Examen de Programación 2

21 de febrero de 2020

Problema 1 (35 puntos)

Considere la siguiente declaración, en C++, del tipo $Lista$, que representa listas dinámicas de enteros:

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

Dadas dos listas $l1$ y $l2$ de números enteros de igual largo ordenadas estrictamente, diremos que $l1$ es parecida a $l2$ si las listas tienen al menos la mitad de sus elementos en común (iguales). Definir una función iterativa ***parecidas*** que determine si $l1$ es parecida a $l2$. Asumimos como precondición que ambas listas tienen igual largo, están ordenadas de menor a mayor y no poseen elementos repetidos. La función ***parecidas*** debe tener $O(n)$ de tiempo de ejecución en el peor caso (siendo n largo de las listas). Cada lista debe recorrerse a lo sumo una vez.

```
bool parecidas (Lista l1, Lista l2);
```

Ejemplos:

Entradas	Resultado de parecidas
$l1 = [2,5,8,9], l2 = [1,2,4,5]$	$true$
$l1 = [2,5,8], l2 = [1,2,4]$	$false$
$l1 = [2,5,8], l2 = [2,5,8]$	$true$
$l1 = [], l2 = []$	$true$

Solución:

```
bool parecidas(Lista l1, Lista l2) {
  //El largo lo vamos a calcular a partir de l1
  unsigned int largo = 0;
  unsigned int iguales = 0;
  while (l1 != NULL && l2 != NULL) {
    if (l1->info == l2->info) {
      l1 = l1->sig;
      largo++;
      l2 = l2->sig;
      iguales++;
    } else if (l1->info > l2->info) {
      l2 = l2->sig;
    } else {
      l1 = l1->sig;
      largo++;
    }
  }

  //Terminamos de calcular el largo de l1
  while (l1 != NULL) {
    l1 = l1->sig;
    largo++;
  }
  return (2*iguales >= largo);
}
```

Problema 2 (25 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).

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

Implemente un procedimiento recursivo *impDesdeNivel(AG a, int k)* que imprima los elementos del árbol ***a*** que se encuentran en un nivel mayor o igual a k, en cualquier orden. Considere que la raíz del árbol se encuentra en el nivel 1. No se pueden definir operaciones auxiliares para implementar *impDesdeNivel*.

Solución:

```
void impDesdeNivel (AG a, int k) {
    if (a != NULL) {
        if (k <= 1)
            printf("%d \n", a->dato);

        impDesdeNivel(a->sH, k);
        impDesdeNivel(a->pH, k-1);
    }
}
```

Problema 3 (40 puntos)

(a) Especifique, con pre y postcondiciones, del TAD Tabla (funciones parciales, *mappings*) no acotada, de elementos Dominio y Rango de tipos genéricos, con las operaciones básicas: *creoTabla*, *insertar* (que actualiza también), *esTablaVacia*, *estaDefinida*, *recuperar* y *borrar*. Además, extienda el TAD con el predicado *esInyectiva* que retorna true si y sólo si, la tabla es inyectiva: no existen dos elementos distintos en el dominio de la tabla con igual imagen (rango).

(b) Implemente el TAD Tabla anterior usando árboles binarios de búsqueda como estructura de datos; asuma que existe un orden total entre los elementos del dominio. Asuma que los operadores $=$, $<$ y $>$, están definidos para los elementos del tipo Dominio y que el operador $=$ está definido sobre los elementos del tipo Rango. Desarrolle el código de *insertar* y *esInyectiva*. Omita el código del resto de las operaciones del TAD, que se asumen implementadas.

(c) Indique para ambas operaciones implementadas en la parte anterior, cuál es el orden del tiempo de ejecución en el peor caso. Además, para la función *insertar* indique cuál es el orden del tiempo de ejecución en el caso promedio. Justifique brevemente.

Solución:

(a)
```
/* Devuelve una tabla no acotada sin asociaciones. */
Tabla creoTabla();

/* Asocia 'clave' con 'valor' en 't'.
   Si 'clave' ya tiene un elemento asociado, entonces el elemento se actualiza por '↩
       valor'. */
void insertar(Dominio clave, Rango valor, Tabla &t);

/* Elimina de 't' la asociación de 'clave'.
   Precondición: estaDefinida(clave, t) */
void borrar(Dominio clave, Tabla &t);

/* Retorna true si no existen asociaciones en la tabla 't'.*/
bool esTablaVacia(Tabla t);

/* Devuelve true si y solo si 'clave' tiene un entero asociado en 't'. */
bool estaDefinida(Dominio clave, Tabla t);

/* Devuelve el valor de Rango que tiene asociado 'clave'.
   Precondición: estaDefinida(clave, t) */
int recuperar(Dominio clave, Tabla t);


/*******************************/
/**** Operaciones adicionales ****/
/*******************************/

/* Devuelve true si y solo si la tabla es inyectiva
(no existen dos elementos distintos en el dominio de la tabla con igual imagen).*/
bool esInyectiva(Tabla t);
```

(b) En esta parte existen distintas soluciones correctas. En este documento presentamos una alternativa que utiliza funciones auxiliares, pero no requiere de estructuras auxiliares. Otra opción podría ser pasar todos los elementos del Rango a una colección (por ejemplo una lista) y verificar que no existan dos repetidos.

```
struct nodoTabla {
  Dominio clave;
  Rango valor;
  nodoTabla *izq, *der;
};

typedef nodoTabla* Tabla;
```

```
void insertar (Dominio clave, Rango valor, Tabla &t) {
  if (t == NULL) {
      t = new nodoTabla;
      t->clave = clave;
      t->valor = valor;
      t->izq = NULL;
      t->der = NULL;
  } else if (t->clave == clave)
      t->valor = valor;
  else if (t->clave < clave)
      insertar (clave, valor, t->izq);
  else
      insertar (clave, valor, t->der);

}

//Funcion auxiliar que verifica si no existen duplicados de 'valor' en 't'
bool hayDuplicado ( Tabla t, Dominio clave, Rango valor ) {
  return  t != NULL && (t->valor == valor && t->clave != clave ||
          hayDuplicado(t->izq, clave, valor) ||
          hayDuplicado(t->der, clave, valor));
}

//Funcion auxiliar
bool esInyectivaAux ( Tabla t, Tabla raiz ) {
  return  t == NULL ||
          !hayDuplicado(raiz, t->clave, t->valor) &&
          esInyectivaAux(t->izq, raiz) &&
          esInyectivaAux(t->der, raiz));
}

bool esInyectiva(Tabla t) {
  return esInyectivaAux(t, t);
}
```

(c) Respecto a la función *insertar*, en el peor caso el árbol que representa la tabla queda con forma de lista, y se deben recorrer todos los nodos de la estructura para poder realizar una inserción, o actualización. En el caso promedio el árbol se encuentra balanceado y hay que llegar a una hoja para relizar la inserción. Un árbol balanceado tiene en promedio altura $log(n)$ siendo n la cantidad de nodos, por lo que la inserción sería $O(log(n))$.

Respecto a la función *esInyectiva*, para la implementación presentada, en el peor caso se debe ejecutar la función *hayDuplicado* por cada uno de los nodos. La función *hayDuplicado* recorre todos los nodos del árbol en el peor caso, por lo cual su orden es $O(n)$. De esta manera la función *esInyectiva* es de orden $O(n^2)$ en el peor caso.