pts)

3.- Si el padre es rojo, balancear con rotación simple o doble dependiendo del caso. (3 pts)

b) (5 pts) Tanto un árbol AVL como un árbol rojo negro son árboles ABB auto balanceados, cuyas operaciones (insertar / eliminar) no son triviales de implementar. Entonces, ¿por qué construir o usar una estructura tan complicada cuando existe otra mucho más simple (un árbol ABB “normal”) que cumple la misma función? (Responde brevemente)

Porque es más eficiente: al mantenerse balanceado, sus operaciones son O(log(n)), mientras que un ABB normal puede desbalancearse, haciendo que sus operaciones sean O(n) en el peor caso.

c) (6 pts) Considera el siguiente caso hipotético: “alguien” desarrolló una librería en C++ con una interfaz de árboles ABB (con operaciones para insertar, eliminar, consultar existencia y obtener mínimo y máximo) y 3 implementaciones: con un árbol ABB simple, un árbol AVL y un árbol rojo-negro. Considera también la siguiente función, que recibe un valor y una cota inferior y superior y verifica si una base de datos contiene el valor y si todos los valores de la base de datos están entre las cotas entregadas:

```
boolean containsAndInRange(int value, int lowerBound,
                int upperBound) {
        DataSource* ds = InMemoryDatabase::datasource();
        ABB* abb = ___________________;
        return abb->find(value)
```

```
                    && lowerBound <= abb->min()
                    && abb->max() <= upperBound;
    }
```

Teniendo esto en cuenta, puedes completar la función con una de estas opciones:

i. ds->getDataInSimpleABBTree()
ii. ds->getDataInAVLTree()
iii. ds->getDataAsRedBlackTree()

Elige una opción, y explica (brevemente) por qué la elegiste por sobre las otras 2.

Dado que lo único que se hará con el árbol es buscar (find) y obtener los valores máximo y mínimo, nos basta con un árbol que esté balanceado. Por esta razón, tanto un AVL como un rojo negro son igualmente válidos como solución. Dado que da igual entre estos 2, es preferible en primer lugar el que sea "nativo" del datasource, y en caso que ambos tengan que ser construidos por el datasource, el Rojo-Negro es preferible, pues es más eficiente en inserción.

Pregunta 4 – Uso de estructuras (20 pts)

En esta pregunta, considera que tienes las siguientes estructuras a tu disposición: ArrayList, LinkedList, DoublyLinkedList, Queue, Stack, Heap, HashMap, ABBTree, AVLTree, RedBlackTree y BTree.

Un programa lee enteros desde dos archivos, cada uno con un número variable entre 1 y 5 millones de datos. El programa debe escribir en un tercer archivo la lista de todos los números, en orden ascendente, que estén en alguno de los dos archivos. Así, por ejemplo, si un archivo contiene 67, 11, y 42 y el otro contiene 11 y 17, el programa deberá escribir un archivo con 11, 17, 42 y 67.

Elige la o las estructura(s) que mejor se ajusta(n) al problema. Para cada estructura, responde brevemente:

a) ¿Que rol desempeña?

b) ¿Por qué esa estructura y no otra?

c) Si esa estructura no estuviera disponible, ¿con cual la reemplazarías? ¿por qué?

Si crees que ayudaría a entender tu respuesta, escribe en seudo código cómo harías este programa usando las estructuras que elegiste.

Algoritmo:

Sea n = número total de datos.

Acá es importante identificar que, dado que los datos están en archivos, estaremos obligados a leerlos secuencialmente.

Si usamos un ABB balanceado, podemos hacer que las operaciones sean O(log(n)) cada una, y al leer secuencialmente un archivo y luego el otro, tendremos ~ n*log(n) operaciones. Si luego recorremos el árbol inorden (lo cual es O(n)), tendremos los datos ordenados.

Estructura(s): Árbol Rojo-Negro.

a) Rol: Los datos cada archivo se leen secuencialmente y se ingresan al árbol (primero un archivo, luego el segundo).

b) Porque permite hacer la carga de los datos en O(n*log(n)) operaciones, y obtener los datos ordenados en O(n). En comparativa con otras estructuras, un AVL es del mismo orden, pero sabemos que un AVL requiere más operaciones en inserción que un rojo negro (en el peor caso, aproximadamente el doble). Un heap también es del mismo orden de inserción, pero obtener los datos ordenados desde el heap requiere O(n*log(n)) operaciones.

Una tabla hash que usa como clave y valor los datos también suena atractivo (sus operaciones son ~O(1), manteniendo el factor de carga bajo 0.75). Sin embargo, esto tiene 2 inconvenientes:

i: si no inicializamos la tabla con el máximo n posible / 0.75, la tabla tendrá que hacer resize, y la operación de resize de una tabla hash es similar a la de un ArrayList : O(n). Así que o desperdiciamos mucho espacio (con una tabla de 13,3 millones de entradas, cuando podrían haber sólo 2 millones de datos), o resentimos el performance de la tabla hash.

Ii: la tabla hash no está ordenada, por lo que estaríamos obligados a ordenar los datos luego. Y sabemos que ordenar, en el mejor caso, es O(n*log(n)) (usando heapsort). Este argumento es fatal para todas las estructuras no ordenadas.

c) Con un AVL, por que inserción y obtener los datos ordenados sigue siendo O(n*log(n)) y O(n) respectivamente, mejor que cualquier otra estructura.