Práctica 4 - Árboles

Árboles Binarios

1. Lea la implementación provista de *árboles binarios enlazados* para datos enteros, y el ejemplo presentado en el archivo `test.c`. Asegúrese comprenderlo.

2. *Búsqueda en profundidad*:

a) Complete la definición de la función `btree_recorrer`. Esta función debe recorrer el árbol mediante el algoritmo de búsqueda en profundidad, llamando a la función visitante en el dato de cada nodo del árbol. La forma de visitar los datos (esto es, *preorden*, *inorden*, *postorden*) se indica mediante una variable enumerada cuya valor está dado por:

```
typedef enum {
  BTREE_RECORRIDO_IN,
  BTREE_RECORRIDO_PRE,
  BTREE_RECORRIDO_POST
} BTreeOrdenDeRecorrido;
```

b) Pruebe su implementación utilizando como función visitante `imprimir_entero` de `test.c`.

c) Diseñe una versión iterativa para el caso preorden. *Ayuda:* puede utilizar una pila general para guardar los nodos a visitar.

3. Extienda la implementación con las siguientes funciones:

a) `btree_nnodos`: que retorne el número de nodos del árbol.

b) `btree_buscar`: que retorne 1 si el número dado se encuentra en el árbol, y 0 en caso contrario.

c) `btree_copiar`: que retorne un nuevo árbol que sea una copia del árbol dado.

d) `btree_altura`: que retorne la altura del árbol.

e) `btree_nnodos_profundidad:` que retorne el número de nodos que se encuentran a la profundidad dada.

f) `btree_profundidad`: que retorne la profundidad del nodo que contiene el número dado, y -1 si el número no se encuentra en el árbol.

g) `btree_sumar`: que retorne la suma total de los datos del árbol.

4. Considerando la siguiente declaración:

```
typedef void (*FuncionVisitanteExtra) (int dato, void *extra);
```

a) Implemente la función `btree_recorrer_extra`. Su comportamiento es similar a `btree_recorrer`, salvo que ahora la función visitante es de tipo `FuncionVisitanteExtra` y se deberá aplicar tanto al dato almacenado en el nodo como al dato extra. Su prototipo es el siguiente:

```
void  btree_recorrer_extra(BTree arbol, BTreeOrdenDeRecorrido orden,
                           FuncionVisitanteExtra visit, void *extra);
```

b) Piense cuáles de las funciones del ejercicio anterior se podrían definir de manera compacta llamando a `btree_recorrer_extra` con un orden, una función visitante y un dato extra apropiados.

5. *Búsqueda por extensión*:

a) Piense cómo se podrían definir funciones que recorran el árbol usando el algoritmo de búsqueda por extensión, tanto de manera iterativa como recursiva. ¿Detecta alguna ventaja/desventaja entre ellas? *Ayuda*: para el caso iterativo puede utilizar una cola general para guardar los nodos a visitar y para el caso recursivo puede definir una función auxiliar que visite solo los nodos que se encuentren a una profundidad determinada.

b) Desarrolle una implementación para alguna de ellas. El prototipo de la función será:

```
void  btree_recorrer_bfs(BTree arbol, FuncionVisitante visit);
```

6. Generalice la implementación de los árbol binarios para que los nodos guarden datos de tipo `void *`. Procure proveer las mismas operaciones que en el ejercicio 1, ajustando los prototipos según lo requiera.

Árboles de Búsqueda Binaria (ABB)

7. Dibuje el resultado de insertar en un ABB vacío la siguiente secuencia de números:

$$10, 20, 15, 25, 30, 16, 18, 19.$$

Dibuje además el resultado de eliminar del mismo el número 20.

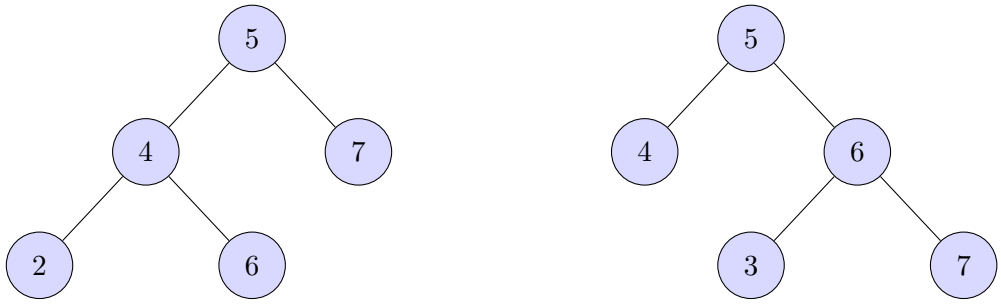
8. Lea la implementación provista de árboles de búsqueda binaria y el ejemplo presentado en el archivo `test.c`. Asegúrese comprenderlo.

9. Implemente una función `bstree_eliminar` que elimine, si es posible, un elemento del árbol. Deberá garantizar que el árbol resultante sea nuevamente un ABB. Su prototipo es:

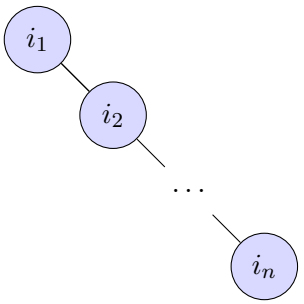
```
BSTree bstree_eliminar(BSTree arbol, void *dato,
                       FuncionComparadora, FuncionDestructora);
```

10. Implemente `bstree_k_esimo_menor` que retorne el k-ésimo menor elemento del ABB, con k parámetro. Intente no usar estructuras auxiliares. *Ayuda:* piense qué recorrido del árbol (preorden, inorden, postorden) procesa los elementos de menor a mayor.
Mencione qué información adicional podría almacenar en los nodos que permita realizar esta operación de manera más eficiente. Es decir, que en cada llamada sea posible saber en cuál de los subárboles está el nodo buscado, evitando explorar ambos.

11. Implemente una función `btree_validar` que determine si el árbol dado cumple la propiedad de los ABB. *Ayuda:* piense si es suficiente con validar recursivamente que los subárboles sean ABB. Por ejemplo, los siguientes árboles no son ABB a pesar de que sus subárboles lo son, ¿por qué?



12. Dé un ejemplo de una secuencia de n números $i_1, i_2, \ldots, i_n$ tales que al ser insertados en un ABB vacío, el árbol resultante tenga en realidad estructura de lista:



Analice la cantidad de nodos explorados por la función `bstree_buscar` cuando el ABB tiene esta estructura, y compárelo con un ABB de n nodos cuya estructura sea la de un árbol completo. ¿Qué puede concluir sobre la eficiencia de `bstree_buscar` cuando el árbol se encuentra *desbalanceado* en altura?

Árboles AVL

13. Resuelva los siguientes puntos:

a) Dibuje el resultado de insertar en un AVL vacío la siguiente secuencia de números:

$$10, 20, 15, 25, 30, 16, 18, 19.$$

Compare la estructura del árbol resultante con aquella obtenida con árboles de búsqueda binaria en el ejercicio 7 de la parte I. Dibuje además el resultado de eliminar del mismo el número 30.

b) Dé un orden posible para la secuencia de números del punto anterior, de modo que al insertarlos en un árbol AVL vacío se obtenga el mismo árbol pero sin haber producido ninguna rotación.

c) Dé un ejemplo de un árbol AVL para el cual, tras eliminar uno de sus elementos, se deban aplicar más de una rotación (simple o doble) para balancearlo.

14. Lea la implementación provista de árboles AVL y el ejemplo presentado en el archivo `test.c`. Asegúrese comprenderlo.

15. Complete el archivo `avl.c` con definiciones de `avl_nodo_rotacion_simple_der` y `avl_nodo_insertar`. Utilice los casos de prueba de `test.c` para controlar que las mismas sean correctas.

16. Implemente una función `avl_eliminar` que elimine un elemento del árbol. Deberá garantizar que el árbol resultante sea nuevamente un árbol AVL.

17. Vimos que almacenar la altura del árbol AVL en la estructura del nodo, hace más eficiente el cómputo del factor de balance. Siguiendo esta idea, ¿será posible aumentar la estructura guardando también el factor de balance en el nodo? Piense como se debería actualizar este valor tras realizar rotaciones, inserciones y eliminaciones. Por último, piense si es posible mantener en la estructura únicamente el factor de balance, en lugar de la altura.