Examen de Programación 2
13 de diciembre de 2019

Problema 1 (20 puntos)

Implemente una función iterativa que remueve todas las ocurrencias de un elemento en una lista ordenada.

```
/* Remueve todas las ocurrencias de 'rem' en 'l'.
   Si 'rem' no está en 'l' la operación no tiene efecto.
   Precondición: 'l' está ordenada de manera no decreciente. */
void removerTodos(int rem, Lista &l);
```

Solución:

Usamos la representación habitual de listas en donde la lista vacía es `NULL`.

```
struct rep_lista {
  int dato;
  rep_lista * sig;
};
typedef rep_lista * Lista;
```

La solución debe aprovechar el conocimiento de que la lista está ordenada, por lo que si `rem` es menor al elemento no se debe seguir la recorrida.

La lista se recorre con otra variable de tipo puntero, `cursor`. La recorrida no se puede hacer con `l` porque es pasado por referencia. Cuando se encuentra un nodo a remover el puntero debe apuntar al nodo anterior al que va a ser removido.

La siguiente es una solución en la que se crea un nodo auxiliar que es removido al final. El propósito es poder tratar el caso en que los elementos a remover están al principio de la lista (lo cual implica modificar el parámetro) sin hacer código específico para ese caso. Se debe notar que esta solución es correcta incluso cuando la lista es vacía.

```
void removerTodos(int rem, Lista &l) {
  rep_lista *ancla = new rep_lista;
  ancla->sig = l;
  rep_lista *cursor = ancla;
  while ((cursor->sig != NULL) && (cursor->sig->dato < rem))
    cursor = cursor->sig;
  while ((cursor->sig != NULL) && (cursor->sig->dato == rem)) {
    rep_lista *aborrar = cursor->sig;
    cursor->sig = cursor->sig->sig;
    delete aborrar;
  }
  l = ancla->sig;
  delete ancla;
}
```

Otra solución, en la que se trata separado el caso en que se deben remover los nodos del inicio:

```
void removerTodos(int rem, Lista &l) {
  while ((l != NULL) && (l->dato == rem)) {
    rep_lista *aborrar = l;
    l = l->sig;
    delete aborrar;
  }
  // l puede ser o haber quedado vacía
  if (l != NULL) {
    rep_lista *cursor = l;
    // l->dato != rem.
    // Si l->dato > rem no se entra al cuerpo de ninguno de los dos ciclos
    while ((cursor->sig != NULL) && (cursor->sig->dato < rem))
      cursor = cursor->sig;
```

```
      while ((cursor->sig != NULL) && (cursor->sig->dato == rem)) {
        rep_lista *aborrar = cursor->sig;
        cursor->sig = cursor->sig->sig;
        delete aborrar;
      }
    }
  }
```

Problema 2 (25 puntos)

Considere la siguiene función:

```
void funcion(int n, int A[], int B[]) {
  for (int i = 0; i < n; i++) {
    B[i] = A[0];
    for (int j = 1; j <= i; j++)
      if (A[j] < B[i])
        B[i] = A[j];
  }
}
```

(a) ¿Qué obtiene esta función?

(b) Calcule la cantidad de asignaciones al arreglo `B` en el peor caso. ¿Cuál es el orden O del tiempo de ejecución?

(c) Modifique la función para que el orden O del tiempo de ejecución sea menor. ¿Cuál es el nuevo orden?

Solución:

(a) Para cada $i, 0 \leq i < n$ calcula en $B[i]$ el mínimo del segmento $A[0 \cdots i]$.

(b) La cantidad de asignaciones es máxima cuando la condición `A[j] < B[i]` siempre se cumple. Esto ocurre si el arreglo `A` está ordenado de manera decreciente estricta. El bucle exterior se ejecuta con `i` desde 0 hasta n - 1. En cada iteración se hace la asignación `B[i] = A[0]` y luego se ejecuta un bucle con `j` desde 1 hasta `i` en el que hay una asignación al arreglo `B` en cada iteración.

$$\sum_{i=0}^{n-1} \sum_{j=0}^{i} 1 = \sum_{i=0}^{n-1} (i+1) = \sum_{h=1}^{n} h = \frac{n(n+1)}{2}.$$

Por cada asignación al arreglo `B` hay una cantidad $O(1)$ de otras operaciones: una comparación entre elementos de los arreglos, una comparación entre índices, un incremento de índices. Por lo tanto la cantidad de asignaciones al arreglo `B` es, a menos de un factor constante, una cota superior del tiempo de ejecución.

Por lo que el tiempo de ejecución es $O(n^2)$.

(c)

```
void minimos(int n, int A[], int B[]) {
  B[0] = A[0];
  for (int i = 1; i < n; i++)
    B[i] = (A[i] < B[i - 1]) ? A[i] : B[i - 1];
}
```

En cada paso se utiliza el resulado obtenido en el paso anterior. El tiempo de ejecución es $O(n)$.

Problema 3 (25 puntos)

Implemente una función recursiva que a todos los nodos de un nivel dado de un árbol general le agrega como primer hijo un elemento pasado como parámetro. La raíz está en el nivel 1.

```
/* Precondición: nivel > 0. */
void insertar(int elem, int nivel, Arbol &a);
```

Solución:

Usamos la representación habitual de árboles generales en donde el árbol vacío es `NULL`.

```
struct rep_arbol {
  int dato;
  rep_arbol * pH, * sH;
};
typedef rep_arbol * Arbol;

void insertar(int elem, int nivel, Arbol &a) {
  if (a != NULL) {
    if (nivel > 1)
      insertar(elem, nivel - 1, a->pH);
    else {
      Arbol nuevo = new rep_arbol;
      nuevo->dato = elem;
      nuevo->pH = NULL;
      nuevo->sH = a->pH;
      a->pH = nuevo;
    }
    insertar(elem, nivel, a->sH);
  }
}
```

Se recorre el árbol buscando los nodos del nivel pedido. Esta recorrida se hace mediante dos invocaciones recursivas por cada nodo. Los casos base de la recusión son cuando el árbol es vacío o cuando se alcanza el nivel buscado. El siguiente hermano es un árbol que está en el mismo nivel, por lo que en la invocación correspondiente el parámetro `nivel` no se modifica. Para la invocación al primer hijo el parámtero `nivel` se decrementa en 1 porque los nodos que están en el nivel `nivel` del árbol `a` están en el nivel `nivel-1` del árbol `a->pH`. Si se encuentra el nivel buscado se interrumpe la recursión en el primer hijo pero no en el siguiente hermano. En ese caso en lugar de la recursión en el primer hijo se inserta un nuevo nodo como primero en la lista de hijos del nodo encontrado. El nuevo nodo es un árbol y debe tener correctamente definidos sus dos punteros.

Problema 4 (30 puntos)

La empresa de entrega de mercaderías *DeliveryYa* necesita gestionar su *callcenter*, al cual se comunican sus clientes para hacer los pedidos y que es atendido por una plantilla de operadores de la empresa.

Cada cliente es identificado por su teléfono (*int*) y no se sabe cuántos clientes pueden estar en espera en cada momento. La empresa tiene operadores que atienden a los clientes y cada uno puede atender hasta un cliente por vez. Los operadores son identificados por un número de operador (*int*) que no se repite. Cada vez que un cliente quiere hacer un pedido es transferido con el operador que ha estado libre durante más tiempo o es puesto en espera hasta que un operador quede libre.

(a) Especifique el `TAD Cola` que sirve tanto para mantener clientes en espera como operadores con disponibilidad para atender clientes.

(b) Defina una representación para `Cola` que permita que el orden del tiempo de ejecución de las operaciones sean $O(1)$ peor caso. Además el espacio ocupado en memoria debe ser proporcional a la cantidad de elementos almacendos en la cola en todo momento. Dar especificación de tipos y explicación gráfica, justificando brevemente por qué la representación elegida cumple con las restricciones de tiempo.

(c) Implemente todas las operaciones del `TAD`.

(d) Implemente la siguiente operación:

```
/* Dada una cola de clientes en espera ECli y una cola de operadores libres EOper,
 se intenta asignar un operador a un cliente.
Si hay operadores libres y clientes en espera: el operador que ha estado libre más
tiempo debe ser asignado al cliente que ha esperado más tiempo y se devuelve true,
en caso contrario se devuelve false.
Si se puede hacer una asignación de un operador a un cliente se debe devolver
en cliAtendido el número de teléfono del cliente atendido y
en operAsignado el número de operador asignado.
Se deben remover cliAtendido y operAsignado de sus colas correspondientes.
 */

bool asignarUnOperador(Cola & ECli, Cola & EOper, int & cliAtendido,
int & operAsignado);
```

Solución:

(a) El TAD Cola con todas las operaciones clásicas.

```
struct RepresentacionCola;
typedef RepresentacionCola* Cola;

void crearCola (Cola & e);
/* Devuelve en e la cola de Cola vacía. */

void encolar (int n, Cola &e);
/* Agrega el elemento n al final de e. */

bool esVaciaCola (Cola e);
/* Devuelve 'true' si e es vacia, 'false' en otro caso. */

int frente (Cola e);
/* Devuelve el primer elemento de e
Precondicion: ! esVaciaCola(e). */

void desencolar (Cola &e);
/* Remueve el primer elemento de e.
Precondicion: ! esVaciaCola(e). */

void destruirCola (Cola &e);
/* Libera toda la memoria ocupada por e. */
```

(b)

```
typedef struct RepresentacionCola{
  int* cola;
  int cantidad;
```

```
    int primero;
    int ultimo;
  }
```

encolar: es O(1) porque se inserta luego del último elemento al cual se tiene referencia.

esVaciaCola: es O(1) porque sólo se evalúa si cantidad es igual a 0.

frente: es O(1) porque se tiene referencia al primer elemento.

desencolar: es O(1) porque se tiene referencia al primer elemento.

(c)

```
struct RepresentacionCola;
typedef RepresentacionCola* Cola;

void crearCola (Cola & e, int N){
  Cola e    = new Cola();
  e->cola     = new int(N);
  e->cantidad   = 0;
  e->primero    = 0;
  e->ultimo     = 0;
  e->capacidad  = N;
}

/* Agrega el elemento n al final de e. */
void encolar (int n, Cola &e){
  if (e->cantidad < e->capacidad){
    e->cantidad++;
    if (e->cantidad == 1){
      e->ultimo  = 1;
      e->primero = 1;
    }else{
      e->ultimo = (e->ultimo + 1) %e->capacidad;
    }
    e->cola[e->ultimo] = n;
  }
}

/* Devuelve 'true' si e es vacia, 'false' en otro caso. */
bool esVaciaCola (Cola e){
  return (e->cantidad == 0);
}

/* Devuelve el primer elemento de e
Precondicion: ! esVaciaCola(e). */
int frente (Cola e){
  return e->cola[e->primero];
}

/* Remueve el primer elemento de e.
Precondicion: ! esVaciaCola(e). */
void desencolar (Cola &e){
  e->cantidad--;
  if (e->cantidad > 0){
    if (e->primero == 0){
      e->primero = e->capacidad - 1;
    }else{
      e->primero--;
    }
  }
}

/* Libera toda la memoria ocupada por e. */
void destruirCola (Cola &e){
  delete [] e->cola;
  delete e;
}
```

(d)
```
Cola ECli, EOper;
crearCola (ECli, N);
crearCola (EOper, M);
```

(e)
```
/* Dado un conjunto de clientes en Cola ECli y un conjunto de operadores libres ↩
    EOper,
se intenta asignar un operador a un cliente.
Si hay operadores libres, el que ha estado libre más tiempo (operAsignado) debe ser
asignado al cliente que ha Colado más tiempo (cliAtendido).
Si no hay operadores libres, se debe dejar el cliente en Cola.
Se deben remover clientes y operadores de los conjutos originales si corresponde.
Si se puede hacer una asignación de un operador a un cliente se devuelve true.
*/

bool asignarUnOperador(Cola & ECli, Cola & EOper, int & cliAtendido, int & ↩
    operAsignado){
  bool asignado = false;
  if (!esVaciaCola(EOper) && !esVaciaCola(ECli)){ //puedo asignar operador
    operAsignado = frente(EOper);
    desencolar(EOper);
    cliAtendido = frente(ECli);
    desencolar(ECli);
    asignado = true;
  }
  return asignado;
}
```