Ejercicios Árboles

Ejercicio 1. Recorridos. Implementa las siguientes funciones:

a) Función de impresión preorder:

```
void tree_preOrder(FILE *f, BSTree *tree, print_element_fn print_elem){
  if (!f || !tree) return;

  return _tree_preorderRec(f, tree->root, print_elem);
}

void _tree_preOrderRec(FILE *f, BTNode *pn, print_element_fn print_elem){
  if (!f || !pn) return;

  print_elem(f, pn->info);
  _tree_preOrderRec(f, pn->left, print_elem);
  _tree_preOrderRec(f, pn->right, print_elem);

  return;
}
```

b) Función de impresión postorder:

```
void tree_postOrder(FILE *f, BSTree *tree, print_element_fn print_elem){
  if (!f || !tree) return;

  return _tree_postOrderRec(f, tree->root, print_elem);
}

void _tree_postOrderRec(FILE *f, BTNode *pn, print_element_fn print_elem){
  if (!f || !pn) return;

  _tree_postOrderRec(f, pn->left, print_elem);
  _tree_postOrderRec(f, pn->right, print_elem);
  print_elem(f, pn->info);

  return;
}
```

c) Función de impresión inorder:

```
void tree_inOrder(FILE *f, BSTree *tree, print_element_fn print_elem){
  if (!f || !tree) return;

  return _tree_inOrderRec(f, tree->root, print_elem);
}

void _tree_inOrderRec(FILE *f, BTNode *pn, print_element_fn print_elem){
  if (!f || !pn) return;

  _tree_inOrderRec(f, pn->left, print_elem);
  print_elem(f, pn->info);
  _tree_inOrderRec(f, pn->right, print_elem);

  return;
}
```

Ejercicio 2. Liberación de árbol.

a) Implementa una función que libere toda la memoria de un árbol y de sus elementos:

```
void tree_destroy(BSTree *tree, process_element free_fn){
  if (!tree || !tree->root) return;

  _btnode_freeRec(tree->root, free_fn);
```

```
  free(tree);
}

void _btnode_freeRec(BTNode *pn, process_element free_fn){
  if (!pn) return;

  _btnode_freeRec(pn->left, free_fn);
  _btnode_freeRec(pn->right, free_fn);
  free_fn(pn->info);
  btnode_free

  return;
}
```

b) ¿Cómo se llamaría a esta función para un árbol con elementos de tipo cadena, `(char *)`?

```
tree_destroy(tree, free);
```

Ejercicio 4. Construcción de ABdBs. Implementa una función que, usando `tree_insert`, añada todos los elementos de un `array` (de tamaño `size`) de puntero a void al ABdB `tree`.

```
Status tree_from_array(BSTree *tree, void **array, size_t size){
  int i;

  if (!tree || !array || size < 0) return ERROR;

  for (i=0; i<size; i++){
    if (!tree_insert(tree, array[i])){
      tree_free(tree);
      return ERROR;
    }
  }

  return OK;
}
```

Ejercicio 8. Propiedades de árboles. Implementa las siguientes funciones:

a) Función que devuelve la cantidad de nodos de un árbol binario:

```
int tree_numNodes(const BTree *tree){
  if (!tree || tree_isEmpty(tree)) return -1;

  return _tree_numNodesRec(tree->root);
}

int _tree_numNodesRec(BTNode *pn){
  if (!pn) return 0;

  return 1 + _tree_numNodesRec(pn->left) + _tree_numNodesRec(pn->right);
}
```

b) Función que devuelve el número de hojas de un árbol binario:

```
int tree_numLeafs(const BTree *tree){
  if (!tree) return -1;

  return _tree_numLeafsRec(tree->root);
}

int _tree_numLeafsRec(BTNode *pn){
  if (!pn) return 0;
  if (!pn->left && !pn->right) return 1;

  return _tree_numLeafs(pn->left) + _tree_numLeafs(pn->right);
```

c) Función que devuelve la suma de los nodos de un árbol binario de elementos de tipo entero:

```
int tree_sum(const BTree *tree){
  if (!tree) return -1;

  return _tree_sumRec(tree->root);
}

int _tree_sumRec(BTNode *pn){
  if (!pn) return 0;

  return *(int*)pn->info + _tree_sumRec(pn->left) + _tree_sumRec(pn->right);
}
```

d) Función que devuelve la profundidad de un árbol binario:

```
int tree_depth(const BTree *tree){
  if (!tree || tree_isEmpty(tree)) return -1;

  return _tree_depthRec(tree->root);
}

int _tree_depthRec(BTNode *pn){
  if (!pn) return 0;

  return 1 + max(_tree_depthRec(tree->left),_tree_depthRec(tree->right));
}

int max(int n1, int n2){
  if (n1 > n2) return n1;
  if (n1 < n2) return n2;

  return n1;
}
```

Ejercicio 9. Recorrido con profundidad acotada. Implementa un algoritmo que imprima todos los nodos de un árbol en preorden hasta una profundidad dada

```
void tree_preorder_depth_bounded(FILE *f, const BSTree *tree, int max_depth, print_element_fm print_elem){
  if (!f || !tree || max_depth < 0) return;

  _tree_preorder_depth_boundedRec(tree->root, 0, max_depth, print);
}

void _tree_preorder_depth_boundedRec(FILE *f, BTNode *pn, int current_depth, int max_depth, print_element_fn print_elem){
  if (!f || !pn || max_depth < 0 || current_depth < 0 || current_depth > max_depth) return;

  print_elem(f, pn->info);
  if (current_depth < max_depth){
    _tree_preorder_depth_boundedRec(f, pn->left, current_depth+1, max_depth, print_elem);
    _tree_preorder_depth_boundedRec(f, pn->right, current_depth+1, max_depth, print_elem);
  }
}
```

Ejercicio 10. Profundidad de un elemento. Implementa una función que devuelva la profundidad a la que se encuentra un elemento en un árbol binario de búsqueda, o -1 si no se encuentra.

```
int tree_elementDepth(const BSTree *tree, const void *elem, cmp_element_fn cmp_elem){
  if (!tree || !elem) return -1;

  return _tree_elementDepthRec(tree->root, elem, 0, cmp_elem);
}

_tree_elementDepthRec(BTNode *pn, const void *elem, int depth, cmp_element_fn cmp_elem){
  if (!pn) return -1;

  if (cmp_elem(pn->info, elem) < 0)
    return _tree_elementDepthRec(pn->left, elem, depth + 1, cmp_elem);
```

```
  else if (cmp_elem(pn->info, elem) > 0)
    return _tree_elementDepthRec(pn->right, elem, depth + 1, cmp_elem);
  else
    return depth;
}
```

Ejercicio 11. Copia de árbol. Implementa una función que devuelve una copia de un árbol, copiando también todos sus elementos con una función `copy_elem`:

```
BSTree *tree_copy(const BSTree *tree, copy_element_fn copy_elem){
  BSTree *copy = NULL;

  if (!tree) return NULL;

  copy = tree_init(tree->cmp_element);
  if (!copy) return NULL;

  copy->root = _tree_copyRec(tree->root, copy_elem);
  return copy;
}

BSNode _tree_copyRec(BTNode *pn, copy_element_fn copy_elem){
  BTNode *new = NULL;

  if (!pn) return NULL;
  new = node_new();
  if (!new) return NULL;

  new->info = copy_elem(pn->info);
  new->left = _tree_copyRec(pn->left, copy_elem);
  new->right = _tree_copyRec(pn->right, copy_elem);

  return new;
}
```

Ejercicio 14. Mínimo y máximo. Implementa las siguientes funciones:

a) Función que devuelve un puntero al elemento mínimo de un ABdB de forma recursiva:

```
void *tree_minElement(const BSTree *tree){
  if (!tree) return NULL;

  return _tree_minElementRec(tree->root);
}

void *_tree_minElementRec(BTNode *pn){
  if (!pn) return NULL;
  if (!pn->left) return pn->info;

  return _tree_minElementRec(pn->left);
}
```

b) Función que devuelve un puntero al elemento mínimo de un ABdB de forma no recursiva:

```
void *tree_minElement_it(const BSTree *tree){
  BTNode *pn = NULL;

  if (!tree) return NULL;

  pn = tree->root;
  while (pn->left)
    pn = pn->left;

  return pn->info;
}
```

c) Función que devuelve un puntero al elemento máximo de un ABdB de forma recursiva:

```
void *tree_maxElement(const BSTree *tree){
  if (!tree) return NULL;

  return _tree_maxElementRec(tree->root);
}

void *_tree_maxElementRec(BTNode *pn){
  if (!pn) return NULL;
  if (!pn->right) return pn->info;

  return _tree_maxElementRec(pn->right);
}
```

d) Función que devuelve un puntero al elemento máximo de un ABdB de forma no recursiva:

```
void *tree_maxElement_it(const BSTree *tree){
  BTNode *pn = NULL;

  if (!tree) return NULL;

  pn = tree->root;
  while (pn->right)
    pn = pn->right;

  return pn->info;
}
```