Examen de Programación 2

25 de julio de 2019

Problema 1 (20 puntos)

Considere los árboles AVL de la Figura 1.

(a) Árbol AVL con 4 elementos. (b) Árbol AVL con 6 elementos.

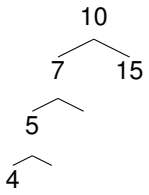
Figura 1: Árboles AVL.

(a) Muestre el árbol resultante de insertar el número 4 en el árbol AVL de la Figura 1a e indique en qué nodo(s) del árbol se viola la propiedad estructural característica de los árboles AVL. Dibuje y explique la secuencia de árboles resultantes de ejecutar la(s) rotación(es) **simple(s)** necesaria(s) para restablecer la condición de `AVL`. Para cada rotación indique en qué nodo se aplica y si es una rotación `izquierda` o `derecha`.

(b) Muestre el árbol resultante de insertar el número 19 en el árbol AVL de la Figura 1b e indique qué nodo(s) del árbol se viola la propiedad estructural característica de los árboles AVL. Dibuje y explique la secuencia de árboles resultantes de ejecutar la(s) rotación(es) **simple(s)** necesaria(s) para restablecer la condición de `AVL`. Para cada rotación indique en qué nodo se aplica y si es una rotación `izquierda` o `derecha`.

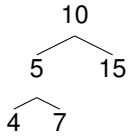
Solución:

(a) El árbol resultante exactamente después de insertar el 4 es:



Los nodos donde no se cumple la propiedad estructural de AVL son: 10 y 7. En ambos casos los subárboles izquierdo aumentan su altura en 1 provocando que la diferencia de alturas con los respectivos subárboles derecho sea igual a 2. 3-1 en el primer caso y 2-0 en el segundo.

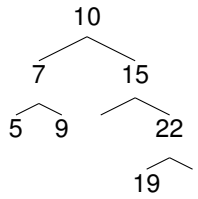
Por lo tanto, debe recuperarse la estructura de árbol AVL primero en 7 (donde se realiza la inserción por último) y luego verificar si se requiere corregir la estructura con relación al 10. Considerando que el desbalanceo en 7 fue generado por una inserción izquierda-izquierda, es necesario hacer una rotación derecha en 7:



El árbol resultante es AVL, de modo que no es necesario aplicar ninguna rotación en 10.

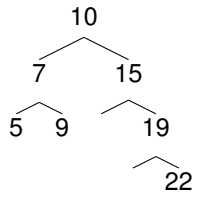
(b)

El árbol resultante exactamente después de insertar el 19 es:

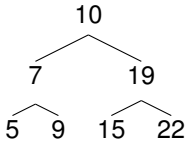


El único nodo donde no se cumple la propiedad estructural de AVL es el nodo 15. Su subárbol derecho pasa a tener altura 2 mientras que su subárbol derecho mantiene su altura 0.

En este caso, dado que la violación se produce a partir de una inserción derecha-izquierda es necesario realizar dos rotaciones: derecha, izquierda.



Árbol desequilibrado después de rotación a la derecha en 22.



Árbol equilibrado después de rotación a la izquierda en 15.

Problema 2 (40 puntos)

Se desea implementar un algoritmo para crear un árbol binario de enteros a partir de un conjunto de árboles pasado como parámetro.

El algoritmo funciona de la siguiente manera: dado un conjunto de árboles, en cada paso se toman los dos árboles con menor valor en su raíz y se crea un nuevo árbol que contiene a estos dos como subárbol izquierdo y derecho **(el que posee la raíz de menor valor se coloca del lado derecho)**. Además, la raíz del nuevo árbol contiene la suma de los valores de los dos subárboles. Luego, el árbol resultante es insertado en el conjunto, de forma de ser considerado por el algoritmo en los próximos pasos. Si el conjunto tiene un sólo árbol, el mismo es retornado como resultado. Ver ejemplo de la Figura 2.

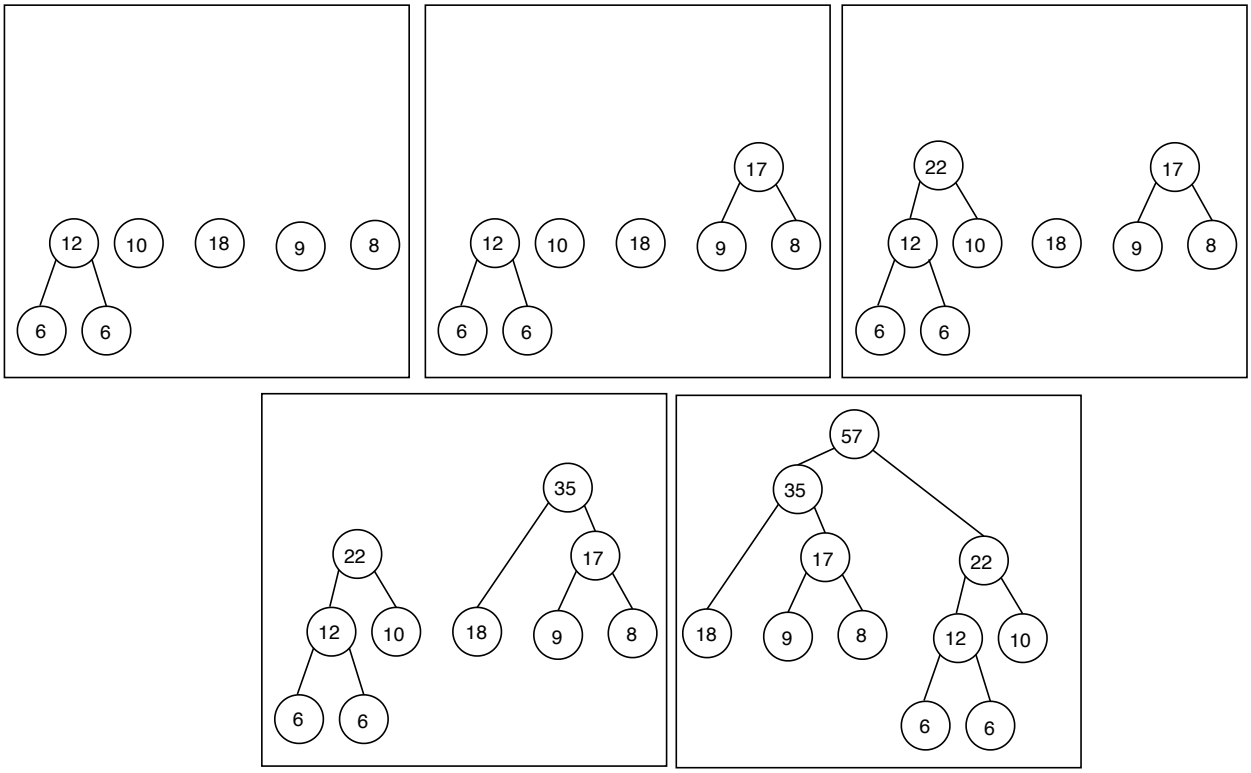


Figura 2: Ejemplo de `crearArbol`.

Considere la siguiente definición de árbol binario.

```
struct nodoAB {
    int valor;
    nodoAB *izq, *der; };
typedef nodoAB * AB;
```

Se pide:

(a) Especifique el `TAD Cola de Prioridad` de elementos de tipo `AB`.

(b) Asumiendo ya implementadas las operaciones del `TAD Cola de Prioridad` de elementos de tipo `AB`. Implemente la operación `crearArbol` que recibe en la cola de prioridad `CP` el conjunto de árboles inicial.

```
//Función que crea un árbol binario a partir de un conjunto de árboles dados en una cola de prioridad.
AB  crearArbol (CP cola);
```

(c) Implemente la operación `imprimirSuma`, que dado un árbol binario, para cada camino de la raíz a una hoja imprime la suma de los valores de sus nodos.

```
//Dado un árbol binario, para cada camino de la raíz a una hoja imprime la suma de los valores de sus nodos.
void imprimirSumaAB (AB a);
```

Solución:

(a)
```
/* Retorna la Cola de prioridad vacía */
ColaPrio crearCP();

/* Inserta el elemento de tipo AB con prioridad p en la cola cp */
void insertarCP (AB elemento, int p, ColaPrio &cp);

/* Verifica si la Cola de prioridad cp esta vacía */
bool esVaciaCP (ColaPrio cp);

/* Retorna el elemento prioritario de cp. Precondicion: !esVaciaCP (cp). */
AB prioritarioCP (ColaPrio cp);

/* Elimina de la Cola de prioridad cp el elemento prioritario. Si esVacia (cp), no ↩
    tiene efecto. */
void eliminarPrioritario (ColaPrio &cp);
```

(b)
```
AB crearArbol (CP cp)
{
  AB superArbol = NULL;
  if (!esVaciaCP (cp))
  {
    //obtengo de la cp el árbol cuya raíz es más chica
    AB subArbolDer = prioritario (cp); // raíz más chica a la derecha
    eliminarPrioritario (cp);
    while (!esVaciaCP (cp))
    {
      //obtengo de la cp el árbol cuya raíz es más chica (mayor que la raíz de ↩
          subArbolDer)
      AB subArbolIzq = prioritario (cp); // raíz más grande a la izquierda
      eliminarPrioritario (cp);

      //creo el nodo con la suma de las dos raíces
      AB nuevoArbol = new nodoAB;
      nuevoArbol->valor = subArbolDer->valor + subArbolIzq->valor;
      nuevoArbol->izq = subArbolIzq;
      nuevoArbol->der = subArbolDer;
      //cargo ese árbol en cp
      insertarCP (nuevoArbol, nuevoArbol->valor, cp);

      //obtengo de la cp el árbol cuya raíz es más chica
      subArbolDer = prioritario (cp); // raíz más chica a la derecha
      eliminarPrioritario (cp);
    }
    superArbol = subArbolDer;
  }
  return superArbol;
}
```

(c)
```
void imprimirSumaAB (AB a)
  if (a != NULL)
    imprimirSumaAux (a, 0);
}

/* PRE: a!=NULL */
void imprimirSumaAux (AB a, int suma)
{
  suma += a->valor; // sumo el valor del nodo actual al parámetro

  if (a->izq == NULL && a->der == NULL);  //si llegué a una hoja
    printf ("%d ", suma);
  else
  {
    if (a->izq != NULL)
      imprimirSumaAux (a->izq, suma);
    if (a->der != NULL)
```

```
            imprimirSumaAux (a->der, suma);
        }
    }
```

Problema 3 (40 puntos)

Se pretende modelar de forma simplificada la asignación de funcionarios a circuitos electorales el día de las elecciones. Cada circuito está identificado por un número en el rango $[0, C]$ y tiene asociado una dirección de tipo `String` y 3 funcionarios que están asignados al circuito el día de las elecciones. Cada funcionario tiene asociado un nombre, un número de cédula de identidad, el circuito en el que vota y el circuito al que está asignado el día de las elecciones. Se sabe que el número de funcionarios en el sistema será $N = 3(C + 1)$ cómo máximo.

Algunas de las operaciones del sistema necesarias son: crear la estructura de datos con la información de circuitos, asignar funcionarios a circuitos, retornar el número de funcionarios asignados a un circuito, imprimir los funcionarios de un circuito, retornar el número de circuito al que está asignado un funcionario, imprimir la información de los funcionarios asignados.

Considere la siguiente definición del tipo `Asignaciones` (ver Figura 3):

```
//Nodos de árbol binario de búsqueda. Cada nodo representa un funcionario.
//El árbol está ordenado por nombre del funcionario.
struct nodoABBF {
  String nom;
  int ci ;
  int circFAsig;
  int circFVota;
  nodoABBF *izq;
  nodoABBF *der;
};
typedef nodoABBF *ABBFuncionarios;

//Nodos para representar un circuito, Identificados por número del circuito.
//Tiene un puntero a cada funcionario asignado a ese circuito.
struct nodoCirc {
  String dir;
  nodoABBF *f1, *f2, *f3;
};

//Nodos para lista de punteros a funcionarios (nodos del árbol).
//Se utilizará para implementar hash abierto.
struct nodoLista{
  nodoABBF *f;
  nodoLista *sig;
};
typedef nodoLista* ListaFuncCI;

//Multiestructura de Asignaciones. Contiene:
//-Árbol Binario de Búsqueda de funcionarios
//-Arreglo de circuitos
//-Hash Abierto de funcionarios (por CI).
struct rep_Asignaciones {
  ABBFuncionarios funcionarios;
  nodoCirc *circuitos ;
  ListaFuncCI *hashFuncCI;
};

typedef struct rep_Asignaciones * Asignaciones;
```

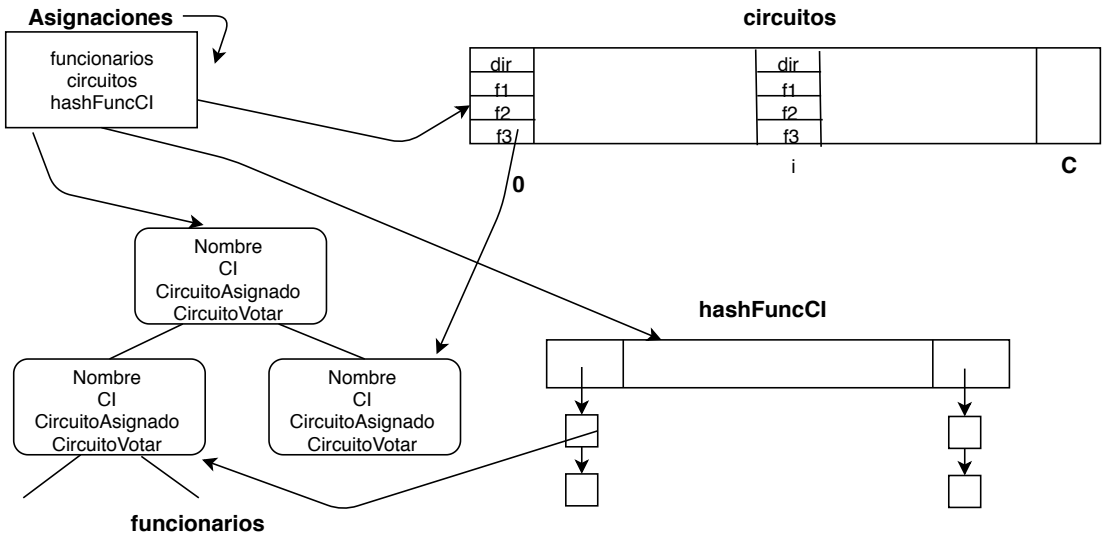


Figura 3: Multiestructura para representar a los funcionarios asignados a los circuitos electorales.

(a) 1. Implemente la siguiente operación del TAD Asignaciones:

```
//Crea un sistema para asignar hasta N funcionarios a C circuitos electorales con ↩
    identificador en el rango [0,C].
//El array 'circuitos' es de largo C+1 y contiene la dirección de cada circuito. La ↩
    posición en el array identifica a cada circuito.
//Devuelve un TAD Asignaciones con circuitos sin funcionarios asignados.
Asignaciones crearAsig(int C, String *circuitos);
```

2. Explique cuál es el orden del tiempo de ejecución para el peor caso.

(b) Implemente la siguiente operación del TAD Asignaciones, de forma que el orden del tiempo de ejecución sea O(1) en promedio. Asuma implementada una función $h : int \rightarrow [0..N-1]$.

```
//Devuelve verdadero si el circuito al que está asignado el funcionario con cédula ci es ↩
    el mismo en el que vota.
//Precondición: existe el funcionario con ci asignado a algún circuito en A.
bool coincide(int ci, Asignaciones A);
```

(c) 1. Implemente un procedimiento que imprima, en orden alfabético, los nombres de los funcionarios que NO están asignados al mismo circuito en el que votan.

2. Explique cuál es el orden del tiempo de ejecución para el peor caso.

Solución:

(a)
```
Asignaciones crearAsig (int C, String *circuitos)
{
  Asignaciones asig = new rep_Asignaciones;
  asig->funcionarios = NULL;
  asig->circuitos = new nodoCirc[C+1];
  asig->hashFuncCI = new ListaFuncCI[3*(C+1)];

  for (int i = 0; i <= C, i++)
  {
    asig->circuitos[i].f1 = NULL;
    asig->circuitos[i].f2 = NULL;
    asig->circuitos[i].f3 = NULL;
    asig->circuitos[i].dir = circuitos[i];
  }

  for (int i = 0; i <= 3*C, i++)
    asig->hashFuncCI[i] = NULL;
```

```
    return asig;
  }
```

Dado que se deben recorrer los dos arreglos, uno con C elementos y otro con N=3*(C+1) elementos, el órden del tiempo de ejecución es O(4*C+3) = O(C).

(b)
```
bool coincide (int ci, Asignaciones a)
{
  ListaFuncCI lista = a->hashFuncCI[h(ci)];
  //Asumo que el funcionario existe.
  while (lista->f->ci != ci)
    lista = lista->sig;
  nodoABBF *func = lista->f;
  return func->circFVota == func->circFAsig;
}
```

Asumiendo que la función de hash se comporta adecuadamente y que no hay mas de N funcionarios en el sistema, el órden del tiempo de ejecución en el caso promedio es O(1).

(c)
```
void imprimirFuncDistintos (Asignaciones a)
{
  imprimirFuncDistintosAux (a->funcionarios)
}

void imprimirFuncDistintosAux (ABBF arbol)
{
  if (arbol != NULL)
  {
    imprimirFuncDistintosAux (arbol->izq);
    if (arbol->circFVota != arbol->circFAsig)
      printf ("%s", arbol->nom);
    imprimirFuncDistintosAux (arbol->der);
  }
}
```

Dado que se deben recorrer todos los nodos del árbol para hacer la comparación y decidir si imprimir o no, el órden del tiempo de ejecución en el peor caso es O(3*(C+1))=O(N), donde N es la cantidad de nodos del árbol (que coincide con la cantidad de funcionarios en el sistema).