Examen de Programación 2

16 de febrero de 2018

Problema 1 (25 puntos)

(a) Defina una estructura de tipo **ABB** que represente el tipo de datos árboles binarios de búsqueda de enteros en memoria dinámica.

(b) Implemente una función **bool elimHoja(ABB & a, int e)** que dado un árbol binario de búsqueda **a** de tipo **ABB** y dado un entero **e**, elimina a **e** del árbol **a** si es el dato de un nodo hoja; retorna *true* si fue posible eliminar **e** de **a** y *false* en caso contrario (si **e** no está en **a** o no es una hoja de **a**). No pueden definirse operaciones auxiliares.

(c) Indique el orden de tiempo de ejecución en el peor caso y en el caso promedio de la operación definida en la parte b). Justifique muy brevemente.

Solución:

(a)
```
struct nodo
{
    int dato;
    struct nodo *izq, *der;
};

typedef nodo* ABB;
```

(b)
```
bool elimHoja(ABB &a, int e){
    if(a == NULL) return false;

    if(e == a->dato){
        if (a->izq == NULL && a->der == NULL){
          delete a;
          a = NULL;
          return true;
        } else {
          return false;
        }
    }

    if(e > a->dato){
        return elimHoja(a->der, e);
    }else{
        return elimHoja(a->izq, e);
    }
};
```

(c) El orden de tiempo de ejecución en el peor caso es $O(n)$, siendo *n* la cantidad de nodos en el árbol. El peor caso ocurre cuando se quiere eliminar una hoja de un ABB cuya estructura se asemeja a la de una lista encadenada, esto es, cuando todos los nodos se ubican siempre a la derecha (o a la izquierda) de su nodo padre (este caso se presenta cuando se inserta una lista de valores en el árbol en orden creciente o decreciente). En este caso, para eliminar la única hoja presente en el árbol se deben recorrer todos los nodos del árbol.

El orden de tiempo de ejecución en el caso promedio es $O(\log n)$, ya que es el orden que tiene la altura de un árbol binario de búsqueda en promedio.

Problema 2 (20 puntos)

Considere un árbol general de enteros representado mediante un árbol binario de enteros con la semántica: puntero al primer hijo (pH), puntero al siguiente hermano (sH).

```
typedef struct nodoAG* AG;
struct nodoAG{
  int dato;
  AG pH, sH;
}
```

Implemente una función **bool amplitudAcotada(AG a, unsigned int k)** que retorne *true* si y solo si todo nodo del árbol general que es representado por **a** tiene a lo sumo **k** hijos. Si el árbol **a** es vacío, la función debe retornar *true*.

Solución:

```
/* Creamos una función auxiliar que dado un nodo del árbol general
   devuelve su cantidad de hijos.
   precondición: a != NULL */
unsigned int cantidad_hijos(AG a) {
  unsigned int res = 0;
  AG cursor = a->pH;
  while (cursor != NULL) {
    res++;
    cursor = cursor->sH;
  }
  return res;
}

bool amplitudAcotada(AG a, unsigned int k){
    if (a == NULL) {
      return true;
    } else {
      return cantidad_hijos(a) <= k && amplitudAcotada(a->pH, k) ↩
          && amplitudAcotada(a->sH, k);
    }
}
```

Problema 3 (55 puntos)

(a) Especificar, con pre y postcondiciones, un TAD **Lineal**, de elementos de tipo *int*, no acotado, adecuado para modelar un sistema con operaciones que permitan:

1. Crear la estructura vacía,
2. Agregar un elemento a la estructura,
3. Eliminar y retornar el primer elemento ingresado (si la estructura es no vacía),
4. Eliminar y retornar el último elemento ingresado (si la estructura es no vacía),
5. Retornar la cantidad de elementos que tiene la estructura,
6. Retornar una copia del TAD sin compartir memoria.
7. Liberar la memoria utilizada por el TAD.

(b) Implementar el TAD **Lineal**, sin usar TADs auxiliares, de tal manera que las operaciones 1–5 sean O(1) peor caso.

(c) Defina una función **capicua** que, dada una estructura lineal *e* de tipo **Lineal**, retorne *true* si y solo si los elementos que se obtienen de *e* siguiendo una política FIFO coinciden con los que se obtienen siguiendo una política LIFO; esto es, si *e* es capicúa. La función no debe modificar *e* ni acceder a la representación elegida para el TAD.

(d) Indique el orden de tiempo de ejecución para el peor caso de la función **capicua**. Justifique brevemente.

Solución:

```
Parte (a)

/******** Constructoras *********/
/*
Post: Devuelve una estructura lineal vacía
*/
Lineal crear_lineal ()

/*
Pre:  l es una estructura lineal
Post: Lineal tiene el elemento e incluido
*/
void agregar_elem (Lineal & l, int e)


/******** Selectoras *********/
/*
Pre:  l tiene al menos un elemento
Post: Elimina el primer elemento ingresado y lo retorna
*/
int obtener_primero (Lineal & l)

/*
Pre:  l tiene al menos un elemento
Post: Elimina el último elemento ingresado y lo retorna
*/
int obtener_ultimo (Lineal & l)

/******** Operaciones adicionales *********/
/*
Pre:  l es una estructura lineal
```

```
Post: Retorna la cantidad de elementos que tiene la estructura
*/
int cant_elem (Lineal l)

/*
Pre:  l es una estructura lineal
Post: Retorna una copia del TAD sin compartir memoria
*/
Lineal copiar_lineal (Lineal l)

/******** Destructoras *********/
/*
Pre:  l es una estructura lineal
Post: Libera la memoria utilizada por la estructura
*/
void destruir_lineal (Lineal & l)

PARTE (b)

/*
La implementación se realiza utilizando una celda como cabezal, con un puntero al inicio de la estructura y otro al final. Esto es para poder tener acceso en O(1) al primero y al último elemento que fueron insertados. Además el cabezal cuenta con la cantidad de elementos que forman parte de la estructura, también para tener O(1) en la función cant_elem. La estructura donde se almacenan los datos es una lista de enteros doblemente enlazada.
*/

struct node
{
    int    elem;
    node * sig;
    node * ant;
};

/* Notar que el puntero primero (primero en llegar) refiere al elemento más antiguo en la lista, mientras que el último (último en llegar) refiere al más nuevo.*/
struct cabezal
{
    node * primero;
    node * ultimo;
    int    cant_elem;
};


typedef cabezal * Lineal;

Lineal crear_lineal ()
{
   Lineal l     = new lineal;
   l->primero   = NULL;
   l->ultimo    = NULL;
```

```
    l->cant_elem = 0;
    return l;
}

void agregar_elem (Lineal & l, int elem)
{
     node * n = new node;
     n->elem  = elem;
     if (l->cant_elem == 0)
     {
          n->sig = NULL;
          n->ant = NULL;
          l->primero = n;
          l->ultimo  = n;

     }
     else
     {
          n->sig = l->ultimo;
          n->ant = NULL;
          l->ultimo->ant = n;
          l->ultimo      = n;
     }
     l->cant_elem++;
}

int obtener_primero (Lineal & l)
{
     node * aux = l->primero;
     if (l->cant_elem == 1)
     {
          l->primero = NULL;
          l->ultimo  = NULL;
     }
     else
     {
          l->primero      = aux->ant;
          l->primero->sig = NULL;
     }
     l->cant_elem--;
     int elem = aux->elem;
     delete aux;
     return elem;
}


int obtener_ultimo (Lineal & l)
{
     node * aux = l->ultimo;
     if (l->cant_elem == 1)
     {
          l->primero = NULL;
          l->ultimo  = NULL;
     }
     else
```

```
    {
        l->ultimo       = aux->sig;
        l->ultimo->ant  = NULL;
    }
    l->cant_elem--;
    int elem = aux->elem;
    delete aux;
    return elem;
}

int cant_elem (Lineal l)
{
    return l->cant_elem;
}

Lineal copiar_lineal (Lineal l)
{
    Lineal copia = crear_lineal ();
    node * index = l->primero;
    /*Se recorre l con un alias (index) y se van agregando los ↩
       elementos en la copia (copia)*/
    while (index != NULL)
    {
        agregar_elem (copia, index->elem);
        index           = index->ant;
    }
    return copia;
}

void destruir_lineal (Lineal & l)
{
    while (l->primero != NULL)
    {
        node * aux = l->primero;
        l->primero = l->primero->ant;
        delete aux;
    }
    delete l;
}


Parte (c)

bool capicua (Lineal e)
{
    Lineal copia = copiar_lineal (e);
    bool capicua = true;
    /*Recorro la estructura hasta que el inicio y el fin sean ↩
       diferentes o llegue al medio*/
    while (capicua && cant_elem (copia) > 1)
    {
       if (obtener_primero (copia) != obtener_ultimo (copia)) ↩
          capicua = false;
    }
    /*Libero la memoria de la copia*/
```

```
    destruir_lineal (copia);
    return capicua;
}

Parte (d)

La función capicúa está compuesta por el llamado a la función ↩
   copiar_lineal, cuyo orden es n dado que se recorren todos los ↩
   elementos de la estructura. Por otro lado lado tenemos el while↩
   , que se ejecutará en el peor caso n/2 veces, esto es cuando la↩
    cantidad de elementos sea un número par y además cumplan la ↩
   propiedad se ser capicúa.Finalmente tenemos el llamado a la ↩
   función destruir_lineal, también de orden n. Luego, el orden de↩
    la función es O(n + n/2 + n) = O(n).
```