ESTRUCTURAS DE DATOS Y ALGORITMOS

CURSO 2009 – PRÁCTICO 5

SOLUCIÓN

Ejercicio 1

a) Dar una representación para árboles binarios de naturales (BinaryTree).

```
struct BinaryTree {
            int elem;
            BinaryTree* izq;
            BinaryTree* der;
};
```

b) Implementar las siguientes operaciones funcionales:

```
/* Devuelve el árbol vacío*/
BinaryTree* NullTree() {
      return NULL;
}

/* Crea un árbol no vacío a partir de un natural y otros dos árboles*/
BinaryTree* ConsTree(int x, BinaryTree *l, BinaryTree *r) {
      BinaryTree *arbol = new BinaryTree;
      arbol->elem = x;
      arbol->izq = l;
      arbol->der = r;

      return arbol;
}

/* Determina si un árbol dado es o no vacío */
bool IsEmptyTree(BinaryTree *t) {
      return (t == NULL);
}

/* Devuelve el valor en la raíz de un árbol no vacío */
int RootTree(BinaryTree *t) {
      return t->elem;
}

/* Devuelve el subárbol izquierdo de un árbol no vacío */
BinaryTree* LeftTree(BinaryTree *t) {
      return t->izq;
}

/* Devuelve el subárbol derecho de un árbol no vacío */
BinaryTree* RightTree(BinaryTree *t) {
      return t->der;
}
```

Ejercicio 2 – Parte a)

Análisis del problema

Se deben implementar las recorridas *EnOrden, PreOrden* y *PostOrden* sobre un árbol binario generando listas ordenadas según el criterio utilizado para recorrer un árbol. No se puede acceder directamente a la representación de los arboles ni de las listas.

En cada paso se genera una lista asegurando que, el orden en que se colocan las sublistas generadas al recorrer cada uno de los sub arboles y la raíz, correspondan con el criterio utilizado para recorrer el árbol.

Para generar la lista en cada paso se necesitan dos funciones auxiliares: *Append* para concatenar dos listas y *Snoc* para insertar un elemento al final de una lista.

Implementación (C/C++)

A continuación se definen las funciones auxiliares a utilizar:

```
LNat* Append(LNat *l1, LNat *l2) {
      if (IsEmpty(l1))
            return l2;
      else
            return (Cons(Append(Tail(l1), l2), Head(l1)));
}

LNat* Snoc(int x, LNat *l) {
      if (IsEmpty(l)) {
            return Cons(Null(), x);
      } else {
            return Cons(Snoc(x, Tail(l)), Head(l));
      }
}
```

A continuación se definen las funciones que implementan cada una de las recorridas:

```
// izquierdo, raiz, derecho
LNat* InOrder(BinaryTree *t) {
      if (IsEmptyTree(t))
            return Null();
      else
            return Append(InOrder(LeftTree(t)),
Cons(InOrder(RightTree(t)),
                        RootTree(t)));
}

// raiz, izquierdo, derecho
LNat* PreOrder(BinaryTree *t) {
      if (IsEmptyTree(t))
            return Null();
      else
            return Append(Cons(PreOrder(LeftTree(t)), RootTree(t)),
                        PreOrder(RightTree(t)));
}
```

```
// izquierdo, derecho, raiz
LNat* PostOrder(BinaryTree *t) {
      if (IsEmptyTree(t))
            return Null();
      else
            return Append(PostOrder(LeftTree(t)), Snoc(RootTree(t),
                        PostOrder(RightTree(t))));
}
```

Ejercicio 3 – Parte a)

```
int max(int a, int b) {
      if (a>b)
            return a;
      else
            return b;
}

//Retorna la distancia máxima desde la raíz a un nodo + 1
int Height(BinaryTree *t) {
      if (t == NULL)
            return 0;
      else {
            int maxizq = Height(t->izq);
            int maxder = Height(t->der);
            int maximo = max(maxizq, maxder);

            return 1+ maximo;
      }
}

/* Retorna la cantidad de nodos del árbol */
int CountElements(BinaryTree *t) {
      if (t == NULL)
            return 0;
      else
            return 1 + CountElements(t->izq) + CountElements(t->der);
}

/* Número de subárboles izquierdos no vacíos */
int SINV(BinaryTree *t) {
      if (t!=NULL) {
            if (t->izq!=NULL)
                  return 1 + SINV(t->izq) + SINV(t->der);
            else
                  return SINV(t->der);
      } else
            return 0;
}
```

```
/* ídem anterior para subárboles derechos */
int SDNV(BinaryTree *t) {
      if (t!=NULL) {
            if (t->der!=NULL)
                  return 1 + SDNV(t->izq) + SDNV(t->der);
            else
                  return SDNV(t->izq);
      } else
            return 0;
}

/* Cantidad de hojas, es decir nodos sin hijos */
int NH(BinaryTree *t) {
      if (t != NULL) {
            if ((t->izq == NULL) && (t->der == NULL))
                  return 1;
            else
                  return NH(t->izq) + NH(t->der);
      } else {
            return 0;
      }
}

/* cantidad de nodos con algún hijo */
int NT(BinaryTree *t) {
      if (t != NULL) {
            if ((t->izq != NULL) || (t->der != NULL))
                  return 1 + NT(t->izq) +NT(t->der) ;
            else
                  return 0;
      } else {
            return 0;
      }
}

/* Cantidad de nodos con exactamente 2 hijos*/
int NE2H(BinaryTree *t) {
      if (t != NULL) {
            if ((t->izq != NULL) && (t->der != NULL))
                  return 1 + NT(t->izq) +NT(t->der);
            else
                  return NT(t->izq) + NT(t->der);
      } else {
            return 0;
      }
}
```

Ejercicio 5

a) La representación –estructura- para ABB (o BST por sus siglas en ingles) es la misma utilizada en los Binary Tree, por lo que:

```
struct BST {
      int elem;
      BST* izq;
      BST* der;
};
```

b) La implementación es análoga a la de Binary Tree, con la excepción de la inserción la cual debe ser ordenada (los elementos menores a la raíz se insertan en el subárbol izquierdo mientras que aquellos elementos mayores a la raíz se ubican en el subárbol derecho)

```
BST* InsertBST(int x, BST *st) {
      BST *b =st;
      if (st == NULL) {
            b= new BST;
            b->elem = x;
            b->der = b->izq = NULL;
      } else if (st->elem > x) {
            // inserto a la izq
            b->izq = InsertBST(x, b->izq);
      } else {
            b->der = InsertBST(x, b->der);
      }

      return b;
}
```

c)

```
void InsertBST(int x, BST &* st) {
      if (st == NULL) {
            st = new BST;
            st->elem = x;
            st->der = st->izq = NULL;
      } else if (st->elem > x)
            InsertBST(x, st->izq);
      else
            InsertBST(x, st->der);
}
```

Ejercicio 6 - RemoveMinBST

```
void RemoveMinBST(BST* t) {
      BST *aux;

      if (t->izq == NULL) {
            aux = t;
            t=t->der;
            delete aux;
      } else
            RemoveMinBST(t->izq);
}
```

Ejercicio 6 - IsElementBST

```
bool IsElementBST(int x, BST *bs) {
      if (bs == NULL)
            return false;
      else {
            if (bs->elem == x)
                  return true;
            else {
                  if (x < bs->elem)
                        return IsElementBST(x, bs->izq);
                  else
                        return IsElementBST(x, bs->der);
            }
      }
}
```

Ejercicio 6 - RemoveBST

Análisis del Problema

Se debe implementar una función que elimine un valor de un árbol de búsqueda (ABB).

Lo primero a tener en cuenta es que el árbol resultante de la eliminación de dicho valor debe seguir siendo un ABB, lo cual implica que debe cumplir con la propiedad de orden entre los valores de sus nodos.

A la hora de borrar un cierto valor **X** de un ABB **A**, suponiendo que exista en **A** un nodo con valor **X**, se pueden considerar tres casos:

1. Que el subárbol izquierdo del árbol que tiene a **X** en la raíz sea vacio.
2. Que el subárbol derecho del árbol que tiene a **X** en la raíz sea vacio.
3. Que ni el subárbol izquierdo ni el derecho del árbol que tiene a **X** en la raíz sean vacios.

En el primer caso, como el subárbol izquierdo es vacio se puede concluir que en ese subárbol todos los nodos contienen elementos mayores que **X**. Por otra parte el subárbol derecho es un ABB (por definición) y por lo tanto cumple con la propiedad de orden. Entonces basta con reemplazar el subárbol que tiene a **X** en la raíz por el subárbol derecho del mismo.

Análogamente, en el segundo caso, se reemplaza el subárbol que tiene a **X** en el raíz por el subárbol izquierdo del mismo.

En el tercer caso se debe encontrar otro valor, en alguno de los dos sub arboles, que mantenga la propiedad de orden. Como para todo nodo se debe cumplir que los valores del subárbol izquierdo sean menores que el valor de la raíz y los valores del subárbol derecho sean mayores que los de la raíz existen, para cualquier ABB, dos candidatos a reemplazar el valor **X**: el mayor de los menores (el máximo del izquierdo) o el menor de los mayores (el mínimo del derecho). Ambas estrategias son equivalentes, y por lo tanto se necesita una función que devuelva el máximo o el mínimo de un árbol.

Una vez hallado el valor **Y** con el cual reemplazar a **X** debo eliminar el valor **Y** de su anterior ubicación, para lo cual se utiliza la propia función *RemoveBST*.

Implementación (C/C++)

Se cuenta con la siguiente especificación de árbol binario de búsqueda (Binary Search Tree o BST).

```
typedef struct BST {
            int elem;
            BST* izq;
            BST* der;
};
```

A continuación se define la función auxiliar a utilizar:

```
// el valor maximo es el valor mas a la derecha en el
// arbol, por lo tanto que no tiene subarbol derecho.
int MaxBST(BST *t) {
      if (t->der == NULL)
            return t->elem;
      else
            return MaxBST(t->der);
}
```

A continuación se define la función que implementa la eliminación de un valor dado:

```
void RemoveBST(int x, BST &*t) {
      BST *aux;

      if (t != NULL) {
            if (t->elem == x) {
                  // caso 1
                  if (t->izq == NULL) {
                        aux = t;
                        t=t->der;
                        delete aux;
                  }
                  // caso 2
                  else if (t->der == NULL) {
                        aux = t;
                        t=t->izq;
                        delete t;
                  }
                  // caso 3
                  else {
                        t->elem = MaxBST(t->izq);
                        // t->elem = MinBST(t->der)
                        RemoveBST(t->elem, t->izq);
                        // RemoveBST(t->elem,t->der)
                  }
            } else if (x < t->elem)
                  RemoveBST(x, t->izq);
            else
                  RemoveBST(x, t->der);
      }
}
```

Ejercicio 7 – Parte a)

Análisis del Problema

Se debe determinar si dos árboles contienen los mismos elementos. Para esto, se generara, a partir de una recorrida en *PreOrden* de cada árbol, dos listas con sus elementos. Al ser arboles binarios de búsqueda se puede inferir que la recorrida retorna listas ordenadas. Una vez obtenidas las listas se comparan con la función *AreEqualLists* para ver si son iguales, si son iguales los dos arboles contienen los mismos elementos y por lo tanto son iguales.

La implementación (C/C++)

A continuación se definen las funciones auxiliares a utilizar:

```
// retorna una lista resultado de concatenar l1 y l2
// esta funcion genera alias y modifica la lista l1
List * Append(List *l1, List *l2) {
	List *iter;

	if (l1 == NULL)
		return l2;
	else if (l2 == NULL)
		return l1;
	else {
		iter = l1;
		while (iter->next != NULL)
			iter = iter->next;
		iter->next=l2;
	}
	return l1;
}

// retorna TRUE unicamente si l1 y l2 son iguales
bool AreEqualList(List *l1, List *l2) {

	if ((l1==NULL) && (l2==NULL))
		return true;
	else if (l1==NULL || l2==NULL || (l1->data != l2->data))
		return false;
	else
		return AreEqualList(l1->next, l2->next);
}

List* ListABB(BST *ab) {
	List *l;

	if (ab == NULL)
		return NULL;
	else {
		l = new List;
		l->data=ab->elem;
		l->next=NULL;
		return Append(ListABB(ab->izq), Append(l, ListABB(ab->der)));
	}
}
```

La siguiente es la función principal que resuelve el problema:

```
bool AreEqualSets(BST *ab1, BST *ab2) {
      return AreEqualList(ListABB(ab1), ListABB(ab2));
}
```

Ejercicio 7 – Parte b)

Análisis del Problema

Se debe definir una función que dado un árbol binario de búsqueda (ABB) de naturales y un natural *k* retorne la lista de elementos que se encuentran en el nivel *k* ordenados de mayor a menor. Si el nivel *k* no existe se debe devolver la lista vacia.

Para resolver el problema, se genera la lista de la siguiente manera:

- Si *k*=1 el nivel a imprimir es el actual
  - La lista generada por *ListLevel* es una lista con el elemento ubicado en la raz.
- Sino
  - Decremento k porque voy a "bajar" un nivel.
  - Obtengo las listas generadas por *ListLevel* sobre los subárboles izquierdo y derecho (LI y LD respectivamente).
  - Se retorna la concatenación de LD y LI. Por la propiedad de orden de los ABB todos los elementos de LD son mayores que los elementos de LI, por lo tanto la lista está ordenada de mayor a menor

La implementación (C/C++)

Este es el código que implementa la solución:

```
// retorna la lista ordenada de forma descendente de los
// elementos que se encuentran en el nivel k del arbol.
// En caso de no existir el nivel, retorna la lista vacia.
List* ListLevel(int k, BST *ab) {
      List *l;

      if (ab == NULL)
            return NULL;
      else if (k == 1) {
            l = new List;
            l->data=ab->elem;
            l->next=NULL;
            return l;
      } else
            return Append(ListLevel(k-1, ab->der),
                          ListLevel(k-1, ab->izq));
}
```

Ejercicio 10 – Parte a)

```
LNat* CaminoMaximo(BST *t) {
      LNat *LI, *LD;

      if (t == NULL)
            return Null();
      else {
            LI = CaminoMaximo(t->izq);
            LD = CaminoMaximo(t->der);
            if (Largo(LI) >= Largo(LD))
                  return Cons(LI, t->elem);
            else
                  return Cons(LD, t->elem);
      }
}
```