Prueba Integradora Programación 2

Julio de 2020

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

Problema 1 (25 puntos)

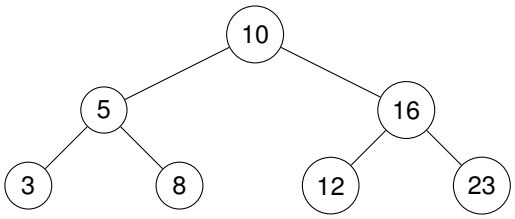
Considere la siguiente declaración del tipo de los nodos de un Árbol Binario de Búsqueda (ABB) de enteros:

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

Defina una función iterativa `menorCercano` que dados un `ABB t` y un entero `x`, retorne el mayor valor perteneciente al árbol que sea menor que `x`; en otras palabras, el valor inferior más cercano a x. Si el árbol es vacío o no existen valores menores que `x` en el árbol, se debe retornar `INT_MIN`. La función debe tener $O(\log(n))$ en el caso promedio. No se permite el uso de operaciones auxiliares para implementar `menorCercano`.

```
int menorCercano (ABB t, int x)
```

Ejemplo: Dado el siguiente `ABB t`




```
menorCercano(t,6) = 5

menorCercano(t,12) = 10

menorCercano(t,25) = 23

menorCercano(t,3) = INT_MIN
```

Solución:

El problema consiste en una búsqueda en un ABB. La parte básica del algoritmo es recorrer nodos del árbol comparando el elemento buscado con el elemento del nodo haciendo uso de la propiedad de orden de los ABB. La recorrida sigue en el subárbol izquierdo si el elemento buscado es menor y en el subárbol derecho si es mayor,

```
if (x < cursor->info)
  cursor = cursor->izq;
else if (x > cursor->info)
  cursor = cursor->der;
```

y depende del problema qué se se debe hacer cuando los elementos son iguales.

En este problema lo anterior debe adaptarse ya que los que se busca no es el parámetro x sino un valor que todavía no se conoce. El resultado debe ser un entero estrictamente menor a ese parámetro, excepto cuando este es INT_MIN, en cuyo caso no hace falta hacer la búsqueda porque ya se conoce que el resultado es INT_MIN.

En otro caso el resultado es un elemento del árbol, el mayor de todos los que son estrictamente menores a x, a menos que esto no sea posible porque x es menor o igual al mínimo de t. En este último caso el resultado también debe ser INT_MIN. Esto se cumple, en particular, cuando t es vacío. En el caso general, como lo que se busca es un elemento estrictamente menor a x, si el valor del elemento es igual a x la búsqueda debe seguir en el subárbol izquierdo.

Entonces la solución puede consistir en empezar asignando el resultado a INT_MIN y "mejorarlo" cada vez que se encuentre un elemento que es mayor al resultado actual y estrictamente menor a x:

```
if ((x > cursor->info) && (cursor->info > res))
  res = cursor->info;
```

En la anterior condición, ¿es necesario incluir el segundo operando? O sea, ¿es posible que se cumpla el primer operando y no el segundo? Puede ocurrir una vez, al principio, si cursor->info == INT_MIN. En este caso no incluir el segundo operando hace que la asignación no sea necesaria pero no produce un error ya que no modifica nada, porque el valor tanto de res como de cursor->info es INT_MIN. Y esta situación solo se puede dar una vez. Luego siempre se va a cumplir cursor->info > res porque cuando se cumple x > cursor->info el cursor se mueve hacia su hijo derecho que, o bien es NIL y el algoritmo termina, o bien por la propiedad de orden de los ABB, el valor de todos los elementos del subárbol que tienen raíz en cursor es mayor que el que queda asignado a res.

Hay que notar que el resultado, aún cuando no es INT_MIN sino un elemento de t, no siempre es un elemento del subárbol con raíz en cursor, sino que puede ser un ancestro propio de ese nodo. En el ejemplo, cuando x es 12, la búsqueda pasa por los nodos con valor 16 y 12 y el resultado es 10, que corresponde a un ancestro de esos nodos.

Además podemos, teniendo en cuenta que el dominio es discreto, detener la búsqueda si el resultado alcanzó $x - 1$ ya que no existe un valor que sea mayor y que además sea menor a x.

Uniendo lo anterior se llega a la siguiente solución

```
int masCercano(ABB t, int x) {
  int res = INT_MIN;
  if (x != INT_MIN) {
    nodoABB * cursor = t;
    while ((cursor != NULL) && (res < x - 1)) {
      if (x <= cursor->info)
        cursor = cursor->izq;
      else {
        res = cursor->info;
        cursor = cursor->der;
      }
    }
  }
```

```
    return res;
  }
```

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; nodoAG* sH; }
typedef struct nodoAG* AG;
```

Implemente la operación recursiva `int podaHojas(AG & t)` que elimine del árbol `t` de tipo `AG` los nodos hoja y retorne la cantidad de nodos suprimidos. Recordar que en un árbol general un nodo es hoja si no tiene hijos y que en la representación pH-sH la raíz del árbol no tiene hermanos. Si el árbol es vacío, deberá retornarse 0. No se permite el uso de operaciones auxiliares para implementar `podaHojas`.

Solución:

```
int podaHojas(AG &t) {
  int res;
  if (t==NULL)
    res = 0;
  else {
    int cantSig = podaHojas(t->sH);
    if (t->pH != NULL)
      res = podaHojas(t->pH) + cantSig;
    else {
      nodoAG* aborrar = t;
      t = t->sH;
      delete aborrar;
      res = 1 + cantSig;
    }
  }
  return res;
}
```

Otra versión:

```
int podaHojas(AG &t) {
  int res;
  if (t==NULL)
    res = 0;
  else {
    if (t->pH != NULL)
      res = podaHojas(t->pH) + podaHojas(t->sH);
    else {
      nodoAG* aborrar = t;
      t = t->sH;
      delete aborrar;
      res = 1 + podaHojas(t);
    }
  }
  return res;
}
```

Otra versión:

```
int podaHojas(AG &t) {
  int res;
  if (t==NULL)
    res = 0;
  else {
    res = podaHojas(t->pH) + podaHojas(t->sH);
    if (t->pH == NULL) {
      nodoAG* aborrar = t;
      t = t->sH;
      delete aborrar;
      res++;
    }
  }
  return res;
```

```
}
```

Problema 3 (32 puntos)

Considere la siguiente especificación del TAD Tabla no acotada de *int* (dominio) en *float* (codominio):

```
struct RepresentacionTabla;
typedef RepresentacionTabla * Tabla;

/*Devuelve la Tabla vacía no acotada, donde cant es una estimación de la cantidad de ↩
    correspondencias a almacenar.*/
Tabla crearTabla (int cant);

/*  Agrega la correspondencia (d,c) en t, si d no tenía imagen en t. En caso contrario ↩
    actualiza la imagen de d con c. */
void insertarTabla (int d, float c, Tabla & t);

/*  Devuelve true si y sólo si d tiene imagen en t. */
bool estaDefinidaTabla (int d, Tabla t);

/*  Retorna la imagen de d en t. Precondición: estaDefinidaTabla(d,t).*/
float recuperarTabla (int d, Tabla t);

/*  Elimina de t la correspondencia que involucra a d, si d está definida en t. En otro caso ↩
    la operación no tiene efecto.*/
void eliminarTabla (int d, Tabla & t);
```

Se pide:

(a) Implemente el TAD Tabla anterior utilizando dispersión abierta (con listas de nodos encadenados) como estructura de datos. Defina el tipo de dato `RepresentacionTabla` e implemente las operaciones *crearTabla* e *insertarTabla*. Omita el código del resto de las operaciones del TAD, que puede asumir implementadas. Considere como función de hash la que realiza la operación módulo entre el valor del dominio y el tamaño de la tabla.

(b) Indique cuáles son las dos condiciones necesarias para que la operación *insertarTabla* tenga O(1) promedio sobre la representación de la parte a).

Solución:

(a)

```
struct nodoT {int dom; float codom; struct nodoT * sigT;};
struct RepresentacionTabla { nodoT * * TablaEnt; int tam;};

typedef RepresentacionTabla * Tabla;

/*  Devuelve la Tabla vacía no acotada, donde se estiman la cantidad (cant) de ↩
    elementos a almacenar (correspondencias).*/
Tabla crearTabla (int cant){
  Tabla nuevaT = new RepresentacionTabla;
  nuevaT->tam = cant;
  nuevaT->TablaEnt = new (nodoT *)[cant];
  for (int i=0; i<cant; i++)
      nuevaT->TablaEnt[i]= NULL;
  return nuevaT;
};


/*  Agrega la correspondencia (d,c) en t, si d no tenía imagen en t. En caso ↩
    contrario actualiza la imagen de d con c.*/
void insertarTabla (int d, float c, Tabla & t){
   int indiceT = d %t->tam;
   nodoT * lEnt = t->TablaEnt[indiceT];
   while ( lEnt != NULL && lEnt->dom != d)
       lEnt = lEnt->sigT;

   if ( lEnt == NULL ){
        nodoT * nuevo = new nodoT;
        nuevo->dom = d;
        nuevo->codom = c;
```

```
            nuevo->sigT = t->TablaEnt[indiceT];
            t->TablaEnt[indiceT] = nuevo;
        }
       else {
            lEnt->codom=c;
       }
    };
```

Versión alternativa de `insertarTabla`

```
void insertarTabla (int d, float c, Tabla & t){
  eliminarTabla(d, t);
  nodoT * nuevo = new nodoT;
  nuevo->dom = d;
  nuevo->codom = c;
  int indiceT = d %t->tam;
  nuevo->sigT = t->TablaEnt[indiceT];
  t->TablaEnt[indiceT] = nuevo;
}
```

(b)

La función de Hash debe realizar una dispersión uniforme de los elementos y el factor de carga (ratio entre el número de elementos y la capacidad de una tabla de dispersión) debe mantenerse menor o igual a uno.