Examen de Programación 2
15 de diciembre de 2016

Generalidades:

a) La prueba es individual y sin material.
b) La duración es **3**hs.
c) Sólo se contestan dudas acerca de la letra.
d) Escriba las hojas de un solo lado y con letra clara.
e) Numere cada hoja, indicando en la primera el total.
f) Escriba número de cédula y nombre en cada hoja.

Ejercicio 1 (50 puntos)

Se quiere determinar cuáles son los productos más comprados en un supermercado. Para esto se cuenta con un TAD `Compra` que representa los pares *(codigo, cantidad)*, donde código representa un código alfanumérico de producto y cantidad es la cantidad de veces que dicho producto fue comprado. En particular, en el TAD `Compra`, se definen las operaciones selectoras `obtenerCodigo` y `obtenerCantidad`. Además se cuenta con el TAD `ListaCompra`, con las operaciones usuales del TAD Lista, para modelar la lista de compras. *Recuerde que la inserción se hace al comienzo.*

Se quiere implementar una función `ListarKMasComprados`, que dada una lista de productos comprados representada con el TAD `ListaCompra` y un entero **k**, devuelve otra `ListaCompra` con los **k** pares *(codigo, cantidad)* que corresponden a los k productos más comprados. Estos pares se devuelven ordenados en forma decreciente según cantidad de compras. No hay un criterio de desempate preestablecido en caso de empates en la cantidad de compras, quedando esto librado a la implementación. Si **k** es mayor que el largo de la lista se devuelven todas las compras. El encabezado de la función `ListarKMasComprados` es el siguiente:

```
ListaCompra ListarKMasComprados (ListaCompra lst; unsigned int k);
```

A continuación se presentan algunos ejemplos de ejecuciones.

```
lst =[(GALL03,3),(POLLO99,2),(PAN35,3),(LECH07,5),(YOGU55,3)]

ListarKMasComprados(lst,0)= []
ListarKMasComprados(lst,1)= [(LECH07,5)]
ListarKMasComprados(lst,2)= [(LECH07,5),(YOGU55,3)] o
                            [(LECH07,5),(GALL03,3)] o
                            [(LECH07,5),(PAN35,3)]
```

(a) (15 puntos) Especifique el TAD `PrioridadCompra` como una Cola de Prioridad acotada de elementos del tipo `Compra`, donde un elemento tiene prioridad sobre otro si el valor de su campo **cantidad** es menor. El tamaño de la Cola de Prioridad se define al momento de crearla. Incluya pre y post condiciones.

(b) (20 puntos) Implemente `ListarKMasComprados` usando una estructura auxiliar de tipo `PrioridadCompra` de tamaño k, las operaciones especificadas en (a) y las de los TADs `Compra` y `ListaCompra`. La lista original **lst** se puede recorrer una sola vez.

(c) (15 puntos) Suponiendo que **k** es acotado por una constante **MAX_K** , explique qué implementación del TAD `PrioridadCompra` de la parte (a) permite realizar `ListarKMasComprados` en O(n * log k), donde n es la cantidad de productos. Describa propiedades de la implementación propuesta y tiempos de ejecución en el peor caso para cada una de las operaciones especificadas.

Solución:

Primeramente, comenzamos especificando el TAD ListaCompra que es común a todo el ejercicio.

```
#include "TADCompra.h"

struct nodo;
typedef nodo * ListaCompra;

/* Crea la lista vacia. */
ListaCompra crearLC();

/* Inserta la compra c al principio de la lista. */
ListaCompra insertarLC(Compra c, ListaCompra l);

/* Verifica si la lista esta vacia. */
bool esVaciaLC( ListaCompra l);

/* Devuelve el primer elemento de la lista. Pre: !esVaciaLC(l) */
Compra cabezaLC(ListaCompra l);

/* Devuelve el resto de una lista. Pre: !esVaciaLC(l) */
ListaCompra colaLC(ListaCompra l);
```

(a)
```
/* Retorna la cola de prioridad vacia de tamanio tam
Prencodicion: tam > 0
*/
PrioridadCompra crearPC (unsigned int tam);

/* Verifica si la cola de prioridad cp esta vacia */
bool esVaciaPC(PrioridadCompra pc);

/* Inserta la compra c con prioridad obtenerCantidad(c)
Precondicion: !estaLlenaPC(c)
*/
void insertarPC(Compra c, unsigned int prio, PrioridadCompra &pc);

/* Quita de la cola de prioridad cp el elemento prioritario.
Precondicion: ! esVaciaPC(cp). */
void eliminarPrioritarioPC(PrioridadCompra &pc);

/* Retorna el elemento prioritario de cp.
Precondicio: ! esVaciaPC(cp). */
Compra prioritarioPC(PrioridadCompra pc);

/* Devuelve true si la cola esta llena, false en caso contrario */
bool estaLlenaPC(PrioridadCompra pc);

/* Libera la memoria de la cola y sus elementos */
void destruirPC(PrioridadCompra &pc);
```

(b)
```
ListaCompra ListarKMasComprados( ListaCompra lst, unsigned int k){

PrioridadCompra pc = crearPC(k);

while (!esVaciaLC(lst)) {

        Compra c = cabezaLC(lst);
        lst = colaLC(lst);

        if (estaLlena(pc)) {

                Compra compraMaxPrio = prioritarioPC(pc);
                if (obtenerCantidad(c) > obtenerCantidad(compraMaxPrio)){
                        eliminarPrioritarioPC(pc);
                        insertarPC(c,obtenerCantidad(c),pc)
                }
        } else
                insertarPC(c,obtenerCantidad(c),pc);
};

ListaCompra res = crearLC();

while (!esVaciaPC(pc)){

        Compra c = prioritarioPC(pc);
        eliminarPrioritarioPC(pc);

        insertarLC(res,c);
}

destruirPC(pc);

return res;
}
```

(c) Podemos implementar la cola de prioridad como un heap. El heap es un árbol parcialmente ordenado y balanceado. Por la propiedad de orden cada nodo tiene prioridad sobre sus descendientes. El balanceo consiste en que está completo salvo quizás el último nivel que se llena de izquierda a derecha. Esto permite implementarlo con un arreglo en el que el padre y los hijos de cada nodo se encuentran en $O(1)$. El tamaño se asigna de forma dinamica en el constructor.

El orden de inserción en el heap es log k. La operación ListarKMasComprados comienza recorriendo una sola vez cada elemento de la lista (O(n)) y lo inserta en el heap (O (log k)). En el peor de los casos la cola siempre está llena y la compra a insertar tiene mayor cantidad que el de maxima prioridad, por lo que también debemos eliminar el elemento más prioritario en O (log k). Hasta este punto tenemos que el orden es n * (log k + log k). Luego, por cada elemento de la cola de prioridad O(k) se obitene el elemento más prioritario (O(1)) y se lo elimina en O(log k), entonces esto es k * log k. Finalmente tenemos que el orden total del procedimiento es n * (2 * log k) + k * log k, lo cual es O (n * log k). Notar que las operaciones de cabezaLC y colaLC son O(1). Además, destruirPC destruye una cola vacía, por lo que es O(1).

Los tiempos de ejecución del TAD PrioridadCompra son:

- crearPC : O(1) se pide memoria para los k elementos de la cola y se la marca como vacía.
- esVaciaPC : O(1) se consulta la marca de si es vacía o no la cola.

- insertarPC : O(log k) se inserta en el heap (arreglo) como último elemento y se realiza el filtrando ascendente.
- eliminarPrioritarioPC : O(log k) se sustituye el elemento más prioritario por el de menos prioridad y se realiza el filtrado descendente.
- prioritarioPC : O(1) se devuelve el primer elemento de arreglo.
- destruirPC : O (k) se recorre todo el arreglo liberando la memoria.

Ejercicio 2 (50 puntos)

(a) Considere la siguiente implementación del tipo `Arbol`, de árboles binarios de un tipo genérico `T`, donde el árbol vacío se implementa con `NULL`.

```
struct nodo {
  T dato;
  nodo *izq, *der;
};
typedef nodo *Arbol;
```

I. (10 puntos) Implemente la función `int Altura(Arbol t)`, que devuelve la altura del árbol `t`. La altura del árbol vacío es 0. A modo de ejemplo considere el árbol de la Figura 1, en el cual una llamada a la función `Altura` sobre el árbol que tiene raíz en `F` devuelve 1, mientras que para el que tiene raíz en `D` devuelve 3.

II. (15 puntos) Implemente la función `bool EsNodoBalanceado(Arbol t)`, que devuelve `true` si y sólo si la raíz de `t` cumple la condición de balanceo de los árboles `AVL`. Se considera que el árbol vacío está balanceado. A modo de ejemplo, una llamada a la función `EsNodoBalanceado` sobre los arboles de la Figura 1 que tienen como raíz a `A` y a `E` debe devolver `false`.

(b) (25 puntos) Considere ahora la siguiente implementación del tipo `Arbol`, donde se agrega un atributo `es_arbol_balanceado` que es `true` si y sólo si el nodo y cada uno de sus descendientes cumplen la condición de balanceo de los árboles `AVL`.

```
struct nodo {
  T dato;
  bool es_arbol_balanceado;
  nodo *izq, *der;
};
typedef nodo *Arbol;
```

Implemente el procedimiento `EstablecerBalanceo(Arbol & t)`, que establece el valor correcto del atributo `es_arbol_balanceado` para cada nodo de `t`. **Su solución puede visitar cada nodo sólo una vez.** A modo de ejemplo, en el árbol de la Figura 1, el atributo `es_arbol_balanceado` debe ser `false` en los nodos `A`, `B` y `E`.

Observaciones:

- La implementación de todas las funciones se debe hacer accediendo a la representación del tipo `Arbol`.
- En ambas partes se puede asumir que están definidas las funciones `max` (que devuelve el máximo de dos enteros) y `abs` (que devuelve el valor absoluto de un entero).

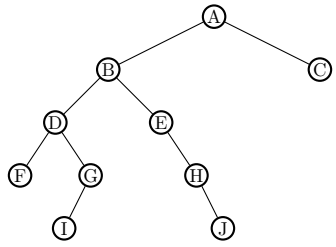

Figura 1: Ejemplo para el Ejercicio 2

Solución:

(a) I.
```
int Altura(Arbol t) {
  int result;
  if (t == NULL)
    result = 0;
  else
    result = 1 + max(Altura(t->izq), Altura(t->der));
  return result;
}
```

II. Un nodo de un árbol binario cumple la condición de balanceo de los árboles `AVL` si la diferencia de las **alturas** de los subárboles izquierdo y derecho no es mayor que 1.

```
bool EsNodoBalanceado(Arbol t) {
  return (t == NULL) || (abs(Altura(t->izq) - Altura(t->der)) <= 1);
}
```

Se debe incluir el operando `(t == NULL)` para, además de devolver `true` cuando el árbol es vacío, poder usar `t->izq` y `t->der` como consecuencia de la semántrica del operador `||`.

(b) Para asignar `true` al atributo `es_arbol_balanceado` de un nodo es condición necesaria que a sus hijos (en caso de no ser vacíos) también se les asigne `true`. Esto podría resolverse con llamadas recursivas. Pero dicha condición no es suficiente, ya que además el propio nodo debe estar balanceado. Esta información no se puede obtener de las llamadas recursivas ya que sólo modifican el árbol y devuelven `void`. Para resolverlo no se puede invocar a la función `Altura` porque esta función recorre todos los nodos del árbol, lo que haría incumplir el requerimiento de visitar cada nodo sólo una vez. Por la misma razón no se puede invocar `EsNodoBalanceado` porque se estaría llamando a `Altura` de manera indirecta.

La solución se obtiene definiendo una función auxiliar que además de modificar el árbol asignando el campo `es_arbol_balanceado` devuelva la altura del árbol.

```
void aux_balanceo(Arbol & t, int & altura) {
  int h_izq, h_der;
  if (t == NULL)
    altura = 0;
  else {
    aux_balanceo(t->izq, h_izq);
    aux_balanceo(t->der, h_der);
    altura = 1 + max(h_izq, h_der);
    t->es_arbol_balanceado =
        ((t->izq == NULL) || t->izq->es_arbol_balanceado) &&
        ((t->der == NULL) || t->der->es_arbol_balanceado) &&
        abs(h_izq - h_der) <= 1;
  }
}

void EstablecerBalanceo(Arbol & t) {
  int altura;
  aux_balanceo(t, altura);
}
```