Algoritmos y Estructuras de Datos – Segundo Parcial

Licenciatura en Tecnologías Digitales, UTDT

Primer Semestre 2024

- 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.**
- **Cada módulo se aprueba con 60 puntos totales y un mínimo de 20 puntos en cada ejercicio**.

M3: Tipos Abstractos de Datos

Contamos con el siguiente tipo abstracto para modelar una biblioteca musical, en donde cada biblioteca tiene un conjunto de canciones (identificadas por su título y autor) y las mismas se pueden ordenar en listas de reproducción. Una lista puede contener canciones repetidas, y puede haber canciones que no aparezcan en ninguna lista.

```
// Renombres para mejorar la legibilidad
typedef string nombre_artista;
typedef string nombre_lista;
typedef string nombre_cancion;

struct Cancion {
  nombre_artista artista;
  nombre_cancion titulo;

  // Utilizado internamente por estructuras que requieran orden entre los
  // elementos (como set o map). Asumir complejidad O(1).
  bool operator<(const Cancion& other) const {
    return titulo < other.titulo;
  }

  // Permite comparar dos canciones mediante "==". Asumir complejidad O(1).
  bool operator==(const Cancion& other) const {
    return titulo == other.titulo && artista == other.artista;
  }
};

class BibliotecaMusical {
  public:
    void agregar_cancion(const Cancion& cancion);
    void eliminar_cancion(const Cancion& cancion);
    void crear_lista(nombre_lista lista);
    void agregar_a_lista(nombre_lista lista, const Cancion& cancion);
    void quitar_de_lista(nombre_lista lista, const Cancion& cancion);

    const set<Cancion>& canciones_de(nombre_artista artista) const;
    const set<nombre_lista>& listas_que_contienen(const Cancion&) const;
  private:
    set<Cancion> _canciones;
    map<nombre_lista, list<Cancion>> _canciones_por_lista;

    // Estructuras auxiliares para mejorar la complejidad de algunas operaciones.
    map<nombre_artista, set<Cancion>> _canciones_por_artista;
    map<Cancion, set<nombre_lista>> _listas_por_cancion;
};
```

Problema 1. (50 puntos)
Se pide:

(a) Escribir en lenguaje natural el invariante de representación de la clase.

(b) Proveer una implementación para `quitar_de_lista` con la mejor complejidad temporal asintótica posible, escribir dicho orden de complejidad y justificarlo utilizando álgebra de órdenes. La complejidad debe ser expresada en función de:

- la cantidad de canciones (C)
- cantidad de artistas (A)
- cantidad de listas reproducción (L)
- la longitud de la lista de reproducción más larga (P)

Problema 2. (50 puntos)
Se desea modificar el TAD Biblioteca Musical con una nueva funcionalidad: la de conocer, para cada autor, sus canciones más populares dentro de la Biblioteca. La popularidad de una canción se mide contando cuántas apariciones en listas de reproducción tiene[1].

Con este objetivo se desea agregar a la interfaz pública de `BibliotecaMusical` la siguiente operación:

```
1 //Pre: El autor tiene al menos una canción registrada en la Biblioteca
2 //Post: res es la lista de canciones del autor, en orden decreciente de popularidad.
3 const list<Cancion> & mas_populares_de_artista(const nombre_artista& artista) const;
```

Para un artista dado, el método debe devolver su lista completa de canciones ordenadas por popularidad, y en caso de empatar en popularidad, ordenadas alfabéticamente por título. La complejidad de la operación debe ser $O(\log(A))$ donde A es la cantidad de autores diferentes presentes en la bibliteca.

(a) Describir qué cambios a la estructura de representación serán necesarios para implementar la nueva funcionalidad. Listar los métodos preexistentes que deban ser modificados y explicar brevemente por qué.

(b) Describir en castellano qué condiciones sería necesario agregar al invariante de representación de la estructura. Listar sólo las condiciones "extra" necesarias para relacionar lo que se agregue a la estructura con los campos preexistentes.

(c) Implementar en C++ la operación `mas_populares_de_artista` y justificar que cumple la complejidad pedida.

(d) £Qué consideraciones deben tenerse en cuenta al implementar `agregar_a_lista` para preservar el invariante de la estructura?

[1] Si una canción aparece n veces en una lista cualquiera, las n apariciones se cuentan por separado a fines de computar la popularidad de la canción.

M4: Algorítmica y Estructuras de Datos

Problema 1. (50 puntos) Ordenamiento
Considerar un sistema para procesar los votos de una elección donde se tienen V votos C candidatos. Cada voto v_i se representa con un número entero tal que $0 \leq v_i < V$, donde el número es un identificador asociado a cada candidato.

(a) Implementar la siguiente función, con complejidad temporal de peor caso $O(V + C)$

```
vector<int> calcular_votos_por_candidato(const vector<int> & votos)
```

La función es tal que:

- $|res| = max(votos) + 1$.
- $(\forall c.\ \texttt{int})\ (0 \leq c < |res| \implies res[c] = \sum_{i=0}^{|votos|-1} \beta(votos[i] = c))$.

(b) Implementar la siguiente función, con complejidad temporal de peor caso $O(V + C)$

```
list<list<int>> calcular_resultados(const vector<int> & votos_por_candidato, int V)
```

La misma toma como entrada el resultado de la función anterior y la cantidad total de votos, y devuelve los candidatos ordenados por cantidad de votos. Las listas internas representan grupos de candidatos con la misma cantidad de votos.

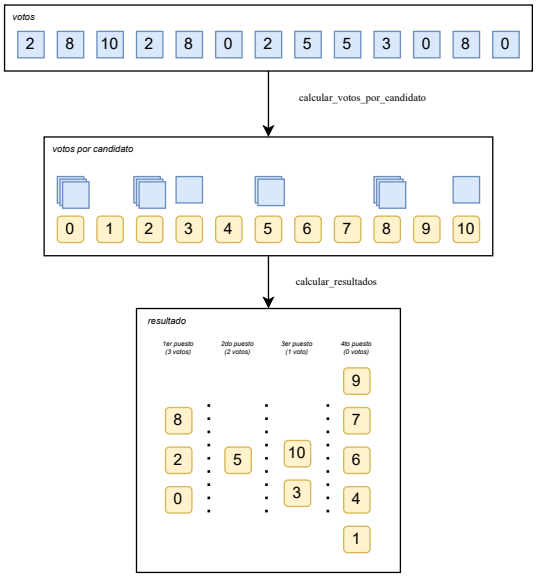
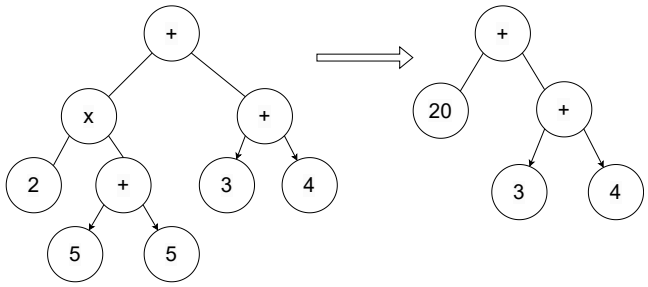


Figura 1: Ejemplo de cálculo de resultados

Problema 2. (50 puntos) Estructuras de Datos
Se quiere implementar una estructura que representa una expresión matemática compuesta por valores numéricos y operaciones de suma o de multiplicación. Para esto se elige una estructura de árbol binario:

```
1  struct Nodo {
2      char op;
3      int valor;
4      Nodo* izq;
5      Nodo* der;
6  };
```



Ejemplo de aplicar `colapsar_multiplicaciones`.

Las hojas del árbol almacenan los operandos de cada operación en el campo `valor`, mientras que los nodos internos poseen un caracter `op` que indica qué tipo de operación se realiza. En la figura se puede ver un árbol que representa la expresión `(2 * (5 + 5)) + (3 + 4)`.

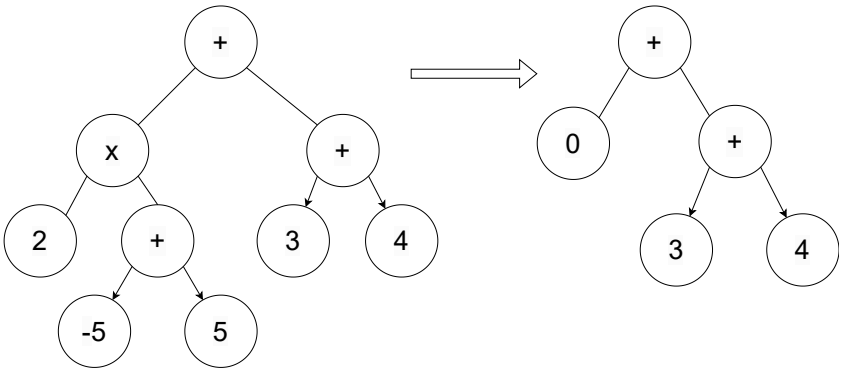
El árbol de operaciones tiene el siguiente invariante de representación:

- Todo nodo tiene 0 o 2 hijos.
- Si un nodo n tiene 0 hijos, entonces `n->op == ' '`.
- Si un nodo n tiene 2 hijos, entonces `n->op == '+'` ó `n->op == '*'`.

Implementar las siguientes funciones:

(a) `void colapsar_multiplicaciones(Nodo *raiz)`: reemplaza todo nodo de multiplicación del arbol por una hoja conteniendo el resultado de calcular ese producto. Borra los nodos excedentes del árbol.

(b) `int evaluar_y_simplificar(Nodo *raiz)`: devuelve el resultado de evaluar la expresión del árbol $raiz$, y simplifica el árbol reemplazando los nodos de operación cuyo resultado sea igual a 0 por hojas con valor igual a 0.



Ejemplo de `evaluar_y_simplificar` un arbol de operaciones. Notar que simplifica dos veces: primero el subárbol `(-5+5)` a 0 y luego simplifica el subárbol resultante `(2*0)` a 0. El valor devuelto por `evaluar_y_simplificar` es, en este caso, `7`.

Ambas operaciones deben tener complejidad O(n) donde n es la cantidad total de nodos en el arbol.

Para implementar las operaciones pedidas, se cuenta con las siguientes funciones dadas:

- `void borrar(Nodo *raiz)` que elimina la estructura recursivamente desde un nodo $raiz$.
- `int evaluar(Nodo *raiz)` que evalúa recursivamente la expresión matemática representada por la estructura partiendo desde un nodo $raiz$ (sin modificar la estructura recibida) y devuelve el valor obtenido. Por ejemplo, para los árboles de la primera figura, devuelve el valor 27, y para los de la segunda, devuelve el valor 7.