Instituto de Computación. Facultad de Ingeniería. Universidad de la República

Segundo Parcial Programación 2

Julio de 2022

Problema 1 (35 puntos)

Una empresa desea administrar su stock de productos. Los productos son identificados por un entero no negativo, de tipo *unsigned int*, y poseen un precio, de tipo *float*. Cada producto en el stock puede contener varias unidades. Por ejemplo, del producto 9, cuyo precio es 7.50, hay 20 unidades; del producto 2871, cuyo precio es 9.99, hay 43 unidades; etc. Para administrar el stock se define el TAD ***Stock*** que contiene, entre otras, las siguientes operaciones:

- ***Stock crear (int cantidadEsperada)***, que crea un stock vacío para el cual se estiman hasta *cantidadEsperada* de productos diferentes.
- ***void insertar (Stock & s, unsigned int id, float precio, int cant)***, que incorpora *cant* unidades del producto con identificador *id*, cuyo valor por unidad es *precio*, al stock *s*. Si *id* ya estaba en *s*, suma a las unidades existentes el valor *cant* y reemplaza su valor (por unidad) con *precio*. Se asume como precondición que la cantidad de productos diferentes en *s* es menor a la cantidad esperada, definida cuando se creó *s*.
- ***int unidades (Stock s, unsigned int id)***, que retorna la cantidad de unidades del producto con identificador *id* en el stock *s* (0 si *id* no está en *s*).
- ***void listado (Stock s)***, que imprime los productos del stock *s* ordenados de menor a mayor según su identificador (*id*). Asuma que existe un procedimiento "*void imprimirProducto (unsigned int id)*" que imprime la información del producto identificado con *id*.
- ***int cantidad (Stock s)***, que retorna la cantidad de productos diferentes del stock *s* (0 si *s* está vacío).

a) Asumiendo que todos los identificadores de productos a incorporar al stock son igualmente probables, se pide: **a-1)** defina una representación del TAD ***Stock*** (llamada ***representacionStock***), de tal manera que la operación ***cantidad*** sea O(1) peor caso y las operaciones ***insertar*** y ***unidades*** sean O(1) promedio; **a-2)** justifique brevemente por qué la representación elegida cumple los órdenes exigidos; y **a-3)** implemente las operaciones ***crear*** e ***insertar***, solamente.

b) Sin considerar la parte **a)**, se pide ahora: **b-1)** Defina una representación del TAD ***Stock*** (llamada ***representacionStock***), de tal manera que la operación ***cantidad*** sea O(1) peor caso, ***insertar*** sea O(log(*n*)) peor caso y ***listado*** sea O(*n*) peor caso, donde *n* es la cantidad de productos diferentes en *s* y asumiendo que *imprimirProducto* es O(1) peor caso; **b-2)** justifique brevemente por qué la representación elegida cumple los órdenes exigidos; y **b-3)** implemente la operación ***listado***, solamente.

Problema 2 (18 puntos)

Dada una ***Pila** p **de enteros***, sin elementos repetidos, desarrolle una **implementación iterativa** de la función "***Pila mayoresQueLosAnteriores(Pila & p)***", que retorna una nueva pila, digamos *r*, que contiene cada elemento de *p* que es mayor estricto a todos los ingresados previamente a este elemento en *p.* Notar como caso particular que el primer elemento ingresado a *p* debería estar en *r*. Los elementos en la pila resultado *r* deben estar en orden inverso al orden en el que fueron ingresados los elementos en *p*. Al terminar la ejecución, la pila *p* debe quedar vacía.

Ejemplo (considerando que en *p* y en *r* los elementos fueron ingresados de derecha a izquierda, siendo 13 el último ingresado en *p* y 5 el último ingresado en *r*):
- Pila parámetro *p* = [13, **14**, 6, 2, **8**, 4, **5**]
- Pila resultado *r* = [**5**, **8**, **14**]

El TAD ***Pila de enteros*** contiene solamente las clásicas operaciones: *crear*, *esVacia*, *apilar*, *cima* y *desapilar*.

El orden de tiempo de ejecución de ***Pila mayoresQueLosAnteriores(Pila & p)*** **debe ser O(*n*)**, donde *n* es la cantidad de elementos de *p*. Se asume que se dispone de una implementación del TAD ***Pila*** en la cual todas las operaciones son O(1) peor caso. **No se puede** acceder a la representación del TAD ***Pila*** ni utilizar estructuras de datos o funciones auxiliares.

Instituto de Computación. Facultad de Ingeniería. Universidad de la República

Segundo Parcial Programación 2

Julio de 2022

SOLUCIONES

Problema 1

a-1)

```
struct nodoHash{
        unsigned int id;            // id del producto
        float precio;               // precio por unidad
        int unidades;               // unidades del producto
        nodoHash* sig;              // siguiente en la lista de colisiones
}
struct representacionStock{
        nodoHash** tabla;           // tabla de hash abierta
        int cantidadEsperada;       // cantidad esperada de productos diferentes
        int productosDiferentes;            // cantidad de productos diferentes
}
```

a-2)

La representación elegida involucra una *tabla* de *hash* abierta con una función que dado el *id* de un producto calcula *id*%*M*, donde *M* es la cantidad esperada de productos diferentes (*cantidadEsperada*) y el tamaño de la tabla. De esta manera se cumplen los órdenes exigidos para ***insertar*** y ***unidades***, ya que los identificadores de productos a incorporar al stock son todos igualmente probables y se puede estimar la cantidad esperada de productos diferentes; en promedio cada lista en la tabla tiene entonces un nodo (O(1) promedio). Notar que cada nodo almacena el *id* de un producto, su precio y la cantidad de unidades disponibles de éste. La operación ***cantidad*** tiene O(1) peor caso, ya que en la representación se lleva la cantidad de productos diferentes.

a-3)

```
Stock crear (int cantidadEsperada){
        Stock s = new representacionStock;
                s->tabla = new nodoHash* [cantidadEsperada];
                for (int i=0; i<cota; i++)
                        s->tabla[i]=NULL;
                s->cantidadEsperada = cantidadEsperada;
                s->productosDiferentes = 0;
                return s;
}
void insertar (Stock & s, unsigned int id, float precio, int cant)
        int posicion = id%(t->cantidadEsperada);
        nodoHash* lista = t->tabla[posicion];
        while (lista!=NULL && lista->id!=id)
                lista = lista->sig;
        if (lista==NULL){
                nodoHash* nuevo = new nodoHash;
                nuevo->id = id;
                nuevo->precio = precio;
                nuevo->unidades = cant;
                nuevo->sig = t->tabla[posicion];
                t->tabla[posicion] = nuevo;
                t->productosDiferentes++;
        }
        else{   lista->precio = precio;
                lista->unidades += cant;
        }
    }
```

Segundo Parcial Programación 2

Julio de 2022

b-1)

```
struct nodoAVL{
        unsigned int id;            // id del producto
        float precio;               // precio por unidad
        int ocurrencias;            // unidades del producto
        int altura                          // altura del árbol con este nodo como raíz
        nodoAVL * izq, * der;       // subárboles izquierdo y derecho
}
struct representacionStock{
        nodoAVL * avl;              // árbol AVL de productos
        int productosDiferentes;            // cantidad de productos diferentes
        int cantidadEsperada;       // opcional: cantidad esperada de productos diferentes
}
```

b-2)

La representación elegida involucra un AVL de productos; cada nodo almacena en particular el *id* de un producto (clave para el orden en el AVL), su precio y la cantidad de unidades disponibles de éste. De esta manera se cumplen los órdenes exigidos para ***insertar*** y ***listado***, ya que la inserción involucra el recorrido de un camino del árbol que en el peor caso tiene O(log(*n*)) nodos y el listado es un recorrido *en orden* del árbol, que visita sus *n* nodos (O(*n*) peor caso). La operación ***cantidad*** tiene O(1) peor caso, ya que en la representación se lleva la cantidad de productos diferentes.

b-3)

```
void listadoAVL (nodoAVL* t){
        if (t!=NULL){
                listadoAVL(t->izq);
                imprimirProducto(t->id);
                listadoAVL(t->der);
        }
}
void listado (Stock s){
        listadoAVL(s->avl);
}
```

Problema 2

```
Pila mayoresQueLosAnteriores(Pila & p){
        Pila r = crear();
        while (! esVacia(p)){
                int c = cima(p);
                while (! esVacia(r) && cima(r) < c)
                        r = desapilar(r);
                r = apilar(c, r);
                p = desapilar(p);
        }
        return r;
}
```

Lo que sigue es una justificación intuitiva de por qué se cumple el orden pedido,
Se puede ver que cada elemento de *p* se apila una vez en *r* y por lo tanto se desapila de *r* a lo sumo una vez. Además por cada iteración del while interno o bien se desapila un elemento de *r* (cuando la condición se cumple), o bien se apila un elemento en *r* (cuando la condición no se cumple). Por lo tanto la cantidad de veces que se evalúa la compara-

ción del while interno es a lo sumo 2 *n*. Entonces por cada elemento de *p* se hace una cantidad máxima fija de operaciones, cada una de las cuales es O(1). Por lo tanto el costo de procesar los *n* elementos es O(*n*).

Otra solución:

```
Pila mayoresQueLosAnteriores(Pila & p){
    Pila r = crear();
    if (! esVacia(p)){
        do {
            apilar(cima(p), r);
            desapilar(p);
        } while (! esVacia(p));
        apilar(cima(r), p);
        desapilar(r);
        while (! esVacia(r)) {
            if (cima(r) > cima(p) {
                apilar(cima(r), p);
            }
            desapilar(r);
        }
        while (! esVacia(p)) {
            apilar(cima(p), r);
            desapilar(p);
        }
    }
    return r;
}
```