Práctica 4 - Árboles: parte II

Heaps Binarios

1. Dibuje el resultado de insertar en un heap binario[1] vacío la siguiente secuencia de números:

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

Dibuje además el resultado de eliminar del mismo el número 30.

2. Implemente heaps binarios generales utilizando arreglos dinámicos. Puede definir su propia estructura o utilizar la que se presenta a continuación. La misma guarda un arreglo dinámico y su capacidad, el último índice ocupado del arreglo, y un puntero a una función de comparación.

```
typedef struct _BHeap {
  void **arr;
  int capacidad;
  int ultimo;
  FuncionComparadora comp;
} *BHeap;
```

Procure que la interfaz provea las siguientes funciones:

a) `bheap_crear`: que cree un heap vacío con una capacidad y una funcion de comparación dadas.

b) `bheap_destruir`: que destruye el heap.

c) `bheap_es_vacio`: que retorne 1 si el heap está vacío y 0 en caso contrario.

d) `bheap_recorrer`: que recorre los nodos utilizando búsqueda por extensión, aplicando la función dada en cada elemento. ¿Es necesario utilizar una cola auxiliar en este caso?

e) `bheap_insertar`: que agregue un elemento al heap, realocando el arreglo en caso de ser necesario. El resultado debe ser nuevamente un heap binario.

f) `bheap_eliminar` que elimine un elemento del heap. El resultado debe ser nuevamente un heap binario.

3. Implemente una función

```
BHeap bheap_crear_desde_arr(void **arr, int largo, FuncionCopiadora copiar,
                            FuncionComparadora comp)
```

que a partir de un arreglo arbitrario cree un heap binario. Intente no hacer uso de `bheap_insertar`.

4. Escribir un programa que utilice un heap binario para ordenar un arreglo de enteros. Pruebe su programa con un arreglo con 10000 números aleatorios.

5. Implemente *colas de prioridad* utilizando heaps binarios. Deberá proveer las siguientes operaciones. Procure manipular el heap únicamente a través de las funciones declaradas en su interfaz:

[1] Consideramos un heap binario como un árbol binario completo con la propiedad de que las claves están ordenadas de arriba hacia abajo. En otras palabras, el padre debe tener siempre mayor clave que sus hijos.

a) `cola_prioridad_es_vacia`: que retorne 1 si la cola está vacía y 0 en caso contrario.

b) `cola_prioridad_maximo`: que retorne el elemento prioritario de la cola.

c) `cola_prioridad_eliminar_maximo`: que elimine el elemento prioritario de la cola.

d) `cola_prioridad_insertar`: que inserte un elemento en la cola con una determinada prioridad.