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

Segundo parcial de Programación 2

Diciembre de 2023

Problema 1 (14 puntos: 7+7)

Parte a)

Enuncie brevemente las dos propiedades que debe cumplir un árbol binario para ser un *Heap*.

Parte b)

Enuncie brevemente las dos propiedades que debe cumplir un árbol binario para ser un *AVL*.

Problema 2 (41 puntos: 15+26)

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

```
struct RepTabla;

typedef RepTabla * Tabla;

typedef unsigned int nat;

// POS: Devuelve la Tabla vacía, sin correspondencias.
Tabla crear();

/* POS: 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 insertar (nat d, float c, Tabla & t);

// POS: Devuelve true si y sólo si d tiene imagen en t.
bool definida (nat d, Tabla t);

// POS: Devuelve la cantidad de correspondencias en t. En particular, 0 si t es la tabla vacía.
int cantidad (Tabla t);

// PRE: definida(d,t). POS: Retorna la imagen de d en t.
float recuperar (nat d, Tabla t);

/* POS: Elimina de t la correspondencia que involucra a d, si d está definida en t. En otro caso la operación no tiene efecto. */
void eliminar (nat d, Tabla & t);

// PRE: cantidad(t)!=0. POS: Retorna el mínimo valor del dominio que tiene imagen en t.
nat minDomino (Tabla t);

// PRE: cantidad(t)!=0. POS: Retorna el máximo valor del dominio que tiene imagen en t.
nat maxDominio (Tabla t);

//POS: Imprime las correspondencias (d,c) de t, ordenadas de mayor a menor por los valores del dominio (d).
void imprimir(Tabla t);
```

Parte a)

Una empresa almacena los precios de sus productos en tablas (de tipo *Tabla*), donde el dominio de tipo *unsigned int* (*nat*) corresponde a los identificadores de los productos (no acotados) y el codominio de tipo *float* corresponde a los precios. La empresa quiere evitar inconsistencias de precios de productos de varias tablas y para esto se propone implementar una función iterativa ***preciosUnicos*** que, dadas dos tablas *t1* y *t2* (de tipo *Tabla*) no vacías genere una nueva tabla (de tipo *Tabla*) que contenga las correspondencias entre productos y precios que no generan conflictos entre *t1* y *t2*. Esto es, una correspondencia (*producto*, *precio*) estará en la tabla resultado si y solo si:

- su *producto* está en una sola tabla (ó en *t1* ó en *t2*), ó
- si el *producto* está en ambas tablas (*t1* y *t2*), el *precio* tiene que ser el mismo.

Implemente *preciosUnicos* sin acceder a la representación del TAD *Tabla* y sin modificar las tablas parámetro.

```
// PRE: t1 y t2 no vacías
Tabla preciosUnicos(Tabla t1, Tabla t2)
```

Parte b)

Defina una representación del TAD *Tabla* *no acotada* (*RepTabla*), en la que las operaciones:

- *crear*, *cantidad*, *minDominio* y *maxDominio* tengan O(1) de tiempo de ejecución en el peor caso;
- *insertar*, *definida*, *recuperar* y *eliminar* tengan O($\log_2(n)$) de tiempo de ejecución en el caso promedio, siendo n la cantidad de correspondencias en la tabla;
- *imprimir* tenga O(n) de tiempo de ejecución en el peor caso, siendo n la cantidad de correspondencias en la tabla.

Explique brevemente cada campo de la representación elegida, que NO debe incluir un árbol AVL. Luego, escriba los **códigos de *crear, insertar e imprimir***; asuma implementadas las restantes operaciones especificadas del TAD (no escriba sus códigos). Considere también que existe la operación *impCorrespondencia (nat d, float c)*, que imprime la correspondencia (*d*,*c*).

SOLUCIONES

Problema 1

Parte a)

Enuncie brevemente las dos propiedades que debe cumplir un árbol binario para ser un *Heap*.

- Propiedad de la estructura: es un árbol binario completamente lleno, con la posible excepción del nivel más bajo, el cual se llena de izquierda a derecha.
- Propiedad de orden: para todo nodo X del árbol, el valor de prioridad del padre de X es menor --si nos basamos en Sets para prioridades-- (menor o igual --si nos basamos en Multisets para prioridades--) que el valor de prioridad de X, con la excepción obvia de la raíz, donde esta el mínimo elemento.

Parte b)

Enuncie brevemente las dos propiedades que debe cumplir un árbol binario para ser un *AVL*.

- Propiedad de la estructura: para cada nodo del árbol, la diferencia entre las alturas de sus subárboles izquierdo y derecho puede diferir en a lo sumo una unidad.

- Propiedad de orden: para cada nodo del árbol los elementos que estén en su subárbol izquierdo deben ser menores y los que estén en su subárbol derecho deben ser mayores.

Problema 2

Parte a)

```
// PRE: t1 y t2 no vacías
Tabla preciosUnicos(Tabla t1, Tabla t2){
      Tabla res = crear();
      nat inf = min (minDominio(t1), minDominio(t2)); // min: mínimo en nat
      nat sup = max (maxDominio(t1), maxDominio(t2)); // max: máximo en nat
      bool def_t1, def_t2;
      float precio;
      for (nat i = inf; i <= sup; i++) {
            def_t1 = definida(i, t1);
            def_t2 = definida(i, t2);
            if (def_t1 && def_t2){
                  precio = recuperar(i, t1);
                  if (precio == recuperar(i, t2))
                        insertar(i, precio, res);
            }
            else if (def_t1)
                  insertar(i, recuperar(i, t1), res);
            else if (def_t2)
                  insertar(i, recuperar(i, t2), res);
      }
      return res;
}
```

Parte b)

```
struct nodoABB{
      nat dom;
      float ran;
      nodoABB * izq;
      nodoABB * der:
}

struct RepTabla{
      nodoABB *  arbol; // ABB para asegurar operaciones O(log2(n)) prom.
      int cantidad; // para tener la cantidad de correspondencias en O(1) pc.
      nodoABB * nodoMinDominio; // para que minDominio sea O(1) pc.
      nodoABB * nodoMaxDominio; // para que maxDominio sea O(1) pc.
}

Tabla crear(){
      Tabla t = new RepTabla;
      t->arbol = NULL;
      t->cantidad = 0;
      t->nodoMinDominio = NULL;
      t->nodoMaxDominio = NULL;
      return t;
}
```

```
void insertar(nat d, float c, Tabla & t){
      nodoABB * insertado = insABB(d, c, t); /* Inserta o actualiza un nodo en
      el ABB. Retorna el puntero al nodo o NULL si es una actualización */
      if (insertado != NULL){
            t->cantidad++;
            if (t->nodoMinDominio == NULL || d < t->nodoMinDominio)
                  t->nodoMinDominio = insertado;
            if (t->nodoMaxDominio == NULL || d > t->nodoMaxDominio)
                  t->nodoMaxDominio = insertado;
      }
}

/* Inserta o actualiza un nodo en el ABB. Retorna el puntero al nodo o NULL
si es una actualización */
nodoABB * insABB(nat d, float c, nodoABB * & a){
      if (a == NULL){
            a = new nodoABB;
            a->dom = d;
            a->ran = c;
            a->izq = a->der = NULL;
            return a; //  retorna el puntero al nodo insertado
      }
      else if (a->dom == d){
            a->ran = c;
            return NULL; //  retorna NULL si actualiza un nodo existente
      }
      else if (d < a->dom)
            return insABB(d, c, a->izq);
      else  return insABB(d, c, a->der);
}

void imprimir(Tabla t){
      imprimirABB(t->arbol); // Impresión en orden del ABB, de mayor a menor
}

// Impresión en orden del ABB, de mayor a menor
void imprimirABB (nodoABB * a){
      if (a != NULL){
            imprimirABB(a->der);
            impCorrespondencia(a->dom, a->ran);
            imprimirABB(a->izq);
      }
}
```