# Trabajo Práctico:
# Compresor por Huffman

Estructuras de Datos
Tecnicatura en Programación Informática
Universidad Nacional de Quilmes

## Introducción

Un compresor por el método de Huffman es un algoritmo de compresión de datos que genera una codificación representando elementos comunes con cadenas binarias cortas y elementos raros con cadenas binarias más largas. La codificación de Huffman posee numerosas propiedades que la hacen idóneas para aplicar en un compresor y es la base de muchas de las aplicaciones de compresión más utilizadas. En el presente trabajo se pide implementar tres estructuras de datos, que permiten hacer la compresión por el método de Huffman de forma eficiente.

Este enunciado está acompañado por archivos base provistos por la cátedra entre los cuales se encuentra la funcionalidad de lectura a archivo, escritura a archivo y compresión:

- `HuffmanCoding.cpp` archivo con el punto de entrada del proyecto (función main) y el algoritmo de compresión por el método de Huffman.
- `ZipTable.h` y `ZipTable.cpp` archivos que proveen la funcionalidad de una cadena de bits (`BitChain`) y la tabla de traducciones extraída a partir de un árbol de Huffman (`ZipTable`).

Se proveen también los archivos de encabezado para cada una de las estructuras que deben ser implementadas:

1. `CharBag.h` encabezado de un multiconjunto de caracteres.
2. `HuffmanTree.h` encabezado de un árbol de Huffman.
3. `PriorityQueue.h` encabezado de una cola de prioridad.

## 1. CharBag

Un `CharBag` es un multiconjunto de caracteres (una estructura en donde no importa el orden en el que se insertan los elementos, pero que a diferencia de los conjuntos permite almacenar elementos repetidos). Se pide implementar esta estructura en el archivo `CharBag.cpp`, indicando la estructura de representación y sus invariantes e incluyendo la implementación de todas las operaciones declaradas en `CharBag.h` cumpliendo la complejidad pedida.

Se recomienda basar la implementación de esta estructura en una representación similar a la vista para tablas de Hash. Note que entre la funcionalidad requerida se pide dar un iterador de la estructura que debe recorrer una vez cada elemento contenido en el multiconjunto indicando su cardinalidad. No es necesario que el iterador respete un orden, pero sí que avance al siguiente elemento en orden constante (es decir independiente de la capacidad del `CharBag` y la cantidad de elementos contenidos en él).

Se pide, también, entregar los tests usados para verificar la correctitud de la implementación en el archivo `CharBagTest.cpp`. Los tests deben ejercitar toda la funcionalidad de la estructura y verificar que los resultados obtenidos son correctos con respecto al propósito declarado.

## 2. HuffmanTree

Un `HuffmanTree` es un árbol binario cuyas hojas tienen asociadas un caracter y un peso (entero). Diremos que el peso de un árbol de Huffman es la suma de los pesos de todas sus hojas. Las ramas del árbol codifican una cadena binaria con la codificación comprimida del caracter almacenado en la hoja (cada hijo izquierdo agrega un bit en 0 a la codificación, mientras que cada hijo derecho agrega un bit en 1). El método de compresión de Huffman (dado en el archivo `HuffmanCoding.cpp`) construye este árbol de forma tal que la codificación generada por sus ramas sea optima. Se pide implementar esta estructura en el archivo `HufmanTree.cpp`, indicando la estructura de representación, sus invariantes e incluyendo la funcionalidad de todas las operaciones declaradas en `HuffmanTree.h` cumpliendo la complejidad pedida.

Note que parte de la funcionalidad del `HuffmanTree` requiere usar las estructuras provistas por la cátedra (`BitChain` y `ZipTable`). Estas estructuras deben ser utilizadas únicamente a través de las interfaces provistas sin alterar ningún aspecto del código.

Se pide, también, entregar los tests usados para verificar la correctitud de la implementación en el archivo `HuffmanTreeTest.cpp`. Los tests deben ejercitar toda la funcionalidad de la estructura y verificar que los resultados obtenidos son correctos con respecto al propósito declarado. Considere agregar funciones auxiliares si necesita verificar el estado de la estructura más allá de lo que la interfaz permite observar.

## 3. PriorityQueue

Una `PriorityQueue` es una cola donde los elementos tienen una prioridad asignada, es decir, es una colección de datos que permite acceder al elemento de mayor prioridad eficientemente. Para lograr esto es necesario mantener fuertes invariantes de representación y que las operaciones de inserción (`enqueue`) y remoción (`dequeue`) de elementos aprovechen y mantengan esos invariantes. En este trabajo se pide implementar una cola donde los elementos a almacenar serán `HuffmanTree`s y donde el árbol de menor peso se considerará prioritario, es decir, a menor peso mayor prioridad. Se pide implementar esta estructura en el archivo `PriorityQueue.cpp`, indicando la estructura de representación y sus invariantes, incluyendo la implementación de todas las operaciones declaradas en `PriorityQueue.h` cumpliendo la complejidad pedida.

Se recomienda basar la implementación de esta estructura en una representación similar a la vista para Heaps. Observe que sólo poseen restricciones de complejidad las operaciones `enqueue` y `dequeue`, lo que permite mantener la estructura simple. Considere crear funciones auxiliares que le permitan manipular la estructura fácilmente.

Se pide, también, entregar los tests usados para verificar la correctitud de la implementación en el archivo `PriorityQueue.cpp`. Los tests deben ejercitar toda la funcionalidad de la estructura y verificar que los resultados obtenidos son correctos con respecto al propósito declarado.

## Consejos

C++ es un lenguaje muy poderoso, pero también complejo y con el que es fácil equivocarse. Nuestras sugerencias para evitar y/o minimizar dificultades son:

- **No procastinar:** Empezar con tiempo y no dejarlo para último momento.
- **No dudar en consultar:** Las instancias intermedias del TP no son objeto de corrección, por lo que es preferible consultar ante la menor duda.
- **Probar código seguido:** Después de resolver cada función escribir el test correspondiente y no dejarlo para el final, la detección temprana de errores disminuye el esfuerzo.