Montículos (Heap)

Objetivos.

- Aprender a realizar una implementación acotada de un montículo.
- Aprender a implementar el algoritmo heapsort.
- Profundizar en el uso de templates c++. En especial usar parámetros templates por defecto.
- Aprender a usar funtores (Functional objects) de la STL de c++.
- Aprender a usar el tipo `std::unique_ptr<>`.

Descripción.

En esta práctica, en primer lugar, hay que desarrollar una implementación acotada del TAD Heap como la vista en clase. Para permitir que la la misma implementación sirva tanto para un montículo de mínimo como de máximo, será utilizada una clase Functor como parámetro del template. Por defecto, este parámetro template tendrá el valor std::less_equal y, eso quiere decir que se obtendrá un montículo de mínimo por defecto.

En segundo lugar hay que implementar el algoritmo HeapSort [ver aquí] usando el template Heap. Su complejidad dependerá de cómo diseñes el algoritmo “heapify” en la clase Heap. Debes intentar que la complejidad de HeapSort sea O(N Log(N)) (sobre todo para pasar el segundo caso de prueba).

Hay dos formas de construir un heap. La primera desde un estado vacío e ir insertando los elementos siendo el Heap el responsable de liberar el espacio asignado de forma dinámica cuando se destruya. Esta forma será utilizada para implementar una cola de prioridad más adelante en otra práctica.

La segunda forma de construir el heap es para ordenar un vector. En este caso el espacio de almacenamiento será externo (el vector a ordenar).

Para soportar las dos formas de trabajar, se pueden usar dos atributos, por ejemplo, `my_storage_` y `data_` (si el alumno encuentra otra forma mejor o similar puede usar la suya).

Cuando el heap se crea vacío con una capacidad dada, `my_stogage_` y `data_` apuntan a un nuevo bloque de memória dinámica de elementos T con la capacidad indicada. Luego, cuando se destruya el heap, se deberá liberar esta memoria. Para gestionar esta memoria dinámica, se recomienda que el atributo `my_storage_` sea del tipo

`std::unique_ptr<T>` (que será el propietario de la memoria asignada) y el atributo `data_` sea del tipo `T*` y apunte al primer elemento del bloque de memoria creado. Así cuando se destruya el heap, ya se habrá destruido el atributo `my_storage_` liberando la memoria asignada. El atributo `data_` se usará para acceder a los elementos como si fuera un vector.

Por otro lado, si queremos usar el heap para ordenar un vector, este vector será el almacenamiento externo, por lo que el atributo `my_storage_` se dejará nulo (su valor por defecto) y `data_` apuntará al primer valor del vector externo. Por supuesto el heap quedará en un estado “full”. Luego, cuando se destruya el heap, no se liberará memoria alguna ya que `my_storage_` es nulo. La memoria apuntada por `data_` es propiedad de otro objeto externo: el vector ordenado.

Como se puede ver, internamente en el heap usaremos `data_[]` para acceder a los elementos T independientemente de la forma en que se ha construido el heap.

En este enlace [1] hay un interesante discusión sobre las diferencias entre los tipos `std::shared_ptr` y `std::unique_ptr`.

Referencias.

[1]
https://stackoverflow.com/questions/6876751/differences-between-unique-ptr-and-shared-ptr