Primer Parcial de Programación 2

5 de mayo de 2017

Problema 1 (15 puntos)

Considere la siguiente definición de nodos de listas dinámicas de enteros:

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

(a) Defina una función iterativa (no recursiva) **mayor** que dada una lista de enteros `L` no vacía y sin elementos repetidos retorne un puntero al nodo de `L` que contiene al mayor elemento:

```
Lista mayor (Lista L);
```

(b) Defina un procedimiento iterativo **ordenar** que, usando la función `mayor`, permita ordenar una lista de enteros `L`, sin elementos repetidos, de mayor a menor sin modificar los punteros de los nodos de `L` (sólo modificando eventualmente los valores enteros en el campo `dato` de los nodos). Si `L` es vacía, el procedimiento no tendrá efecto.

```
void ordenar (Lista & L);
```

(c) Calcule el orden de tiempo de ejecución en peor caso de las operaciones `mayor` y `ordenar`.

Solución:

(a)
```
Lista mayor (Lista L)
{
  // Puntero al nodo con el mayor elemento.
  Lista pmayor = L;
  // Por letra la lista L no es vacía.
  Lista iterador = L->sig;
  while (iterador != NULL)
  {
    if (iterador->dato > pmayor->dato)
      pmayor = iterador;
      iterador = iterador->sig;
  }
  return pmayor;
}
```

(b)
```
void ordenar (Lista &L)
{
  Lista iterador = L;
  while (iterador != NULL)
  {
    // Busco el mayor del resto de la lista
    Lista pos_mayor = mayor (iterador);

    //Hago el swap del mayor con el elemento actual (↩
       iterador)
    int aux = iterador->dato;
    iterador->dato = pos_mayor->dato;
    pos_mayor->dato = aux;
```

```
        //Avanzo el iterador
        iterador = iterador->sig;
      }
    }
```

(c) El tiempo de ejecución en el peor caso para la función `mayor` lo podemos escribir como (ignorando las constantes):

$$T(n) = \sum_{i=1}^{n} c,$$

donde c es una constante que corresponde a las instrucciones dentro del `while` y n es la cantidad de elementos de la lista. Entonces, obtenemos que $T(n) = n \cdot c$. Por lo tanto el orden de ejecución es $O(n)$.

En la función `ordenar` podemos ver que dentro del `while` lo que se hace es llamar a la función `mayor` cada vez con una lista de tamaño menor. Entonces podemos escribir el tiempo de ejecución de la función `ordenar` como:

$$T(n) = \sum_{i=1}^{n} T'(i)$$

donde T' es el tiempo de ejecución de la función `mayor`. Desarrollando obtenemos:

$$\begin{aligned} T(n) &= \sum_{i=1}^{n} c \cdot i \\ &= c\left(\frac{n(n+1)}{2}\right) \\ &= \frac{c}{2}n^2 + \frac{c}{2}n. \end{aligned}$$

Entonces el orden de ejecución de la función `ordenar` es $O(n^2)$.

Problema 2 (13 puntos)

Considere la siguiente declaración del tipo de los nodos de un Árbol Binario de Búsqueda (`ABB`) de enteros:

```
struct nodoABB {
  int info;
  nodoABB *izq;
  nodoABB *der;
};
typedef nodoABB* ABB;
```

(a) Defina una función **aplanar** que dado un `ABB A` retorne una lista ordenada de menor a mayor que contenga a todos los elementos de `A`. Si el árbol es vacío, la función debe retornar la lista vacía. Use la definición de listas de enteros dada en el Problema 1.

```
Lista aplanar(ABB A);
```

No use operaciones auxiliares propias en la implementación de `aplanar`, excepto el siguiente procedimiento `concatenar`, que puede asumir implementado:

```
void concatenar(Lista & L1, Lista L2);
```

`concatenar(L1, L2)` agrega los elementos de `L2` al final de `L1`, en el mismo orden (`L1 = L1++L2`).

(b) ¿Cuál es el orden de tiempo de ejecución en el peor caso de la función `aplanar`, asumiendo que `concatenar(L1, L2)` tiene O(`n`) en el peor caso, siendo `n` la suma de los largos de `L1` y `L2`? Justifique muy brevemente.

Solución:

(a) Hay dos casos: el árbol es vacío o no. En caso de que el árbol sea vacío debemos devolver una lista vacía, en caso contrario debemos devolver una lista que contenga tenga los elementos del subárbol izquierdo aplanado, el elemento de la raíz y los elementos del subárbol derecho aplanado.

```
Lista aplanar (ABB A)
{
  Lista aplanadoI = NULL; // lista resultado
  if (A != NULL)
  {
    // subarbol izq aplanado
    aplanadoI = aplanar(A->izq);
    // subarbol der aplanado
    Lista aplanadoD = aplanar(A->der);
    // nodo para elemento raiz
    Lista medio = new nodoLista;
    medio->dato = A->info;
    // nodo con elemento raiz se coloca al principio
    // de la lista con subarbol der aplanado
    medio->sig = aplanadoD;
    concatenar(aplanadoI, medio);
  }
  return aplanadoI;
}
```

(b) **¿Cuál es el orden de tiempo de ejecución en el peor caso de la función aplanar? Justifique muy brevemente.**

La función `aplanar` consiste en una recorrida en postorden del árbol A que es parámetro de entrada. Por lo tanto el tiempo de ejecución en el peor caso de `aplanar` se puede calcular como la suma de las acciones en cada nodo.

En la recorrida, para cada nodo del árbol A se hace una cantidad constante de operaciones (creación de un nuevo nodo, asignación de valores, punteros, etc.) y una invocación al procedimiento `concatenar` que sabemos que opera sobre listas que en todos los casos tienen una suma de elementos menor a n, donde n es el numéro total de nodos del árbol.

Dado que el el tiempo de ejecución en el peor caso de `concatenar` es O(`m`) en el peor caso, siendo `m` la suma de los largos de las listas que se concatenan, se puede deducir que `concatenar` en todos los nodos del árbol tiene orden de ejecución en el peor caso menor que n siendo `n` el total de nodos de A (porque se concatenan listas cuyo largo total siempre es menor o igual a n):

$$T(n) \leq \sum_{i=1}^{n} (c + T_{concatenar}(n))$$
$$T(n) \leq \sum_{i=1}^{n} c' \cdot n$$
$$T(n) \leq c' \cdot n^2.$$

$T(n)$ es $O(n^2)$.

Problema 3 (12 puntos)

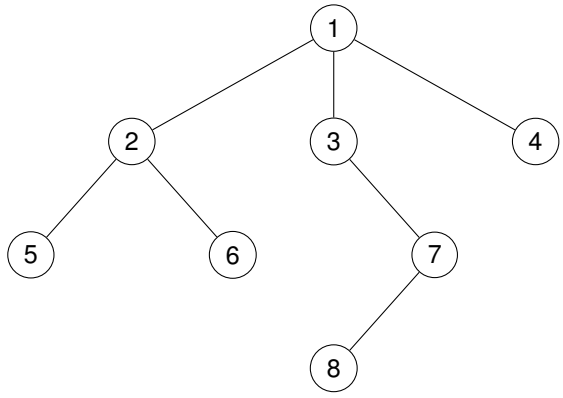
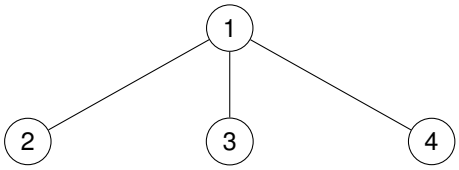
Considere la siguiente definición en C/C++ del tipo ArbGen de árboles generales o finitarios de enteros representados con árboles binarios con la semántica: primer hijo (pH) – siguiente hermano (sH):

```
struct nodoArbGen {
  int dato;
  nodoArbGen * pH;
  nodoArbGen * sH;
};
typedef nodoArbGen* ArbGen;
```

Defina una función recursiva **copiaParcial** en C/C++ que dados un árbol A de tipo ArbGen y un entero positivo k, retorne una copia de A, sin compartir memoria con éste, con todos los nodos que están en un nivel menor o igual a k. Si A es vacío o k es cero, el resultado debe ser el árbol vacío. Tenga en cuenta que en un árbol no vacío la raíz está en el nivel 1 y asuma que A->sH es NULL. No use operaciones auxiliares propias en la implementación de copiaParcial.

```
ArbGen copiaParcial (ArbGen A, unsigned int k);
```

Ejemplo:

Entrada	Resultado
$k = 2$ A 	

Solución:

```
ArbGen copiaParcial (ArbGen a, unsigned int k)
{
  //Controlo que no haya llegado al limite de k y que el arbol↩
      no sea vacio
  if (k > 0 && a != NULL)
  {
    //Copio el nodo actual
    ArbGen copiaNodo = new nodoArbGen;
    copiaNodo->dato = a->dato;
    //Hago una copia parcial de los hijos del nodo actual con ↩
       k-1, dado que bajo al siguiente nivel
    copiaNodo->pH = copiaParcial (a->pH, k-1);
    //Hago una copia parcial de los hermanos del nodo actual ↩
       manteniendo el valor de k dado que no estoy bajando de ↩
       nivel
```

```
        copiaNodo->sH = copiaParcial (a->sH, k);
        return copiaNodo;
      }
      return NULL;
    }
```