Algoritmos y Estructuras de Datos – Segundo Parcial

Licenciatura en Tecnologías Digitales, UTDT

Primer 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:	1	2	3	4	Total (sobre 100)
Nota:					

Problema 1. (20 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
6      int tercio = (fin - inicio) / 3;
7      int punto1 = inicio + tercio;
8      int punto2 = inicio + 2 * tercio;
9
10     if (arreglo[punto1] == objetivo) {
11         return true;
12     } else if (arreglo[punto2] == objetivo) {
13         return true;
14     } else if (objetivo < arreglo[punto2]) {
15         return busquedaRecursiva(arreglo, inicio, punto1 - 1, objetivo)
16                 || busquedaRecursiva(arreglo, punto1 + 1, punto2 - 1, objetivo);
17     } else {
18         return busquedaRecursiva(arreglo, punto1 + 1, punto2 - 1, objetivo)
19                 || busquedaRecursiva(arreglo, punto2 + 1, fin, objetivo);
20     }
21 }
22
23 bool buscar(const vector<int>& arreglo, int objetivo) {
24     return busquedaRecursiva(arreglo, 0, arreglo.size() - 1, objetivo);
25 }
```

(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. (25 puntos) Ordenamiento

Llamamos *escalera* a cualquier secuencia ordenada de valores, ya sea en orden ascendente o en orden descendente. Por ejemplo, el vector $[9, 3]$ es una escalera descendente y el vector $[5, 8, 10]$ es una escalera ascendente.

Dado un vector de enteros v de longitud n, se desea ordenarlo en complejidad temporal de peor caso $O(n \cdot k)$, donde k es la cantidad de escaleras que hay en el vector. Por ejemplo, supongamos que tenemos el siguiente vector $v = [8, 2, 5, 8, 10, 7, 6, 4]$. En este caso, $n = 8$ y $k = 3$ ya que contiene las siguientes escaleras:

- $[8, 2]$ (descendente),
- $[5, 8, 10]$ (ascendente), y
- $[7, 6, 4]$ (descendente).

Un algoritmo correcto debería ordenar el vector, obteniendo $[2, 4, 5, 6, 7, 8, 8, 10]$, en tiempo $O(n \cdot k)$.

(a) Implementar la función **`ordenar_escaleras`** que recibe por parámetro un vector de enteros v y lo ordena en complejidad temporal $O(n \cdot k)$ en peor caso, donde k es la cantidad de escaleras que hay en el vector.

(b) Sea v un vector de enteros cualquiera, considerando la complejidad temporal ¿Conviene ordenarlo con un algoritmo en $O(n \cdot k)$ o con el algoritmo **mergesort**? Justificar su respuesta.

Problema 3. (30 puntos) Diseño de TADs
Se desea almacenar y editar un documento de texto simplificado (palabras separadas por un único espacio) y poder determinar rápidamente qué palabra se encuentra en una posición dada. En este contexto, cada posición del texto se refiere a **un caracter individual del texto.**

```
1  class Editor {
2  public:
3      Editor(const string& texto_inicial);
4
5      // longitud del texto medido en CARACTERES incluyendo espacios
6      int longitud() const;
7
8      // cantidad de palabras del texto
9      int cantidad_palabras() const;
10
11     // Pre: 0 <= pos < longitud()
12     const string& palabra_en(int pos) const;
13
14     // Pre: 0 <= i < cantidad_palabras();
15     const string& iesima_palabra(int i) const;
16
17     // Pre: la palabra está en el texto
18     int primera_aparicion(const string & palabra) const;
19
20     // inserta una palabra antes de la palabra ubicada en pos
21     // Pre: 0 <= pos <= longitud() y palabra no contiene espacios
22     void insertar(string palabra, int pos);
23
24 private:
25     vector<string>  _palabras;
26     /* completar */
27 };
```

Con la estructura de representación dada se almacenan las palabras individuales en `_palabras`, en el orden en el que aparecen en el texto. Considere las sigueintes restricciones de complejidad, donde N es la cantidad de palabras totales del texto y S es la longitud de la palabra más larga del texto:

- `longitud` en $O(1)$, devuelve la cantidad total de caracteres del texto,
- `palabra_en` en $O(\log N)$, dada una posición (de un caracter) devuelve la palabra completa que se encuentra en esa posición,

- `iesima_palabra` en $O(1)$, que devuelve la i-ésima palabra del texto,
- `primera_aparicion` en $O(S \cdot \log N)$, que devuelve la posición del primer caracter de la primera aparición de la palabra en el texto, y
- `insertar`, sin requerimiento de complejidad, que dada una posición de un caracter, inserta una palabra justo antes de la palabra que se encuentra en esa posición.

(a) Agregar a la estructura interna lo que considere necesario para que se puedan implementar las operaciones en la complejidad deseada.

(b) Escribir en español y en lenguaje formal el invariante de representación de la estructura interna.

(c) Dar la implementación del método `insertar` respetando el invariante propuesto y calcular el orden de complejidad O en peor caso (NO se debe demostrar formalmente con la definición de O).

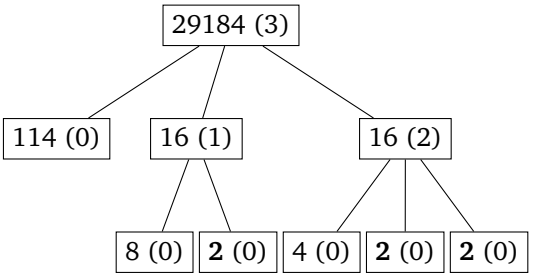
Problema 4. (25 puntos) Estructuras de datos
Sea la siguiente definición e invariante de la estructura de datos **arbol ternario de divisores**:

```
struct arbol_ternario {
  int valor;
  arbol_ternario* izq;
  arbol_ternario* med;
  arbol_ternario* der;
  int cant_descendientes_primos;
};
```

- todos los `valores` del arbol son enteros positivos mayores a 1;
- un nodo nunca puede tener un sólo hijo no nulo;
- el valor de un nodo es igual al producto de los valores de sus hijos directos;
- `cant_descendientes_primos` almacena, para un nodo, la cantidad de números primos presentes en sus subárboles hijos;

Por ejemplo, en el siguiente arbol ternario de divisores,

- `cant_descendientes_primos` de (29184) es 3,
- `cant_descendientes_primos` del primer (16) es 1,
- `cant_descendientes_primos` del segundo (16) es 2,
- y `cant_descendientes_primos` de todo otro nodo es 0.



A: Ejemplo de arbol ternario de divisores. (Descendientes primos se indica entre paréntesis)

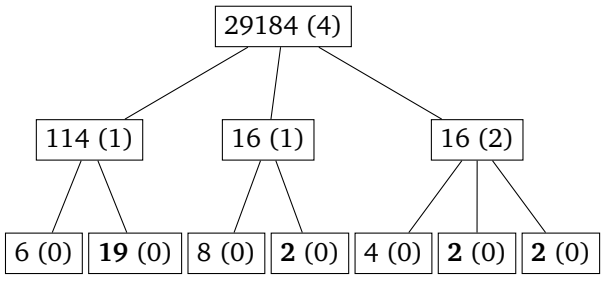
Implementar las siguientes operaciones:

(a) `void desplegar(arbol_ternario* raiz, int elem)`

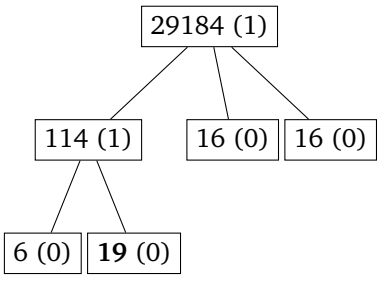
que asume que **elem** no es primo y agrega al menos dos hijos a todos los nodos **sin hijos** que contengan el valor **elem**, manteniendo el invariante;

(b) `void plegar(arbol_ternario* raiz, int elem)`

que, de manera inversa a `desplegar`, elimina todos hijos de los nodos que contengan `valor` **elem**, actualizando como corresponda la estructura para mantener el invariante;



B: Resultado de `desplegar(A,114)`



C: Resultado de `plegar(B,16)`

Para implementar las funciones puede asumir que existen las funciones auxiliares:

- `bool es_primo(int n)` que devuelve `true` si `n` es primo y `false` en caso contrario, y
- `int dame_un_divisor(int n)` que asume que `n` no es primo y devuelve un divisor de `n` (distinto de 1 y de `n`).