Primer Parcial de Programación 2

4 de mayo de 2018

En recuadros con este formato aparecerán aclaraciones que cumplen una función explicativa pero que no eran requeridos como parte de la solución.

Problema 1 (15 puntos)

(a) Complete la siguiente definición general de orden de tiempo de ejecución: "T(n) es O(f(n)) si ..."

(b) Considere la siguiente representación para una lista encadenada de enteros:

```
struct nodo {
  int elem;
  nodo *sig;
};
typedef nodo *lista;
```

Sea `P` un procedimiento que recibe como parámetro una lista de enteros de largo n, definido de la siguiente forma:

```
void P(lista l, unsigned int n) {
  for (int i = n; i > 0; i--) {
    lista itera = l;
    for (int j = 1; j < i; j++)
      itera = itera->sig;
    printf("%d", itera->elem);
  }
}
```

¿Qué imprime el procedimiento `P`, dada un lista de enteros de largo `n`?

(c) Asuma que el tiempo de ejecución en peor caso de `P`, T(n), está determinado por la cantidad de veces que se ejecuta la sentencia `itera=itera->sig`. Calcule T(n) y el orden de T(n), en el peor caso.

(d) ¿Podría implementarse otro procedimiento que imprima lo mismo que `P` pero que tenga un menor orden de tiempo de ejecución en el peor caso? Justifique en caso negativo y en caso afirmativo escriba el procedimiento, indicando el orden (O).

Solución:

(a) $T(n)$ es $O(f(n))$ si existen c y n_0 constantes positivas para los cuales se cumple que $T(n) \leq c * f(n) \forall n > n_0$.

(b) Imprime los elementos de la lista l desde el último al primero.

(c) El tiempo de ejecución en el peor caso para el procedimiento `P` asumiendo que está determinado por la cantidad de veces que se ejecuta la sentencia `itera=itera->sig` y donde n es la cantidad de elementos de la lista, lo podemos escribir como:

$$T(n) = \sum_{i=1}^{n} \sum_{j=1}^{i-1} 1 = \sum_{i=1}^{n} (i-1) = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2}$$

Por lo tanto el orden de ejecución es $O(n^2)$.

(d) Sea `PRec` un procedimiento recursivo que recibe como parámetro una lista de enteros de largo n e imprime los elementos de la lista l desde el último al primero:

```
void PRec(lista l, unsigned int n) { // void PRec(lista l) {
  if (n != 0) {                      //   if (l != NULL) {
    PRec(l->sig, n - 1);             //     PRec(l->sig)
    printf("%d", l->elem);
  }
}
```

Para `PRec` $T(n)$ tiene la siguiente forma:

$$T(0) = 1$$

$$T(n) = 1 + T(n-1)$$

Por lo tanto el orden de ejecución es $O(n)$.

Sea `PIt` un procedimiento iterativo que recibe como parámetro una lista de enteros de largo n e imprime los elementos de la lista l desde el último al primero:

```
void PIt(lista l, unsigned int n) {       // void PIt(lista l) {
  lista itera, listaInversa = NULL;
  for (unsigned int i = 1; i <= n; i++) { //   while (l != NULL) {
    itera = new nodo;
    itera->elem = l->elem;
    l = l->sig;
    itera->sig = listaInversa;
    listaInversa = itera;
  }
  for (unsigned int i = 1; i <= n; i++) { //   while (listaInversa != NULL) {
    printf("%d", listaInversa->elem);
    itera = listaInversa;
    listaInversa = listaInversa->sig;
    delete itera;
  }
}
```

Para `PIt` $T(n)$ es de la siguiente forma: $\sum_{i=1}^{n} 1 + \sum_{i=1}^{n} 1 = n + n = 2n$

Por lo tanto el orden de ejecución es $O(n)$.

En ambas soluciones se mantiene el parámetro n por compatibilidad con la letra y para resolver un problema más general al planteado, el de imprimir los primeros los primeros n elementos en orden inverso. Se ven, comentados, los cambios para la opción sin el parámetro n.

Otra opción válida para la versión iterativa es usar un arreglo en vez de una `lista`. Hay que destacar que en cualquiera de las opciones de la versión iterativa es necesario liberar la memoria usada para la lista invertida.

Problema 2 (13 puntos)

Considere la siguiente definición del tipo `ABB` de árboles binarios de búsqueda de enteros:

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

(a) Implemente un procedimiento iterativo `concat` que, dadas dos listas `l` y `p`, agregue al final de la lista `l` los elementos de `p`, en el mismo orden. La lista `l` resultante comparte memoria con `p`.

```
void concat (lista &l, lista p);
```

(b) Implemente una función recursiva `menores` que dado un árbol binario de búsqueda `t` de tipo `ABB` y dado un entero `k`, retorne una lista (del tipo definido en el Problema 1) que contenga a todos los elementos de `t` menores (estrictos) a `k` y que esté ordenada de mayor a menor. Si no hay elementos en `t` menores a `k` (en particular si `t` es vacío), la lista que se obtiene es vacía. La función `menores` debe evitar recorrer nodos del árbol que no sean estrictamente necesarios. No se permite definir operaciones auxiliares, aunque puede usarse el procedimiento `concat` implementado en la parte anterior.

```
lista menores (ABB t, int k);
```

Solución:

(a) Hay dos casos: la lista l es vacía o no. En caso de que la lista sea vacía, l debe ser p, en caso contrario debemos encontrar el último nodo de l para enlazar la lista p al final de l.

```
void concat(lista &l, lista p) {
  if (p != NULL) {
    if (l == NULL)
      l = p;
    else {
      lista itera = l;
      while (itera->sig != NULL)
        itera = itera->sig;
      itera->sig = p;
    }
  }
}
```

(b) 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 los elementos menores que k del derecho, el elemento de la raíz si es menor que k y los elementos del subárbol izquierdo menores que k .

```
lista menores(ABB t, int k) {
  lista result = NULL; // lista resultado
  if (t != NULL) {
    lista menoresI = menores(t->izq, k);
    if (t->dato < k) {
      lista lRaiz = new nodo;
      lRaiz->elem = t->dato;
      lRaiz->sig = menoresI;
      result = menores(t->der, k);
      concat(result, lRaiz);
    } else
      result = menoresI;
  }
  return result;
}
```

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

En la recorrida, para cada nodo del árbol t 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 t (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 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;
```

(a) Implemente una función recursiva `buscar` que dado un árbol general `t` sin elementos repetidos y dado un entero `x`, retorne un puntero al nodo del árbol que tenga a `x` como dato, si `x` pertenece a `t`, o NULL en caso contrario. No se permite definir operaciones auxiliares.

```
ArbGen buscar(ArbGen t, int x);
```

(b) Implemente, utilizando la función `buscar`, un procedimiento `insertar` que dado un árbol general `t` sin elementos repetidos y dados enteros `v` y `w`, inserte a `v` como primer hijo de `w` en `t` (hijo más a la izquierda), siempre que `w` pertenezca al árbol y `v` no pertenezca al árbol. En caso contrario, la operación no tendrá efecto.

```
void insertar(ArbGen & t, int v, int w);
```

Solución:

(a)
```
ArbGen buscar(ArbGen t, int x) {
    ArbGen res;
    if (t == NULL)
      res = NULL;
    else if (t->dato == x)
      res = t;
```

```
    else {
      res = buscar(t->pH, x);
      if (res == NULL)
        res = buscar(t->sH, x);
    }
    return res;
  }
```

(b)
```
  void insertar(ArbGen &t, int v, int w) {
    ArbGen padre = buscar(t, w);
    if (padre != NULL && buscar(t, v) == NULL) {
      ArbGen hijo = new nodoArbGen;
      hijo->dato = v;
      hijo->pH = NULL;
      hijo->sH = padre->pH;
      padre->pH = hijo;
    }
  }
```