Práctica 4 - Árboles

Árboles Binarios:

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

2. En cada caso escriba una función que:

a Calcule la suma de los elementos de un árbol de enteros.

b Cuente la cantidad de nodos de un árbol de enteros.

c Calcule la altura de un árbol de enteros.

3. Implemente la función `btree_recorrer` que utilice el recorrido *primero en profundidad*. Agregue como parámetro un elemento del siguiente tipo, que determine el orden del recorrido:

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

Considerar una posible implementación recursiva y una iterativa (utilizando una pila).

4. Considere la siguiente definición:

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

a Implemente la función `btree_recorrer_extra` con un comportamiento similar al de la función `btree_recorrer`. Ahora, la función visitante se deberá aplicar, en cada momento, tanto al dato almacenado como al dato extra. Su signatura será la siguiente:

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

b) Reimplemente el ejercicio 2 utilizando `btree_recorrer_extra`. ¿Es esto posible para todos los ítems del ejercicio?

5. Implemente la función `btree_recorrer_bfs` que utilice el recorrido *primero por extensión* Ayuda: puede utilizar una cola para guardar los nodos a visitar.

6. Escribir una función `mirror` que dado un árbol binario, genere el árbol binario espejo, donde el hijo derecho de cada nodo pasa a ser izquierdo y el izquierdo pasa a ser derecho.

Árboles Binarios de Búsqueda:

7. Escribir un tipo de datos `BSTree` que implemente *árboles binarios de búsqueda* (enlazados). Diseñe y programe las siguientes funciones:

- `bstree_insertar` que agregue un elemento a un árbol binario de búsqueda.
- `bstree_eliminar` que elimine un elemento de un árbol binario de búsqueda.

- `bstree_contiene` que determine si un elemento está en un árbol binario de búsqueda.
- `bstree_nelementos` que devuelva la cantidad de elementos de un árbol binario de búsqueda.
- `bstree_altura` que devuelva la altura de un árbol binario de búsqueda.
- `bstree_recorrer` que aplique una función en cada nodo de un árbol binario de búsqueda.

Indique qué condición es necesaria para que tras agregar varios elementos, el árbol resultante tenga en realidad forma de lista. Haga un análisis de la cantidad de nodos que hay que visitar en el caso de buscar un elemento en esa lista, y compárelo con un árbol que no tenga la forma de lista.

8. Indique cuál de los posibles órdenes de recorrido primero en profundidad (mencionados en el ej. 3) permite recorrer los elementos de un árbol binario de búsqueda de menor a mayor.
Implementar `bstree_imprimir` que imprima los elementos de forma ordenada.

9. Implementar `bstree_minimo` que permita obtener el mínimo elemento de un árbol binario de búsqueda, de manera eficiente.

10. Implementar `bstree_acceder` que dado un índice y un árbol binario de búsqueda, retorne el elemento en esa posición (Ayuda: utilizar un recorrido in-order).
Mencione qué información podría almacenar en los nodos que permita realizar esta operación de manera más eficiente.

Heaps Binarios y Colas de Prioridad

11. Implementar *heaps binarios* (árboles binarios completos) utilizando arreglos. Utilizar la siguiente estructura:

```
typedef struct _BHeap {
  int datos[MAX_SIZE];
  int nelems;
} *BHeap;
```

Procure que la interfaz provea las siguientes funciones:

- `bheap_crear` que crea un nuevo heap.
- `bheap_destruir` que recibe un heap y lo destruye.
- `bheap_es_vacio` determina si el heap está vacío.
- `bheap_insertar` que agregue un elemento a un heap en la siguiente posición disponible.
- `bheap_nelementos` que devuelve la cantidad de elementos en el heap.
- `bheap_recorrer` que recorra los nodos primero en profundidad aplicando la función dada en cada nodo.
- `bheap_recorrer_bfs` que recorra los nodos primero por extensión aplicando la funcion dada en cada nodo ¿Es necesario utilizar una cola en este caso?

12. Reimplementar `bheap_insertar` para que mantenga la *propiedad de max-heap*: los datos de los nodos están ordenados de arriba hacia abajo, es decir, el dato de todo nodo (diferente a la raíz) es menor o igual al dato de su padre. Agregar las siguientes funciones a la interfaz:

a `bheap_maximo`: toma un heap y devuelve el mayor elemento.

b `bheap_eliminar_maximo`: toma un heap y borra su mayor elemento.

13. Implementar `heapify(int arr[], size_t tamanio)` que transforme un arreglo dado en un heap con la propiedad de max-heap.

14. Proveer una implementación para *colas de prioridad* utilizando:

1) Arreglos circulares ordenados.

2) Lista enlazadas ordenadas.

3) Heaps binarios.

La misma deberá contar con las siguientes funciones:

a `int cola_prioridad_es_vacia(PCola)` : determina si la cola está vacía.

b `int cola_prioridad_maximo(PCola)` : obtiene el elemento prioritario.

c `void cola_prioridad_eliminar_maximo(PCola)` : quita el elemento prioritario.

d `void cola_prioridad_insertar(PCola, int)` : inserta un elemento con determinada prioridad.

¿Cuáles son las ventajas y desventajas de cada implementación, en relación a la eficiencia de cada función?

Árboles AVL:

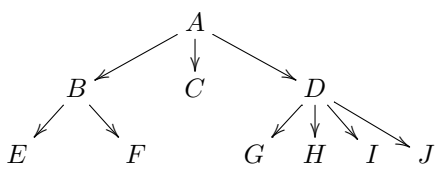
15. Reimplementar bstree_insertar y bstree_eliminar para que mantengan el árbol balanceado en altura, es decir, que su resultado sea un árbol AVL.

Árboles Generales:

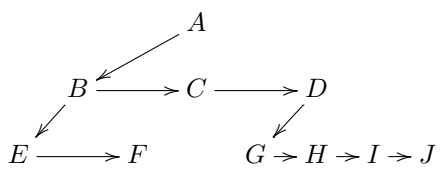
16. Un *árbol rosa (rose tree)* es un árbol general, donde cada nodo puede tener un número arbitrario de hijos representado a través de una lista.
Implemente las funcionalidades básicas utilizando listas enlazadas generales (`void *`) para almacenar la lista de nodos hijos.

17. En clase hemos visto que todo árbol con un número arbitrario de hijos puede implementarse como un árbol binario. Cada nodo almacena un puntero a su primer hijo y uno a su primer hermano. A continuación podemos observar un ejemplo de este tipo de árboles (figura **a.**) y su representación como árbol binario (figura **b.**).

a.



b.



a De forma similar a árboles binarios, defina una estructura (llamémosla MTree, por árbol de múltiples hijos) para esta nueva forma de representar árboles generales.

b Diseñe e implemente las siguientes funciones:

- `mtree_crear`, `mtree_destruir` análogas a las correspondientes a árboles binarios.
- `mtree_agregar_n_hijo` que dado un dato, agregue una nodo hijo en la posición indicada. Si la posición recibida es mayor al número de hijos, procure que el nuevo nodo hijo se agregue al final.
- `mtree_recorrer` que implemente alguno de los recorridos vistos.
- `mtree_recorrer_extra` análoga a `mtree_recorrer` pero que lleve datos extra.
- `mtree_cantidad`, `mtree_altura`, utilizando `mtree_recorrer_extra`.

c Implemente una función que tome un RoseTree y retorne el MTree equivalente.