Algoritmos y Estructuras de Datos – Segundo Parcial

Licenciatura en Tecnologías Digitales, UTDT

Segundo Semestre 2023

- No está permitido comunicarse por ningún medio con otros estudiantes ni con otras personas durante el examen, excepto con los docentes de la materia.
- Puede consultarse a los docentes solo por aclaraciones específicas del enunciado.
- El examen es a libro abierto: está permitido tener todo el material **impreso** y apuntes personales que deseen traer. **No está permitido el uso de dispositivos electrónicos para este fin.**
- **Cada ejercicio debe resolverse en hoja aparte.**

Por favor, no escribir en este espacio.

Problema:	M3P1	M3P2	Total M3	M4P1	M4P2	Total M4
Nota:						

M3: Complejidad y Ordenamiento

Problema 1. (50 puntos) Complejidad
Sea el siguiente programa:

```
1  bool busquedaRecursiva(const vector<int>& arreglo, int inicio, int fin, int objetivo) {
2      if (inicio > fin) {
3          return false;
4      }
5      int cuarto = (fin - inicio) / 4;
6      int punto1 = inicio + cuarto;
7      int punto2 = inicio + 2 * cuarto;
8      int punto3 = inicio + 3 * cuarto;
9
10     if (arreglo[punto1] == objetivo || arreglo[punto2] == objetivo ||
11         arreglo[punto3] == objetivo) {
12         return true;
13     } else if (objetivo < arreglo[punto1]) {
14         return busquedaRecursiva(arreglo, inicio, punto1 - 1, objetivo);
15       else if (objetivo > arreglo[punto3])
16         return busquedaRecursiva(arreglo, punto3 + 1, fin, objetivo);
17     } else {
18         return busquedaRecursiva(arreglo, punto1 + 1, punto2 - 1, objetivo)
19             || busquedaRecursiva(arreglo, punto2 + 1, punto3 - 1, objetivo);
20     }
21 }
22 bool buscar(const vector<int>& arreglo, int objetivo) {
23     return busquedaRecursiva(arreglo, 0, arreglo.size() - 1, objetivo);
24 }
```

(a) Anotar el costo de cada línea del programa y dar la función de costo temporal $T(n)$ para el peor caso de la función `busquedaRecursiva`, donde n es el tamaño de entrada.

(b) Dibujar el arbol de recursión para la función de costo temporal $T(n)$ indicando los costos de cada nodo, los costos de cada nivel y la altura del árbol.

(c) Escribir la expresión de costo total del árbol de recursión realizado (no se pide el órden de complejidad asintótico O).

Problema 2. (50 puntos) Ordenamiento
Se quiere implementar una función
`vector<int> estan_en_alguno(const vector<int> & A, const vector<int> & B, const vector<int> & C)`
que toma por parámetro tres vectores de `int`. Se tiene como precondición que cada vector `A`, `B` y `C` está ordenado de forma creciente y no tiene elementos repetidos. Se debe devolver un vector que contenga todos los elementos de A que están en B o en C.

Ejemplos:

- si `A = {1,2,3,4,5}, B = {-1,1,2,5,9}, C = {-5,0,2,5,10}` se debe devolver `{1,2,5}`.
- si `A = {1,2,3,4,5}, B = {1,3,5,7,9}, C = {2,4,6,8,10}` se debe devolver `{1,2,3,4,5}`.
- si `A = {-2,-1,3,6,7}, B = {1,2,4,5,9}, C = {-5,0,2,5,10}` se debe devolver `{}`.

(a) Implementar la función **`estan_en_alguno`**. El algoritmo debe tener órden de complejidad temporal $O(n + m + k)$ en peor caso, donde $n = |A|$, $m = |B|$, $k = |C|$. Puede asumir que ejecutar X veces la operación `push_back` de `vector<int>` es de tiempo $O(X)$.

(b) Dar un ejemplo de vectores (todos del mismo tamaño N) `A, B, C` que sea el peor caso de ejecución del algoritmo y un ejemplo de vectores `A, B, C` que no sea el peor caso de ejecución del algoritmo. Justificar cada caso.

(c) Calcular la función de costo temporal $T(n, m, k)$ para el peor caso en función de n, m y k las longitudes de los vectores. Justificar por qué se cumple el órden $O(n + m + k)$ (ya sea utilizando la definición formal de O u operando con álgebra de órdenes).

M4: Estructuras de Datos

Problema 1. (50 puntos) Diseño de TADs
Se desea implementar un gestor de reservas de butacas de una sala de cine. Las butacas están numeradas de 1 a N. Los clientes, al crear una reserva, solicitan una cantidad de butacas **contiguas**. Si la cantidad pedida está disponible, se asigna algún rango de butacas; de lo contrario la reserva queda en *lista de espera* por si una cancelación futura de otra reserva libera butacas. Cuando ocurre una cancelación de reserva con butacas asignadas, se intenta cumplir con todas las reservas pendientes posibles siguiendo el orden de llegada original; es decir, una cancelación de una reserva grande potencialmente podría generar múltiples asignaciones de reservas pendientes.

```
1  class Reservas{
2  public:
3      Reservas(int cant_butacas);              // Pre: cant_butacas > 0.
4      int cantidad_butacas() const;
5      const set<string> & reservas() const;
6      const list<pair<string,int>> & pendientes() const;
7      // Pre: id no es reserva, cant_butacas > 0.
8      void reservar(const string& id, int cant_butacas);
9      void anular_reserva(const string& id);  // Pre: id es reserva
10     // Pre: id es reserva y no está en espera
11     const pair<int,int>& butacas_reserva(string id) const;
12
13 private:
14    int cantidad_butacas;
15    set<string> reservas_totales;
16    list<pair<string,int>> reservas_pendientes;
17    map<string,pair<int,int>> reservas_asignadas;
18    list<pair<int,int>> rangos_disponibles;
19 };
```

En la estructura dada, `cantidad_butacas` es el total de butacas existentes; `reservas_totales` registra todas las reservas por nombre; `reservas_pendientes` registra, para cada nombre de reserva pendiente, la cantidad de butacas requeridas; `reservas_asignadas` registra el rango de (butaca inicial, butaca final) de butacas asignadas a una reserva; y finalmente `rangos_disponibles` registra de manera **ordenada por cantidad de butacas consecutivas** los rangos de butacas vacíos, expresados con pares (butaca inicial, cantidad).

Se tienen las siguientes restricciones de complejidad, donde n es la cantidad de reservas totales y m es la cantidad total de butacas:

- `reservar(const string& id, int cant_butacas)` - $O(log\ n + m)$ en peor caso,

(a) Escribir en español el invariante de representación de la estructura interna.

(b) Dar un ejemplo de valores para las variables de la estructura interna que cumpla el invariante de representación y otro ejemplo de valores que no lo cumpla. Los ejemplos deben tener al menos 3 reservas asignadas, 3 reservas pendientes y butacas disponibles.

(c) Dar la implementación del método `reservar` respetando el invariante propuesto.

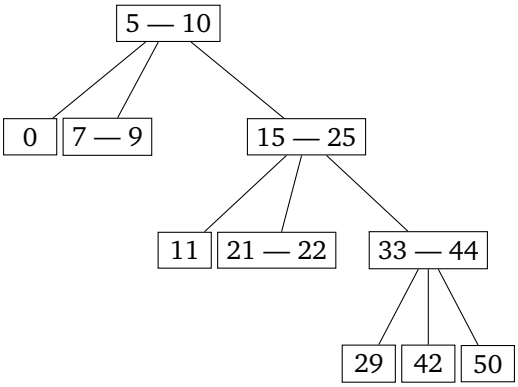
Problema 2. (50 puntos) Estructuras de Datos
Sea la siguiente definición e invariante de la estructura de datos **arbol ternario de busqueda (ATB)**:

```
1 struct ATB {
2   ATB* izq;
3   int valor1;
4   ATB* med;
5   int valor2;
6   ATB* der;
7 };
```

- `n.valor1 <= n.valor2`
- `n.izq`, `n.med` y `n.der` son arboles ternarios de búsqueda,
- si `n.valor1 == n.valor2`, entonces `n.izq`, `n.med` y `n.der` son `nullptr` (o sea, `n` es hoja con 1 valor)
- todo elemento de `n.izq` es menor que `n.valor1`,
- todo elemento de `n.med` es mayor que `n.valor1` y menor que `n.valor2`
- todo elemento de `n.der` es mayor que `n.valor2`,

La búsqueda e inserción de valores en un ATB son análogas a la búsqueda e inserción en ABB pero con 1 o 2 valores almacenados por nodo y 3 posibles ramas de búsqueda e inserción recursiva.
• Si al insertar el valor v se llega a un `nullptr`, se reemplaza con un `nodo nuevo` que tiene `valor1 == valor2 == v`;
• Si al insertar el valor v se llega a un nodo n donde n.valor1 == n.valor2,

- si v <n.valor1, se pone n.valor1 = v;
- si v >n.valor1, se pone n.valor2 = v;
- en ambos casos se termina inmediatemente (sin necesidad de crear un nodo nuevo).



ATB resultante luego de insertar: `10,5,7,0,15,25,11,22,33,21,44,42,29,50,9`
En los nodos con tres hijos nil(`nullptr`), se omiten los hijos por cuestiones de espacio.

Se pide implementar la siguiente operación:
`ATB* insertar(ATB* raiz, int elem)`
que, de manera recursiva, inserta el valor `elem` en el árbol y devuelve la raíz, manteniendo el invariante.