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

Examen de Programación 2 – Julio de 2016

Generalidades:

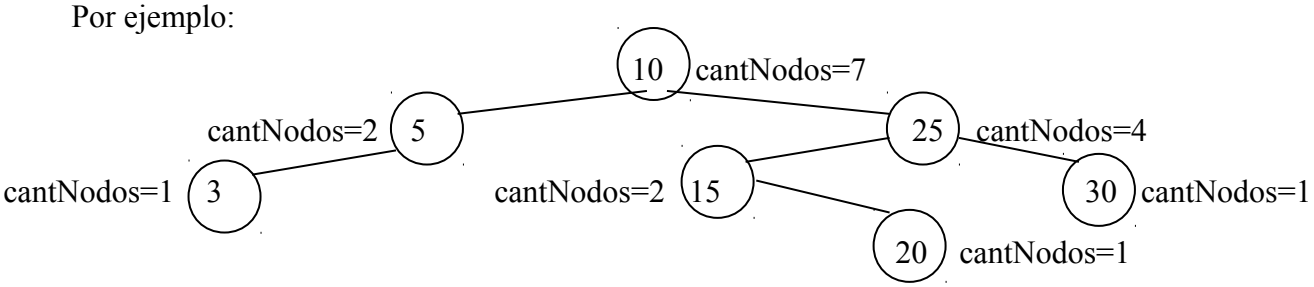
a. La prueba es individual y sin material. b. La duración es 3hs. c. Sólo se contestan dudas acerca de la letra.	d. Escriba las hojas de un sólo lado y con letra clara. e. Numere cada hoja, indicando en la primera el total. f. Coloque su nro. de cédula y nombre en cada hoja.

Problema 1. (40 puntos)

Los árboles binarios de búsquedas (*ABB*) balanceados podrían definirse teniendo en cuenta la cantidad de nodos en vez de la altura de los subárboles. Diremos que un *ABB* está balanceado si para cada nodo se cumple que la diferencia de la cantidad de nodos de sus subárboles (izquierdo y derecho) difiere en a lo sumo en *K* unidades, siendo *K* una constante conocida (que puede referirse desde el código).

Considere nodos de árboles binarios de búsqueda de enteros (de tipo puntero a *nodoABB*) con los siguientes campos: *dato* de tipo entero; *cantNodos* de tipo entero (no negativo); *izq* y *der* de tipo puntero a *nodoABB*. Cada nodo en el campo *cantNodos* almacena la cantidad de nodos del árbol cuya raíz es dicho nodo. Recordar que la cantidad de nodos de un árbol vacío es 0, por definición.

Por ejemplo:



Se pide:

a) Defina los tipos *nodoABB* y *ABB* (árbol binario de búsqueda de enteros como un puntero a *nodoABB)*.

b) Implemente una función recursiva *Insertar* que, dado *A* de tipo *ABB* y un entero *x*, inserte a *x* en *A* y retorne true (verdadero) si, y sólo si, luego de la inserción cada nodo cumple la condición de balanceo previamente definida.

Si *x* ya estaba en *A*, la función no debería modificar el árbol y el resultado sería true. *Insertar* debe dejar el campo *cantNodos* de cada nodo consistente con su definición, asumiendo que en el árbol parámetro cada nodo almacena la cantidad de nodos del árbol cuya raíz es dicho nodo y además, que cada nodo (del árbol parámetro) cumple la condición de balanceo referida.

Cabe aclarar que no se pide balancear el árbol, sino simplemente insertar según el criterio *ABB*, actualizar los campos *cantNodos*, que correspondan, y retornar un booleano que indique si luego de la inserción se cumple la condición de balanceo definida, asumiendo que originalmente ésta se verificaba en cada nodo del árbol parámetro.

La función *Insertar* debe tener tiempo de ejecución promedio logarítmico.

Se puede utilizar la operación *abs* que devuelve el valor absoluto de una expresión numérica.

c) Implemente una función recursiva *estaBalancedo* que, dado *A* de tipo *ABB*, retorne true (verdadero) si, y sólo si, todos los nodos de *A* cumplen la condición de balanceo previamente definida. Si *A* es el árbol vacío, la función debe retornar true.

La función debe tener *O(n)* de tiempo de ejecución en el peor caso, siendo *n* la cantidad de nodos del árbol.

Solución del Problema 1

a)
```
struct nodoABB {
        int dato, cantNodos;
        node *izq , *der;
};
typedef nodoABB *ABB;
```

b)
```
bool insertar (ABB &A, int x){
        bool balanceado = true;
        if (A == NULL) {
                A = new(nodoABB);
                A->dato = x;
                A->cantNodos = 1;
                A->izq = NULL;
                A->der = NULL;
        }
        else if (A->dato > x)
                if (A->der == NULL) {
                        balanceado = insertar(A->izq,x) && A->izq->cantNodos<=K;
                        A->cantNodos++;
                } else {
                        balanceado = insertar(A->izq,x) &&
                                     abs(A->izq->cantNodos – A->der->cantNodos)<=K;
                        A->cantNodos = A->izq->cantNodos +  A->der->cantNodos + 1;
                }

        else if (A->dato < x)
                if (A->izq == NULL) {
                        balanceado = insertar(A->der,x) && A->der->cantNodos<=K;
                        A->cantNodos++;
                } else {
                        balanceado = insertar(A->der,x) &&
                                     abs(A->izq->cantNodos – A->der->cantNodos)<=K;
                        A->cantNodos = A->izq->cantNodos +  A->der->cantNodos + 1;
                }
        return balanceado;
}
```

c)
```
bool estaBalanceado(ABB A) {
        bool eb = true;

        if (A != NULL) {
                int cantIzq = cantDer = 0;
                if (A->izq != NULL)
                        cantIzq =  A->izq->cantNodos;
                if (A->der != NULL)
                        cantDer =  A->der->cantNodos;

                eb = estaBalanceado(A->izq) &&
                     estaBalanceado(A->der) &&
                     abs(cantIzq – cantDer) <= K;
        }
        return eb;
};
```

Problema 2 (60 puntos)

Considere un sistema de administración de impresión se tienen documentos, de tipo *string*, con prioridades en el rango [1:*K*], siendo *K* una constante conocida (que puede referirse desde el código).

Cuando llega un documento a imprimirse, se coloca al final de la secuencia de documentos que esperan ser impresos (en particular, la secuencia puede estar vacía).

Al momento de imprimir se selecciona el documento de mayor prioridad existente, se devuelve y se elimina de la cola. Si hay más de un documento de la misma prioridad se debe elegir al que que llegó primero, o sea al más antiguo de los elementos de la misma prioridad.

Dadas dos prioridades *i* y *j* en el rango [1:*K*], diremos que *i* es una prioridad mayor a *j* si $i > j$.

Se pide:

a) Especifique un TAD *ColaImpresora* de elementos de tipo *string* y con prioridades en el rango [1:*K*] que permita resolver adecuadamente el sistema de administración de impresión previamente descrito. Considere operaciones constructoras, selectoras/destructoras y predicados. Incorpore además una función *clon*, que dada una *ColaImpresora cp* retorne una copia idéntica de *cp* que no comparta memoria con ésta.
Asuma que el *string* es un tipo primitivo, para el cual pueden usarse los operadores habituales de comparación y asignación.

b) Proponga una implementación del TAD *ColaImpresora* en la cual las operaciones de inserción, recuperación y eliminación de elementos sean O(1) peor caso. Desarrolle la representación del TAD e impleméntelo completamente, excepto la función *clon*. Explique gráficamente la representación usada antes de implementar las operaciones.

c) Dadas *cp1* y *cp2* de tipo *ColaImpresora*, diremos que *cp1* y *cp2* son indistinguibles (cp1 ≈ cp2) si la secuencia de documentos (strings) que se obtiene al procesar, según la política definida, todos los elementos (*strings*) de *cp1* coincide con la secuencia para *cp2*. Defina (utilizando únicamente las operaciones del TAD *ColaImpresora* y sin acceder a la representación del TAD) una función *indistinguibles* que dadas *cp1* y *cp2* de tipo *ColaImpresora,* retorne true (verdadero) si y solo si $cp1 \approx cp2$. La función no debe modificar *cp1* ni *cp2*.

   Indique el orden de tiempo de ejecución en el peor caso de la función *indistinguibles*. Justifique brevemente.

Solución del Problema 2

a)
```
/* CONSTRUCTORAS */
/* Crea una cola de impresión vacía */
ColaImpresora crearColaImpresora();

/* Inserta el elemento doc con prioridad prio en la cola ci */
void insertarImpresion(ColaImpresora &ci, string doc, int prio);

/* Retorna una copia limpia de la cola ci */
ColaImpresora clon(ColaImpresora ci);

/* SELECTORAS */
/* Obtiene el elemento prioritario y lo elimina de la cola de impresión ci */
/* PRE: !esVaciaColaImpresion(ci) */
string obtenerImpresion(ColaImpresora &ci);

/* PREDICADOS */
/* Verifica si la cola de impresión ci es vacía */
bool esVaciaColaImpresora(ColaImpresora ci);

/* DESTRUCTORAS */
/* Libera la memoria asociada a la cola de impresión ci */
void destruirColaImpresora(ColaImpresora &ci);
```

b)
```
struct nodoCola {
 string doc;
 nodoCola *sig;
};

struct cabezalCola {
 nodoCola *ultimo, *primero;
};

typedef cabezalCola *Cola;

typedef Cola *ColaImpresora;
```

La representación de la ColaImpresora se basa en un arreglo de tamaño K+1 donde cada celda contiene una cola a excepción de la celda 0 que permanecerá inutilizada. Los trabajos de impresión con prioridad *i* se almacenan en la cola de la celda *i*, respetando el orden de llegada.

* Inserción
La búsqueda de la cola con mayor prioridad, en el peor caso, es de O(K). Luego la inserción en la cola se hace al final de la misma en O(1). Como K no varía con la cantidad de elementos insertados, el O(t(insertarImpresion)) = O(1).

* Recuperación
La búsqueda de la cola con mayor prioridad, en el peor caso, es de O(K). Este caso se verifica por ejemplo cuando todos los elementos de la cola son de máxima prioridad. Como K no varía con la cantidad de elementos insertados, el O(t(obtenerImpresion)) = O(1). Al tener el puntero al primer elemento de la cola, obtener el mismo se hace en O(1) y eliminarlo de la cola también se hace en O(1) porque es el primero.

```
/* Crea una cola de impresión vacía */
ColaImpresora crearColaImpresora(){
	ColaImpresora ci = new Cola[K+1];
	for (int i = 1; i <=K; i++){
		ci[i] = new cabezalCola;
		ci[i]->primero = ci[i]->ultimo = NULL;
	};
	return ci;
};

/* Inserta el elemento doc con prioridad prio en la cola ci */
void insertarImpresion(ColaImpresora &ci, string doc, int prio){
	nodoCola *nuevoT = new  nodoCola;
	nuevoT->doc = doc;
	nuevoT->sig = NULL;

	if (ci[prio]->primero == NULL)
		ci[prio]->primero = nuevoT;
	else
		ci[prio]->ultimo->sig = nuevoT;
	ci[prio]->ultimo = nuevoT;
};


/* SELECTORAS */
/* Obtiene el elemento prioritario y lo elimina de la cola de impresión ci */
/* PRE: !esVaciaColaImpresion(ci) */
string obtenerImpresion(ColaImpresora &ci){
	nodoCola  *aBorrar;
	string trabajo;
	int i;
	for (i = K; (i >=1) && (ci[i]->primero != NULL); i--);

	ABorrar = ci[i]->primero; // se que existe al menos uno por la precondición.
	trabajo = aBorrar->doc;
	ci[i]->primero = ci[i]->primero->sig;
	if (ci[i]->primero == NULL)
		ci[i]->ultimo = NULL;
	delete aBorrar;
	return trabajo;
};

/* Verifica si la cola de impresión ci es vacía */
bool esVaciaColaImpresora(ColaImpresora ci){
	int i;
	for (i = 1; i <= K && ci[i]->primero==NULL; i++);
	return i>K;
};


/* DESTRUCTORAS */
/* Libera la memoria asociada a la cola de impresión ci */
void destruirColaImpresora(ColaImpresora &ci){
	int i;
	for (i = 1; i <= K; i++){
		nodoCola *nc = ci[i]->primero;
		while (nc != NULL){
			nodoCola *aBorrar = nc;
			nc = nc->sig;
			delete aBorrar;
		}
		delete ci[i];
	}
	delete [] ci;
}
```

```
c) bool indistinguibles (ColaImpresora cp1, ColaImpresora cp2) {
          bool indistinguibles;
          if esVaciaColaImpresora(cp1) && esVaciaColaImpresora(cp2)
                indistinguibles = true;
          else if esVaciaColaImpresora(cp1) || esVaciaColaImpresora(cp2)
                indistinguibles = false;
          else {
                ColaImpresora copia1, copia2;
                copia1 = clon(cp1);
                copia2 = clon(cp2);
                indistinguibles = true;
                while (indistinguibles &&
                      !esVaciaColaImpresora(copia1) &&
                      !esVaciaColaImpresora(copia2))
                      indistinguibles = (obtenerImpresion(copia1) == obtenerImpresion(copia2));

                indistinguibles = esVaciaColaImpresora(copia1) &&
                                  esVaciaColaImpresora(copia2) &&
                                  indistinguibles;
                destruirColaImpresion(copia1);
                destruirColaImpresion(copia2);
          }
          return indistinguibles;
     };
```

En el peor caso, la función debe recorrer cada una de las copias de las colas en su totalidad, por lo que es de O(n).