Algoritmos y Estructuras de Datos – Recuperatorios

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

Problema 1. (50 puntos) Tipos Abstractos
Contamos con el siguiente tipo abstracto para modelar un hostel, en donde hay reservas con fecha de inicio, fin, cantidad de camas reservadas y un string que identifica el nombre del huesped que realizó la reserva. El hostel nunca acepta reservas si estas exceden su capacidad de `_camas_totales` para algún día.

```
// Renombres para mejorar la legibilidad
typedef int Fecha;
typedef string Nombre;

struct Reserva {
    Fecha desde;
    Fecha hasta;
    int cantidad_camas;
    Nombre huesped;

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

class Hostel {
    public:
        bool reservar(Fecha desde, Fecha hasta, int cantidad_camas, Nombre huesped);

    private:
        int _camas_totales;
        list<Reserva> _reservas;

        // Estructuras auxiliares para mejorar la complejidad de algunas operaciones.
        map<Nombre, set<Reserva>> _reservas_por_huesped;
        map<Fecha, set<Reserva>> _reservas_por_fecha;
};
```

La función `reservar` verifica si la reserva excede la capacidad del hostel. Si hay camas disponibles para todas las fechas, entonces agrega la reserva y devuelve `true`. En otro caso devuelve `false` sin modificar la estructura. Se pide:

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

(b) Proveer una implementación para `reservar` 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 longitud de la reserva más larga, medida en fechas (L).
- la cantidad total de reservas (R).
- la cantidad de huespedes distintos (H).
- la cantidad de fechas distintas utilizadas (F).

Problema 2. (50 puntos)
Se desea modificar el TAD Hostel con una nueva funcionalidad: la de conocer sus huespedes más frecuentes. Los huespedes más frecuentes se miden contando la cantidad de fechas distintas en las que un determinado huesped reservo al menos una cama. Por ejemplo, si Juan tiene 2 reservas, una de 3 días y otra de 4, su frecuencia será de 7 (sin importar cuántas camas reservó).

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

```
1 //Pre: True
2 //Post: res es la lista de huespedes, en orden decreciente de frecuencia.
3 const list<Nombre> & huespedes_frecuentes() const;
```

El método debe devolver la lista completa de huespedes ordenados por frecuencia, y en caso de empatar, ordenados alfabéticamente por nombre. La complejidad de la operación debe ser $O(1)$.

(a) Describir qué cambios a la estructura de representación serán necesarios para implementar la nueva funcionalidad.

(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 `huespedes_frecuentes` y justificar que cumple la complejidad pedida.

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

Algoritmos y Estructuras de Datos – Recuperatorios

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**.

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.

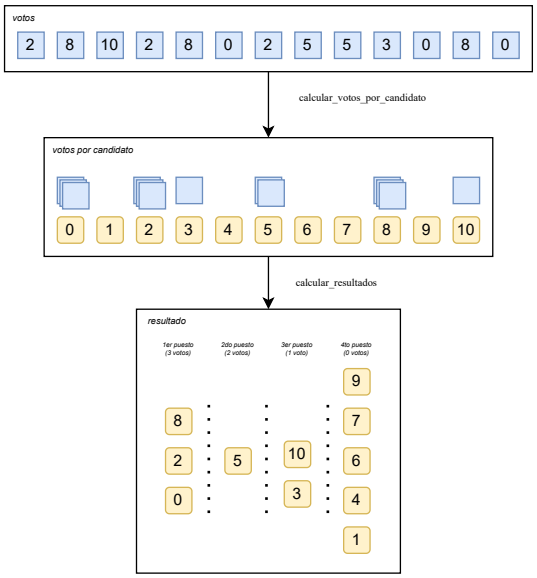
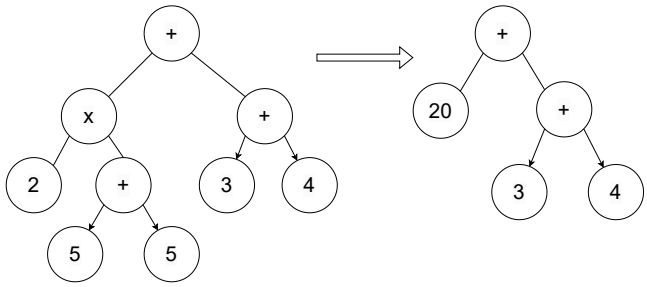


Figura 1: Ejemplo de cálculo de resultados

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:

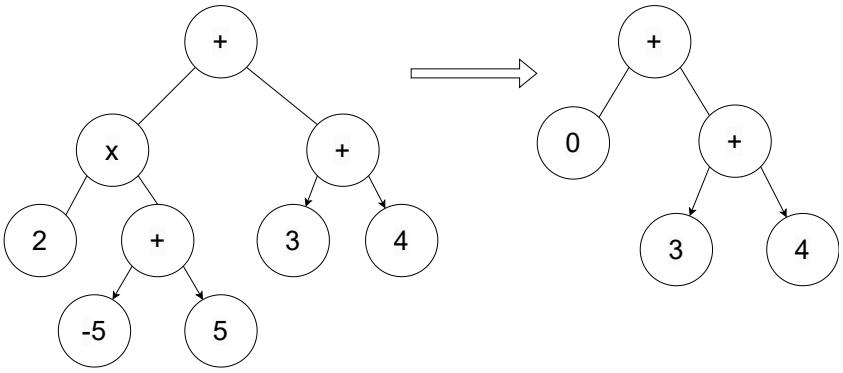
- Todo nodo tiene 0 o 2 hijos.
- Si un nodo n tiene 0 hijos, entonces `n->op == ' '`.
- Si un nodo n tiene 2 hijos, entonces `n->op == '+'` ó `n->op == '*'`.

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.

Para implementar estas operaciones se cuenta con la función `void borrar(Nodo *raiz)` que elimina la estructura recursivamente desde un nodo $raiz$. **Ambas operaciones deben tener complejidad O(n) donde n es la cantidad total de nodos en el arbol**.



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`.