Instituto de Computación - Facultad de Ingeniería, Universidad de la República

Examen Programación 2 - Soluciones

17 de Febrero de 2017

Generalidades:

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

Problema 1 (a) 10 puntos b) 35 puntos)

Considere el TAD *ColaPrio* que es una cola de prioridad no acotada, donde los elementos guardados son códigos alfanuméricos (o *strings*) y la prioridad de cada uno está dada por un entero en el rango *[0, K]*, donde un menor valor indica una mayor prioridad. *K* es una constante.

Además de las operaciones clásicas del TAD, se debe contar con la operación **`listaOrdenada`** que devuelve la lista de códigos alfanuméricos ordenados de mayor a menor prioridad de una *ColaPrio* (los códigos que tienen asociada la misma prioridad deben quedar en el orden en que fueron insertados).

Se pide:

a) Especifique completamente el TAD *ColaPrio* (con pre y post condiciones).

b) Implemente el TAD definiendo el tipo para la representación elegida y desarrollando solamente el código de las operaciones constructoras y de **`listaOrdenada`**, de forma tal que la operación selectora tenga O(1) de tiempo de ejecución en el peor caso y **`listaOrdenada`** tenga O(n) de tiempo de ejecución en el peor caso, donde *n* es la cantidad de elementos de la cola. **Explique cómo la implementación elegida permite cumplir los requerimientos.**

Se puede suponer que los códigos alfanuméricos se pueden asignar y comparar como los tipos básicos. Para la implementación de **`listaOrdenada`** se dispone del *TAD ListaStr* que tiene las siguientes operaciones básicas (asuma que están implementadas):

```
// Crea y devuelve una lista vacía
ListaStr vaciaLStr();
// Inserta x al principio de l
ListaStr insertarIni(ListaStr l, string x);
// Devuelve el primer elem de l no vacía
string primeroLStr(ListaStr l);
// Devuelve la lista l sin el primer elemento
ListaStr restoLStr(ListaStr l);
// Devuelve true si l es vacía
bool esVaciaLStr(ListaStr l);
```

También se puede suponer que todas las operaciones del *TAD ListaStr* tienen tiempo de ejecución O(1) en el peor caso.

Problema 2 (a) 10 puntos b) 25 puntos c) 20 puntos)

Considere el TAD *ConcCarnaval* que permite registrar los nombres de las agrupaciones de carnaval y para cada una su puntaje en el concurso. Es decir que este TAD es una función parcial entre *strings* del dominio (posibles nombres de los agrupaciones de carnaval) y *enteros* del codominio (puntaje actual de cada agrupación).

Se pide:

a) Especifique completamente el TAD *ConcCarnaval* (con pre y post condiciones), que tenga al menos operaciones para:

- Crear un *ConcCarnaval* vacío
- Asociar a *agrup* el valor *punt* en un *ConcCarnaval*; si ya tenía un valor asociado, lo redefine
- Consultar si un *agrup* tiene asociado un puntaje en un *ConcCarnaval*
- Devolver puntaje del codominio asociado a *agrup* en un *ConcCarnaval*
- Eliminar la asociación de *agrup* en un *ConcCarnaval.*

b) Proponga una implementación del TAD *ConcCarnaval*, sin usar TADs auxiliares y asumiendo que:

- cualquier string puede ser un nombre de agrupación
- es igualmente probable que se desee asociar un puntaje a cualquier agrupación
- la cantidad de asociaciones a almacenar puede estimarse en un valor *N*
- los nombres de las agrupaciones 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]*).

Defina el tipo para la representación elegida del TAD y desarrolle sólo el código de la operación para crear un *ConcCarnaval* vacío y para devolver el puntaje asociado a una agrupación, de modo que las operaciones selectoras y la operación de inserción tengan O(1) de tiempo de ejecución en el caso promedio. **Explique cómo la implementación elegida permite cumplir con los requerimientos.**

c) Usando las operaciones del TAD *ConcCarnaval* de la parte a) implemente la operación **`incEscalonado`** que, dada una lista de nombres de agrupaciones y un *ConcCarnaval,* incrementa de forma escalonada el puntaje de las agrupaciones de la lista, es decir que a la primera agrupación le incrementa el puntaje en 1, a la segunda en 2, a la tercera en 3 y así sucesivamente (para aquellas agrupaciones que tengan asociado un puntaje en el *ConcCarnaval*):

```
/* Dados una lista de nombres de agrupaciones l (de tipo ListaStr)
y un ConcCarnaval cc, incrementa el puntaje de forma escalonada de
las agrupaciones de la lista l que si tienen asociado un puntaje
en cc. */
void incEscalonado(ListaStr l, ConcCarnaval & cc);
```

Ejemplo (el orden de las asociaciones no necesariamente corresponde al que se mantiene en **`cc`**):

Entradas	Resultado
cc = [(“La Lunática”, 3), (“Araca la Cana”, 5), (“Momosapiens”, 4), (“La Clave”, 3)] l = [“Momosapiens”, “La Clave”, “Don Timoteo”, “Araca la Cana”]	**cc** = [(“La Lunática”, 3), (“Araca la Cana”, 9), (“Momosapiens”, 5), (“La Clave”, 5)]

Se debe usar el *TAD ListaStr* del Ejercicio 1).

Soluciones

Problema 1 (a) 10 puntos b) 35 puntos)

Parte a)

```
/*************** Constructoras ***************/
/* Devuelve la cola de prioridad vacía */
ColaPrio crearPQ();

/* Pre: 0 <= prio <= K. Devuelve una nueva cola de prioridad que consiste
de todos los elementos de cp más el elemento codigo con su prioridad */
ColaPrio insertarPQ (string codigo, unsigned int prio, ColaPrio cp);

/*************** Predicados ***************/
/* Devuelve true si y sólo si cp es vacía */
bool esVaciaPQ (ColaPrio cp);

/*************** Selectoras ***************/
/* Pre: !vaciaPQ (cp). Devuelve el elemento más prioritario de cp */
string minPQ(ColaPrio cp);

/* Devuelve la lista de códigos ordenados de mayor a menor prioridad de pc,
los códigos que tienen asociada la misma prioridad quedan en el orden en
que fueron insertados */
ListaStr listaOrdenada(ColaPrio pc);

/*************** Destructoras ***************/
/* Pre: !vaciaPQ (cp). Devuelve cp sin el elemento más prioritario */
ColaPrio removerPQ(ColaPrio cp);
```

Parte b)

```
typedef struct repPQ{
     ListaStr nodos [K+1];
     unsigned int pos_min;
}* ColaPrio;

ColaPrio crearPQ(){
     ColaPrio cp;
     cp = new repPQ;
     for (int i = 0; i <= K; i++)
          cp->nodos[i] = vaciaLStr();
     cp->pos_min = K + 1;
     return cp;
}

ColaPrio insertarPQ (string codigo, unsigned int prio, ColaPrio cp){
     insertarIni(cp->nodos[prio], codigo);
     if (prio < cp->pos_min)
```

```
        cp->pos_min = prio;
      return cp;
}

ListaStr listaOrdenada(ColaPrio cp){
      // El array se recorre en orden decreciente de la posición
      // ya que se inserta al principio de la lista resultado
      ListaStr lstPos;
      ListaStr lista = vaciaLStr();

      for(int prio = K; prio >= 0; prio--){
            lstPos = cp->nodos[prio];
            while (! esVaciaLStr(lstPos)){
              insertarIni(lista, primeroLStr(lstPos));
              lstPos =  restoLStr(lstPos);
            };
      };
      return lista;
}
```

La implementación elegida es un array con índices en [0,K] donde en cada posición hay una ListaStr cuyos elementos tienen prioridad igual a dicha posición. Además se mantiene el campo **pos_min**, que indica cuál es la menor posición con una lista no vacía.

El tiempo de ejecución en el peor caso de la operación selectora **minPQ** está en O(1) ya que mediante **pos_min** se determina cuál es la lista en que está el elemento buscado. Una implementación alternativa es no mantener el campo **pos_min** y buscar la menor posición que tenga una lista no vacía, lo cual está en O(1) dado que K es una constante.

El tiempo de ejecución en el peor caso (y en todos los casos) de **listaOrdenada** está en O(n), dado que se debe construir la lista resultado mediante la recorrida de los **n** elementos de las listas del array **nodos** y **n** inserciones (cada una de estas operaciones se hace en un tiempo que está en O(1)), y el recorrido de las posiciones del array está en O(1) (porque que K es constante). El array **nodos** se procesa desde la posición K hasta la 0 para que (como consecuencia de que la inserción es al inicio de la lista) la lista resultado quede ordenada de mayor a menor prioridad (mayor prioridad significa estar asociado a un menor valor de **prio** y por lo tanto menor posicion en el array). Hay que notar que cada lista del array **nodos** está en orden inverso del tiempo de inserción, lo cual hace que al recorrerlas e insertar en la lista resultado se obtenga una lista ordenada de la forma pedida.

Observación

Una lista ordenada de todos los códigos con sus respectivas prioridades también permitiría cumplir con los requerimientos. Para esto se necesita definir una lista en cuyos nodos se mantenga, además del código, la prioridad asociada.

Problema 2 (a) 10 puntos b) 25 puntos c) 20 puntos)

Parte a)

```
// Crea y devuelve un ConcCarnaval vacío
ConcCarnaval crearCC();

/* Asocia a agrup el puntaje punt en cc; si ya tenía un valor asociado, lo
redefine */
void insertarAgrupEnCC(string agrup, int punt, ConcCarnaval & cc);

// Devuelve true sii agrup tiene asociado un puntaje en cc
bool agrupEnCC(string agrup, ConcCarnaval cc);

/* Devuelve el puntaje asociado a la agrupacion agrup en cc
Pre: agrupEnCC(agrup, cc) */
int recuperarPuntCC(string agrup, ConcCarnaval cc);

/* Modifica cc de modo que agrup no tenga un puntaje asociado.
Si !agrupEnCC(agrup, cc), la operación no tiene efecto */
void eliminarAgrupCC(string agrup, ConcCarnaval & cc);
```

Parte b)

```
struct nodoCc {
    string agrup;
    int punt;
    nodoCc * sig;
};

typedef nodoCc * ListaCc;

struct cabezalCc {
    ListaCc * hashCc;
};

typedef cabezalCc * ConcCarnaval;

ConcCarnaval crearCC() {
   ConcCarnaval cc;
   cc = new cabezalCc;
   cc->hashCc = new ListaCc[N];
   for (int i = 0; i < N; i++)
      cc->hashCc[i] = NULL;
   return cc;
};
```

```
/* Devuelve el puntaje asociado a la agrupacion agrup en cc
Pre: agrupEnCC(agrup, cc) */
int recuperarPuntCC(string agrup, ConcCarnaval cc) {
   ListaCc lst = cc->hashCc[h_N(agrup)];

   while (lst->agrup != agrup)
      lst = lst->sig;
   return lst->punt;
};
```

El hash abierto como implementación elegida permite cumplir con los requerimientos, porque al crearse un array (ListaCc) de tamaño N, que es la cantidad esperada de asociaciones a almacenar, el factor de carga se vuelve 1. En el caso promedio tanto para la selección (agrupEnCC, recuperarPuntCC) 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 que es O(1).

Parte c)

```
/* Dados una lista de nombres de agrupaciones l (de tipo ListaStr) y un
ConcCarnaval cc, incrementa el puntaje de forma escalonada de las
agrupaciones de la lista l que si tienen asociado un puntaje en cc. */
void incEscalonado(ListaStr l, ConcCarnaval & cc) {
   int incr = 1;
   int  punt_viejo;   // puede ser más eficiente definir  estas dos
   string agrup;      // variables auxiliares dentro del while

   while (!esVaciaLStr(l)) {
      agrup = primeroLStr(l);
      if (agrupEnCC(agrup, cc)) {
         punt_viejo = recuperarPuntCC(agrup, cc);
         insertarAgrupEnCC(agrup, punt_viejo + incr, cc);
      };
      incr++;
      l = restoLStr(l);
   };
}
```