Cola de proridad.

Antes de explicar que son las colas de prioridad se explicará que es un heap, montículo, debido a que es una de las mejores estructuras para implementar una cola de prioridad.

Un heap es una estructura de datos que almacena un árbol binario completo. Además, el árbol debe cumplir la siguiente propiedad: para cada nodo su valor debe ser menor al valor de sus hijos en un *Min Heap*.

Para representar un heap como un arreglo se toman las siguientes convenciones: 1) el nodo que está en la posición **i** tiene su hijo izquierdo en la posición **2i** y el hijo derecho en la posición **2i+1**; 2) el padre de **i** está en la posición **entero(i/2)**.

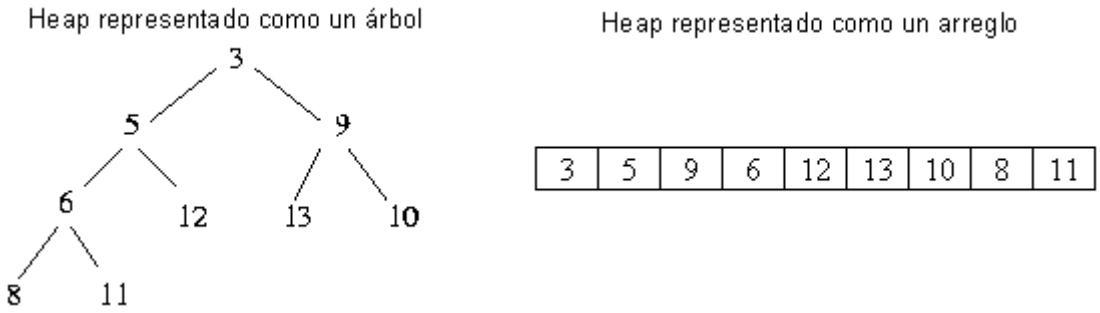

Figura 1. Un Min Heap

Debido a que en un *min heap* el valor de cada nodo es menor que el valor de sus hijos, esto nos garantiza que el nodo raíz es el menor de los elementos contenidos en el heap; si usamos un arreglo para representar el heap, el nodo raíz es el primer elemento del arreglo.

Para obtener el siguiente menor elemento del heap seguimos los siguientes pasos: 1) quitamos la raíz del heap, 2) convertimos el último nodo en la raíz del heap, y 3) restauramos la propiedad del heap. Si usamos un arreglo, el paso 1 y 2 consiste en reemplazar el primer elemento del arreglo por el último y disminuir en uno el tamaño del arreglo.

La cola de prioridad es entonces una estructura que almacena objetos con una cierta prioridad asociada, y que permite extraer el elemento de menor prioridad. Algunas ventajas de usar de una cola de prioridad son:

- Permite disponer rápidamente del elemento con menor prioridad.
- Es dinámica, porque permite insertar y quitar elementos en tiempo O(log N) (donde N es la cantidad de elementos en la cola), ejecutando las operaciones en cualquier order. Contrapuesto a un vector ordenado, que una vez creado, para insertar un elemento tiene un coste es O(log N).
- En una cola de prioridad (ordenada para mínimo), es fácil obtener el elemento de mínima prioridad mínimo pero muy costoso obtener el de máxima.

Las operaciones más frecuentes de una cola de prioridad son:

- **Crear** (heap_init) una nueva cola de prioridad
```
int heap_init(HEAP* h, long size);
```

- **Destruir** (heap_delete) la cola
```
void heap_delete(HEAP* h);
```

- **Inserción** (heap_addItem) de un elemento en la cola
```
int heap_addItem(HEAP* h, ITEM* p);
```

- **Consulta** (heap_queryMin) del elemento con menor prioridad (no se quita del heap)
```
ITEM* heap_queryMin(HEAP* h);
```

- **Extracción** (heap_extractMin) del elemento con menor prioridad (se quita del heap)
```
ITEM* heap_extractMin(HEAP* h);
```

- **Actualización** (heap_updateItem) de la prioridad de un elemento que ya esta en la cola
```
void heap_updateItem(HEAP* h, ITEM *p);
```

- **Eliminación** (heap_delItem) de un elemento que esta en la cola
```
void heap_delItem(HEAP* h, ITEM *p);
```

- **Estado** (heap_isEmpty, heap_isFull) del heap
```
int heap_isEmpty(HEAP *h);
int heap_isFull(HEAP*h);
```

Como crear y usar una cola de prioridad, paso a paso:

El siguiente instructivo explica como adaptar y utilizar la cola de prioridad. También en se incluye un ejemplo de utilización en `heaptest.c`.

0) Asegurarse de incluir `heap.h` al comienzo del programa principal

```
#include "heap.h"
```

1) Definir los objetos que se almacenaran en la cola.

Los objetos (ITEM) son administrados por el usuario. O sea el usuario tiene que crearlos y destruirlos, la cola de prioridad solamente permite accederlos en orden.
Pero para que el la cola de prioridad pueda utilizarlos es necesario que tengan algunos campos específicos: FITNESS e idx.

a) En `heap.h` localizar la declaración de la estructura ITEM

```
/* HEAP ITEM STRUCTURE
 * the ITEM structure can customized by adding new fields
 *
 * FITNESS: is to the cost/fitness value used to sort the heap
   idx    : is an internal index that should not be modified */

typedef struct
{
   /* ALL YOUR DATA HERE */
   int label;

   /* PRIVATE DATA */
   float FITNESS;       /* cost/fitness key*/
   int idx;             /* internal index : DON'T TOUCH*/
} ITEM;
```

Personalizar la estructura agregando los campos que se deseen en la parte marcada como `/* ALL YOUR DATA HERE */` en este caso nos interesa solamente almacenar una etiqueta (**label**) de tipo entero.
NOTE: que entre los campos privados **FITNESS** es el que usaremos para indicar la prioridad del elemento en cuestión.
NOTE: no se puede cambiar el nombre del campo **FITNESS**.
NOTE: **idx** es un campo de uso interno de la estructura y no deberá alterarse en ningún momento.

b) Una vez personalizada la estructura, se deberán crear los elementos ITEM.

Supongamos que solo disponemos de 200 objetos, entonces declaramos en el programa principal:

```
ITEM items[200];
```

2) Declarar e inicializar el objeto cola:

```
HEAP cola;
heap_init(&cola, 200);
```

3) Asignar una prioridad a los items e insertarlos en la cola:

```
items[4].FITNESS = 2150.9;
items[4].label = 1996
heap_addItem(&cola,&(items[4]));
```

Repetir tantas veces como sea necesario.
NOTA: dado que la cola tiene un tamaño máximo no se pueden insertar más de 200 elementos en la cola. Se puede verificar si la cola esta llena con: `heap_isFull(&cola)` (retorna: 1(FULL), 0(NOT FULL))
NOTA: esta operacion tiene un coste del orden O(log N), donde N es la cantidad de elementos en la cola.

4) Extraer o Consultar el elemento con menor FITNESS de la cola.

Extract y Query devuelven un puntero al objeto que tiene el menor FITNESS,por tanto deberemos declarar un puntero. Extract remueve el elemento de la cola, mientras que Query solo consulta.

```
ITEM *p_item;
p_item = heap_queryMin(&cola);
printf("label: %d\n",p_item->label);
p_item = heap_extractMin(&cola);
printf("label: %d\n",p_item->label);
```

Repetir tanto como sea necesario.
NOTA: esta operación remueve el elemento de la cola, por lo que si la cola esta vacía ya no se puede ejecutar. Para verificar esto se puede usar: `heap_isEmpty(&cola)` (retorna: 1(EMPTY), 0(NOT EMPTY))

5) destruir la cola (luego de finalizar su uso).

```
heap_delete(&cola);
```