Examen de Programación 2

26 de julio de 2018

Problema 1 (40 puntos)

(a) Defina la estructura de datos vista en el curso para la representación de árboles binarios de enteros `AB`.

(b) Especifique completamente el TAD `Cola` de `AB` (con pre y post condiciones).

(c) Suponiendo que el árbol no cumple ninguna propiedad de orden y que puede contener elementos repetidos, se pide implementar una operación `nivelSumaS` que recibe un un árbol `a` y devuelve el nivel hasta el cual se deben sumar los elementos del árbol desde la raiz, para alcanzar un valor dado `S`. Para el árbol vacío el resultado debe ser -1 y el nivel de la raíz se considera que es el 0.

```
int nivelSumaS(AB & a, int S);
```

(d) Indique el orden del algoritmo implementado en la parte (c) justificando brevemente en lenguaje natural.

Solución:

(a)

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

(b)

```
// Crea una cola no acotada de elementos de tipo AB, vacia.
Cola crearCola();

// Encola un elemento de tipo AB.
void encolar(Cola & cola, AB t);

// Desencola el elemento mas antiguo de cola. Precond: !esVacia↩
   (cola)
void desencolar(Cola & cola);

// Retorna el elemento mas antiguo de cola. Precon: !esVacia(↩
   cola)
AB frente(Cola cola);

// Retorna true si la cola esta vacia.
bool esVacia(Cola cola)

// Destruye la memoria asociada a la cola pero no la memoria ↩
   asociada a sus elementos.
void destruirCola(Cola & cola);
```

(c)

```
/**
  Tener en cuenta que la funcion se implemento de tal forma que↩
     :
   * Los nodos se procesan por nivel, de izquierda a derecha
```

```
   * En caso de no llegar al valor S una vez sumados todos los nodos, se devuelve el mayor nivel alcanzado.
*/
int nivelSumaS(AB & a, int S) {
    int result = -1;
    bool termine = false;

    if (a != NULL) {
        Cola cAux = crearCola();
        encolar(cAux, NULL);
        /** Se inserta el arbol NULL como una marca de "fin de nivel". Notar ue cada vez que se llega a esta marca, se inserta otra
        marca, que siempre queda encolada despues de todos los nodos del nivel que se esta procesando.
        La primera es especial porque su funcion es llevar a que result valga 0.
        */
        encolar(cAux, a);
        while (!esVacia(cAux) && !termine) {
            AB frenteAB = frente(cAux);
            desencolar(cAux);
            if (frenteAB == NULL) {
                result++;
                encolar(cAux, NULL);
            } else {
                S = S - frenteAB->valor;
                if (S <= 0) {
                    termine = true;
                } else {
                    encolar(cAux, frenteAB->izq);
                    encolar(cAux, frenteAB->der);
                };
            };
        };
        destruirCola(cAux);
    };
    return result;
}
```

(d) El peor caso es O(n) siendo n la cantidad de elementos del árbol porque pasa por cada nodo una vez. Se da cuando los nodos del árbol suman menos que S.

Problema 2 (30 puntos)

Considere el TAD Pila que es una pila no acotada, donde los elementos guardados son enteros. Se pide:

(a) Especifique completamente el TAD `Pila` (con pre y post condiciones).

(b) Implemente el TAD definiendo el tipo para la representación elegida y desarrollando el código de todas las operaciones.

(c) ¿Qué cambiaría en la especificación de la parte (a) si la pila fuera acotada?

Solución:

(a)

```
// CONSTRUCTORAS
/* Crea una pila vacía. */
Pila crearPila();
/* Inserta i en la cima de p. */
void apilar(int i, Pila &p);

// SELECTORAS
/* Devuelve la cima de p. Precondicion: !esVaciaPila(p). */
int cima(Pila p);
/* Remueve la cima de p. Precondicion: !esVaciaPila(p). */
void desapilar(Pila &p);

// PREDICADOS
/* Devuelve 'true' si p es vacia, 'false' en otro caso. */
bool esVaciaPila(Pila p);

// DESTRUCTORA
/* Libera toda la memoria ocupada por p. */
void destruirPila (Pila &p);
```

(b)

```
struct RepresentacionPila {
    int valor;
    RepresentacionPila* sig;
};

typedef RepresentacionPila * Pila;

Pila crearPila() {
    Pila p = NULL;
    return p;
};

void apilar(int i, Pila &p){
    RepresentacionPila * nuevo = new RepresentacionPila;
    nuevo->valor = i;
    nuevo->sig = p;
    p = nuevo;
};

int cima(Pila p){
    return (p->valor);
```

```
};

void desapilar(Pila &p){
    RepresentacionPila * temp = p;
    p = p->sig;
    delete temp;
};

bool esVaciaPila(Pila p){
    return(p == NULL);
};

void destruirPila (Pila &p){
    RepresentacionPila * temp;
    while (p != NULL){
        temp = p;
        p = p->sig;
        delete temp;
    }
};
```

(c) Para hacer que la pila sea acotada, una solución son las siguientes tres cosas: Cambiar el cabezal de crearPila, para que permita una cota, crear el predicado estaLlenaPila, que indique cuando la pila está llena. Y finalmente agregarlo como precondición en apilar, para que solo se apile un elemento cuando la pila no esté llena. Es decir:

```
  /* Crea una pila vacía, que podrá contener hasta cota ↩
     elementos. */
Pila crearPila (int cota);

/* Devuelve 'true' si p tiene cota elementos, donde cota es el ↩
   valor del parametro pasado en crearPila, 'false' en otro ↩
   caso.  */
bool esLlenaPila(Pila p);

/* Inserta i en la cima de p. Precondición: !esLlenaPila(p) */
void apilar(int i, Pila &p);
```

Una solución alternativa respecto a apilar es que no tenga precondición y que en la especificación se indique que se elimina el elemento más antiguo en caso de estar llena.

Problema 3 (30 puntos)

Considere el TAD `Tabla` para almacenar los números de teléfono (números enteros) asociados a nombres de personas (strings), para la cual puede estimarse un una cantidad N de teléfonos a almacenar.

```
Tabla crearT (unsigned int N);
//Crea una tabla vacía

void AsociarT (string nom, unsigned int tel, Tabla & t);
//Asocia el teléfono tel al nombre nom en la Tabla t
//Si ya tenía asociado un teléfono, se modifica

bool estaNomT (string nom, Tabla t);
//Devuelve verdadero si el nombre nom tiene un teléfono asociado

unsigned int darTelT (string nom, Tabla t);
//Devuelve el teléfono asociado al nombre nom
//Precond:  el nombre nom tiene un teléfono asociado

void eliminarT (string nom, Tabla t);
//Elimina la asociación del nombre nom
//Precond:  el nombre nom tiene un teléfono asociado
```

Asumiendo que para la agenda telefónica implementada con una `Tabla` se cumple lo siguiente:

- cualquier string puede ser un nombre de persona
- es igualmente probable que se desee asociar un número de teléfono a cualquier persona
- la cantidad de asociaciones a almacenar puede estimarse en un valor N
- los nombres de las personas no se repiten
- los strings se pueden asignar y comparar como los tipos básicos
- se dispone de una función h_N que dado un string devuelve un entero en el rango [0, N-1] (y distribuye de manera uniforme los strings en [0, N-1])

Se pide:

(a) Defina el tipo para la representación elegida del TAD `Tabla` de modo que todas las operaciones tengan O(1) de tiempo de ejecución en el caso promedio, excepto `crearT`.

(b) Desarrolle el código de las operaciones `crearT` y `AsociarT`. `AsociarT` debe tener O(1) de tiempo de ejecución en el caso promedio.

(c) Explique cómo la implementación elegida permite cumplir con los requerimientos.

Solución:

(a)

```
struct nodoT {
    string nombre;
    int telefono;
    nodoT * sig;
};

typedef nodoT ** Tabla;
```

```
Tabla crearT(unsigned int n) {
    Tabla res = new nodoT * [n];
    for (int i = 0; i <= n - 1; i++) {
        res[i] = NULL;
    }
    return res;
}

void asociarT(string nom, unsigned int tel, Tabla & t) {
    //busca el nombre y si ya esta modifica el telefono, sino ↩
       lo agrega al comienzo de la lista.
    int indice = h(nom); //h es la funcion de hash

    //tengo que buscar un nodo de nombre nom para modificarlo. ↩
       SI no esta, lo inserto al comienzo.
    nodoT * iter = t[indice];
    while (iter != NULL && iter->nombre != nom) {
        iter = iter->sig;
    }
    if (iter != NULL) { //actualizo
        iter->telefono = tel;
    } else { //inserto al comienzo
        nodoT * nuevo = new nodoT;
        nuevo->nombre = nom;
        nuevo->telefono = tel;
        nuevo->sig = t[indice];
        t[indice] = nuevo;
    }
}
```

El hash abierto como implementación elegida permite cumplir con los requerimientos, porque la función de hash distribuye uniformemente. Por lo tanto al crearse un array de tamaño N, cantidad esperada de asociaciones a almacenar, el factor de carga va a tender a 1 a medida que se van agregando los N elementos. En el caso promedio tanto para la selección como la inserción, las operaciones de comparación serán 1 + factor de carga / 2 (que es lo esperado al ir hasta la mitad de la lista en cada posición). Por lo tanto el orden promedio es O(1).