Instituto de Computación. Facultad de Ingeniería. Universidad de la República

Primer Parcial de Programación 2

Abril de 2023

Problema 1 (18 puntos: 6+8+4)

Considere la siguiente definición del tipo ***Lista*** de enteros, doblemente encadenada y con punteros (accesos) al inicio y al final de la lista:

```
struct cabezal{
  nodoLista * inicio;
  nodoLista * final;
}
typedef cabezal * Lista
```

```
struct nodoLista{
  int dato;
  nodoLista * sig;
  nodoLista * ant;
}
```

Tener en cuenta que en una lista vacía existe el *cabezal*, donde los punteros *inicio* y *final* de éste son NULL.

Parte 1) Implemente el procedimiento ***insFinal*** que, dados un entero *x* y una lista *l* de tipo *Lista*, agregue *x* al final de *l* en O(1) peor caso.

```
void insFinal(int x, Lista & l)
```

Parte 2) Implemente una <u>función iterativa</u> ***pivote*** que, dados un entero *x* y una lista *l* de tipo *Lista* que <u>no contiene a *x*</u>, construya y devuelva una nueva lista (que no comparta memoria con *l*) que tenga a *x* y a todos los elementos de *l*, de tal manera que: todos los elementos de *l* menores a *x*, si existen, deben estar antes que *x* (entre el inicio de la lista resultado y *x*, en cualquier orden) y todos los elementos mayores a *x*, si existen, deben estar después de *x* (entre *x* y el final de la lista resultado, en cualquier orden). Para implementar *pivote* <u>asuma implementados</u> correctamente los procedimientos *insFinal* (de la Parte 1) e *insInicio* (que inserta un entero al inicio de una lista de tipo *Lista*), considerando que ambos insertan en O(1) peor caso. La <u>única manera permitida</u> de agregar elementos a una lista de tipo *Lista* es a través de *InsFinal* y *insInicio*. La función *pivote* no debe modificar la lista parámetro *l* ni usar estructuras de datos auxiliares (como arreglos u otro tipo de listas, por ejemplo).

```
Lista pivote(int x, Lista l)
```

<u>Ejemplo</u>: Si *x* = 7 y *l* = [9, 3, 11, 9, 5, 2, 4], la lista resultado de *pivote(x, l)* podría ser: [4, 2, 5, 3, <u>7</u>, 9, 11, 9].

Parte 3) Calcule el orden O de tiempo de ejecución en el peor caso de *pivote*.

¿Coincide con Θ? Justifique muy brevemente.

Problema 2 (19 puntos: 15+4)

Considere la siguiente definición del tipo ***ABB*** de árboles binarios de búsqueda de números reales (sin elementos repetidos):

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

Parte 1) Implemente una <u>operación recursiva</u> ***borrarMayores*** que, dados un número real *x* y un árbol binario de búsqueda *t* de tipo *ABB*, elimine de *t* todos los elementos mayores estrictos que *x* y retorne la cantidad de elementos suprimidos. El árbol resultante debe ser también binario de búsqueda. Si no hay elementos mayores que *x* en *t* (en particular si *t* es el árbol vacío, NULL), *borrarMayores* no tendrá efecto en *t* y el retorno será 0. La operación deberá evitar recorrer nodos innecesarios de *t*. <u>No defina operaciones auxiliares</u> para implementar *borrarMayores*.

```
int borrarMayores(float x, ABB & t)
```

Parte 2) Indique el orden de tiempo de ejecución en el peor caso de *borrarMayores*. Justifique muy brevemente.

Instituto de Computación. Facultad de Ingeniería. Universidad de la República

Primer Parcial de Programación 2

Abril de 2023

Soluciones

Problema 1

Parte 1)

```
// Inserta el elemento x al final de la lista l
void insFinal(int x, Lista & l){
 nodoLista* nuevo = new nodoLista;
 nuevo->dato = x;
 nuevo->sig = NULL;
 if (l->inicio == NULL){
       l->inicio = nuevo;
       nuevo->ant = NULL;
   }
 else{
       nuevo->ant = l->final;
       l->final->sig = nuevo;
 }
 l->final = nuevo;
}
```

Parte 2)

```
/* Devuelve una lista que no comparte memoria con l, tiene a x y a todos los
elementos de l, de forma tal que los menores que x están al comienzo
y los mayor al final. Se asume x no pertenece a l. */

Lista pivote(int x, Lista l){
 Lista lres = new cabezal;
 // Se crea e inicializa una lista nueva que solo tiene a x
 lres->inicio = lres->final = NULL;
 insInicio(x, lres);
 nodoLista* it = l->inicio; // it se usar para recorrer l
 while (it!=NULL){
       if (it->dato < x) insInicio(it->dato, lres);
       else insFinal(it->dato, lres);
       it = it->sig;
 }
 return lres;
}
```

Parte 3)

Primero notamos que tanto *insFinal* como *insInicio* son O(1). Como el resto de las operaciones que se realizan por fuera del *while* son de tiempo constante, podemos decir que el tiempo total de ejecución de las instrucciones fuera del *while* está acotado por una constante $c1$. Por otro lado, como las operaciones del cuerpo del *while* son de tiempo constante, el tiempo de ejecución del cuerpo del while está acotado por una constante $c2$. Por último, como en cada iteración del *while* se avanza el puntero *it,* el cuerpo del *while* se ejecuta n veces, siendo n la cantidad de nodos de la lista. De esta manera el tiempo de ejecución esta acotado superiormente de la siguiente manera, $T(n) < c1 + c2 \,.\, n < (c1 + c2) \,.\, n,$ para todo $n>0$. Aplicando la definición de orden vemos que el tiempo de ejecución de la función es O(n).

Primer Parcial de Programación 2

Abril de 2023

Para que coincida con Θ, la función debe ser también $\Omega(n)$. Como en cualquiera de los casos el cuerpo del while se ejecuta n veces, y cada iteración lleva al menos un mínimo de tiempo $c3$, el tiempo de ejecución en cualquier caso está acotado inferiormente de la siguiente manera $T(n) > n. c3$, en particular en el peor caso. Aplicando la definición de Ω vemos que $T(n)$ es $\Omega(n)$ en el peor caso y por lo tanto también es $\Theta(n)$.

Problema 2

Parte 1)

```
/* Elimina de t los elementos mayores que x y retorna la cantidad de
elementos suprimidos. Se asume que no hay elementos repetidos. */

int borrarMayores(float x, ABB & t){
 int result = 0;
 if (t != NULL){
       if (t->dato > x){
           result = 1 + borrarMayores(x, t->izq) + borrarMayores(x, t->der);
           ABB aBorrar = t;
           t = t->izq;
           delete aBorrar;
       }
       else  result = borrarMayores(x, t->der);
 }
 return result;
}
```

Parte 2)

Se visita cada nodo del árbol una vez realizando instrucciones de O(1). En el peor caso se visitan todos los nodos; esto ocurre cuando todos los elementos en *t* son mayores que *x*. Por lo tanto, en el peor caso $T(n)$ es $O(n)$, siendo n la cantidad de nodos de *t*.