

Examen de Programación 2

Febrero de 2025

Problema 1 (40 puntos: 20+20)

Se quieren definir procedimientos para eliminar hojas de árboles binarios (AB) y árboles generales (AG). Tener presente que un nodo es hoja si no tiene hijos, tanto en un AB como en un AG. Los procedimientos deberán tener $O(n)$ en el peor caso, siendo n la cantidad de nodos del árbol, en cada caso.

a) Considere la siguiente definición del tipo **AB** de árboles binarios de enteros, en memoria dinámica:

```
typedef nodoAB * AB;
struct nodoAB { int dato; AB izq, der; };
```

Implemente un procedimiento recursivo **elimHojasAB** que dado un AB t , elimine de t todos sus nodos hojas, liberando la memoria que corresponda. Si el árbol es vacío (t es NULL), el procedimiento no tendrá efecto. No se pueden implementar funciones o procedimientos auxiliares.

```
void elimHojasAB (AB & t)
```

b) Considere ahora la siguiente definición del tipo **AG** de árboles generales de enteros, representados con árboles binarios mediante la semántica: primer hijo (pH) – siguiente hermano (sH).

```
typedef nodoAG * AG;
struct nodoAG { int dato; AG pH, sH; };
```

Implemente un procedimiento recursivo **elimHojasAG** que dado un AG t , elimine de t todos sus nodos hojas, liberando la memoria que corresponda. Si el árbol es vacío (t es NULL), el procedimiento no tendrá efecto. No se pueden implementar funciones o procedimientos auxiliares, ni usar iteración (while/for).

```
void elimHojasAG (AG & t)
```

Problema 2 (60 puntos: 8+34+18)

a) **Especifique**, incluyendo pre y postcondiciones, un TAD **Ranking** para representar y manipular espectáculos según sus taquillas, donde los títulos de los espectáculos son de tipo *char ** y las recaudaciones de taquilla son de tipo *int*. En el **Ranking** los espectáculos están ordenados según el valor de recaudación, de mayor a menor. El espectáculo más taquillero es el que ocupa la primera posición del **Ranking**. No hay límite en la cantidad de espectáculos que pueden almacenarse. Considere únicamente las siguientes operaciones:

1. **crear**, que genera un Ranking vacío.
2. **ingresar**, que recibe un nuevo título y su recaudación, y almacena el espectáculo dentro del **Ranking**. Se asume como precondition que el espectáculo (su título) no existe previamente en el **Ranking**. Si más de un espectáculo tiene la misma recaudación, aparecerá antes en el **Ranking** el último que fue ingresado.
3. **posicion**, que recibe el título de un espectáculo y retorna su posición dentro del **Ranking**, si existe. En caso contrario, retorna cero (0).
4. **taquilla**, que recibe el título de un espectáculo incluido en el **Ranking** (precondition) y retorna su valor de recaudación de taquilla.
5. **ranking**, que recibe una posición mayor que cero y retorna el título del espectáculo que ocupa esa posición. Se asume como precondition que: $posicion > 0$ y $posicion \leq$ que la cantidad de espectáculos en el **Ranking**.
6. **taquillero**, que retorna el título de del espectáculo con mayor taquilla de un **Ranking** no vacío. Recordar que si más de un espectáculo tiene la misma recaudación, aparecerá antes en el **Ranking** el último que fue ingresado.
7. **cantidad**, que retorna la cantidad de espectáculos de un **Ranking**.
8. **destruir**, que elimina todos los espectáculos del **Ranking** y libera toda su memoria.

b) Desarrolle una **implementación** del TAD anterior en la que las operaciones 1, 6 y 7 tengan $O(1)$ de tiempo de ejecución en el peor caso, y las restantes operaciones tengan $O(n)$ de tiempo de ejecución en el peor caso, siendo n la cantidad de espectáculos en el *Ranking*. Concretamente, **escriba la representación elegida y el código de todas las operaciones, excepto 4 y 5**, que puede asumir implementadas.

c) Implemente una operación **unir** que, dados dos *Rankings* $r1$ y $r2$, agregue a $r1$ los espectáculos de $r2$ que no existen en $r1$, de forma ordenada en el *Ranking* según su recaudación. Los espectáculos de $r1$ que también existan en $r2$ no serán modificados.

`void unir (Ranking & r1, Ranking r2)`

Asuma que los elementos de tipo *char* * (cadenas) se pueden asignar con = y comparar con los operadores ==, <, >.

SOLUCIONES

1-a)

```
void elimHojasAB (AB & t){
    if (t != NULL){
        if (t->izq == NULL && t->der == NULL) {
            delete t;
            t = NULL;
        } else{
            elimHojasAB (t->izq);
            elimHojasAB (t->der);
        }
    }
}
```

1-b)

```
void elimHojasAG (AG & t){
    if (t != NULL){
        if (t->pH == NULL) {
            AG aBorrar = t;
            t = t->SH;
            delete aBorrar;
            elimHojasAG (t);
        } else{
            elimHojasAG (t->pH);
            elimHojasAG (t->SH);
        }
    }
}
```

2-a)

```
struct representacionRanking;
typedef representacionRanking* Ranking;

// POS: genera un Ranking vacío.
Ranking crear();

// PRE: no existe el título en r
//POS: almacena el espectáculo en el Ranking en la posición que corresponda.
void ingresar (char* titulo, int taquilla, Ranking & r);

// POS: retorna posición dentro del Ranking si existe, si no existe retorna cero.
int posicion (char* titulo, Ranking r);

//PRE: el espectáculo está incluido en el Ranking.
//POS: retorna su valor de recaudación de taquilla.
int taquilla (char* titulo, Ranking r);

//PRE: cantidad(r) >= posición > 0
//POS: retorna el título del espectáculo que ocupa la posición dada en r.
char* ranking (unsigned int pos, Ranking r);

//PRE: Ranking no vacío
* POS: retorna el título del espectáculo con mayor taquilla. Si más de un espectáculo
tiene la misma recaudación, aparecerá antes el último que fue ingresado */
char* taquillero (Ranking r);

// POS: retorna la cantidad de espectáculo de un Ranking.
int cantidad (Ranking r);
```

```
// POS: elimina todos los espectáculos del Ranking y libera su memoria
void destruir (Ranking &r);
```

2-b)

```
struct nodoLista {
    char* titulo;
    int recaudacion;
    nodoLista* sig;
};

struct representacionRanking {
    nodoLista* lista;      // lista ordenada por taquilla, de mayor a menor
    unsigned int cantidad; // cantidad de espectáculos en el Ranking
};

typedef representacionRanking* Ranking;

Ranking crear() {
    Ranking nuevo = new representacionRanking;
    nuevo->lista = NULL;
    nuevo->cantidad = 0;
    return nuevo;
}

void ingresar (char* titulo, int taquilla, Ranking & r) {
    assert(posicion (titulo,r) == 0); // No existe el titulo en r. Chequeo opcional
    nodoLista* espectaculo = new nodoLista;
    espectaculo->titulo = titulo;
    espectaculo->recaudacion = taquilla;
    if (r->lista == NULL || r->lista->recaudacion <= taquilla) {
        espectaculo->sig = r->lista;
        r->lista = espectaculo;
    }
    else {
        nodoLista* iter = r->lista;
        while (iter != NULL && iter->sig != NULL &&
            iter->sig->recaudacion > taquilla) {
            iter = iter->sig;
        }
        espectaculo->sig = iter->sig;
        iter->sig = espectaculo;
    }
    r->cantidad++;
}

int posicion (char* titulo, Ranking r) {
    int pos = 0;
    nodoLista* iter = r->lista;
    while (iter != NULL && iter->titulo != titulo) {
        pos++;
        iter = iter->sig;
    }
    if (iter != NULL) return pos;
    else return 0;
}

char* taquillero (Ranking r) {
    assert(cantidadPeliculas(r) != 0); // Chequeo opcional
    return r->lista->titulo;
}

int cantidad (Ranking r) {
    return r->cantidad;
}
```

```

void destruir (Ranking& r) {
    while (r->lista != NULL) {
        nodoLista* borro = r->lista;
        r->lista = r->lista->sig; // delete [] borrar->titulo;
        delete borro;
    }
    delete r;
}

```

2-c)

```

void unir (Ranking & r1, Ranking r2) {
    int cant = cantidad (r2);
    for (int i = 1; i <= cant; i++) {
        char* titulo = ranking (i, r2);
        int recaudado = taquilla (titulo, r2);
        if (posicion (titulo, r1) == 0)
            ingresar (titulo, recaudado, r1);
    }
}

```