# Trabajo práctico 1 de Diseño y Análisis de Algoritmos

Alejandro Mujica

- **Lee completamente este documento antes de comenzar a trabajar o formular preguntas.**
- **Esta asignación es evaluada y la calificación que obtengas cuenta para tu calificación definitiva.**
- **Fechas válidas de entrega: desde el 23/10/2018 hasta el 28/10/2018.**
- **El rango de fechas de entrega no será cambiado bajo ninguna circunstancia. Toma tus precauciones al respecto.**
- **Tu entrega consiste de un archivo llamado `Huffman.H`, Por favor, al inicio de este archivo, en comentario, pon tu número de cédula, nombre y apellido.**

## 1. Introducción

El objetivo de este trabajo práctico es que aprehendas una de las aplicaciones de los algoritmos Ávidos y el uso de las estructuras de datos adecuadas para este tipo de algoritmos.

Para la solución de esta asignación debes hacer revisión de la documentación de la Standar Template Library (STL) del lenguaje de Programación `C++`. Sobretodo los tipos `vector`, `priority_queue` y `map`. También sería bueno hacer la revisión de la documentación del tipo `string`.

El problema que resolverás aquí es la codificación de archivos por el algoritmo de Huffman aplicado a texto plano. Al terminar, podrás codificar un archivo de texto plano según el árbol óptimo de Huffman y también podrás decodificarlo para recuperar el texto original.

## 2. Trabajo práctico

Para la realización de este trabajo, se te provee un archivo llamado `defs.H`, el cual contiene la definición de las abstracciones que utilizarás para solucionar el problema. **NO modifiques este archivo, pero estúdialo bien**. Las abstracciones que se te proveen en el archivo son las siguientes:

- `HuffmanNode` es un tipo que representa un nodo para árboles binarios con clave de tipo `string`.
- `HeapInfo` es un alias que se la da al tipo concreto `tuple<HuffmanNode *, unsigned long>`. Este tipo almacena en el primer elemento la raíz de un árbol de Huffman y como segundo elemento almacena la suma total de la frecuencia de los símbolos que este (el árbol) alberga. Este es el tipo de dato que almacenará el `Heap` que se utilizará de apoyo en la construcción del árbol óptimo de Huffman.
- `HeapCmp` es el tipo que funge de criterio de comparación para ordenar el `Heap`.Usa como clave para ordenamiento el valor de la frecuencia en la raíz albergada.
- `Heap` es un alias al tipo concreto `priority_queue` con clave `HeapInfo` y criterio de comparación `HeapCmp`.
- `eof` cadena constante con el pseudo fin de archivo.
- `byte` es un alias que se le da a un entero sin signo de 8 bits.
- `byte_set` es un alias que se le da al tipo `vector` con `byte` como argumento plantilla.
- `BitSet` es una clase que permite manejar colecciones de bits mediante el tratamiento de una colección de bytes. Este tipo está diseñado con las operaciones esenciales para este trabajo. Un constructor vacío; un método `size` para consultar la cantidad de bits almacenada; un método `add_bit` que recibe como arguemento un valor 0 o 1 para añadirlo al final de la secuencia; `get_bit` que recibe un entero como argumento y retorna el valor del bit ubicado en la posición dada por el argumento; `save` y `load` reciben un flujo de salida y de entrada de archivos respectivamente y se encargan de almacenar y recuperar los bits en y de la memoria secundaria. Finalmente, se provee el operador de flujo de salida << para este tipo de dato.

También, se te provee un archivo llamado `helpers.H`, el cual contiene algunas funciones de ayuda que podrás utilizar para solucionar el problema. **Nuevamente, NO modifiques este archivo, pero estúdialo bien**. Las operaciones que se te proveen en el archivo son las siguientes:

- `K`, `L` y `R`. Todas reciben como argumento un apuntador a nodo de árbol de Huffman y retornan referencias a su clave, apuntador al hijo izquierdo y apuntador al hijo derecho respectivamente.
- `destroy_tree` es una función que recibe la raíz de un árbol de Huffman y libera toda la memoria.
- `is_leaf` es una función que recibe un apuntador a un nodo de árbol de Huffman y retorna `true` si el nodo es una hoja y `false` en caso contrario.

- `prefix_traverse`, `infix_traverse` y `sufix_traverse` son funciones de orden superior las cuales efectúan recorridos en prefijo, infijo y sufijo respectivamente sobre un árbol de Huffman. Reciben como parámetros la raíz del árbol y una operación que se efectuará sobre cada nodo del árbol.

- `tree_to_bits` es una función que recibe como argumento la raíz de un árbol de Huffman y retorna un objeto de tipo `BitSet` con el árbol codificado en bits (0 por cada nodo externo y 1 por cada nodo interno). Nota que esta función no almacena las claves del árbol, solo la forma.

- `bits_to_tree` es una función que recibe como argumento un objeto de tipo `BitSet` con un árbol de Huffman codificado en bits, reconstruye el arbol y retorna el apuntador a su raíz. Nota que esta operación no recupera claves, solo la forma del árbol.

  `load_text` y `save_text` las cuales, respectivamente, recuperan de y almacenan de la memoria secundaria el texto plano.

  `load_encoded_text` y `save_encoded_text` las cuales, respectivamente, recuperan de y almacenan de la memoria secundaria el texto codificado.

Se te provee, además, un archivo llamado `draw_tree_utils.H` el cual contiene una función llamada `draw_Huffman_tree`. Esta función escribe sobre un flujo de salida la información necesaria para dibujar el árbol de Huffman con el programa `btreepic` el cual se te provee (ejecuta `btreepic --help` para leer la documentación de sus argumentos). Recibe como parámetros un apuntador la raíz del árbol óptimo de Huffman y una referencia al flujo de salida sobre el cual va a escribir.

También se te provee un archivo llamado `test.C` el cual contiene algunos ejemplos de pruebas de las operaciones que debes resolver. Escribe más pruebas en este archivo para cubrir todos los casos posibles.

Para probar de una manera más entretenida el funcionamiento de tu solución, está el archivo `compressor.C` el cual te permite codificar o decodificar un archivo dado como entrada y almacena el resultado en un archivo dado como salida. A este programa le puedes indicar con cual función partirá el texto y otra opción te permitirá generar el archivo para dibujar el árbol de Huffman generado mediante `btreepic`. Para compilar este programa, se requiere que tengas instalada la biblioteca Tclap. Ejecuta `./compressor --help` para observar la documentación de los parámetros requeridos.

Tienes un `Makefile` para compilar al cual debes ajustarle el compilador según el que estés utilizando.

Finalmente se te provee el archivo denominado `Huffman.H` el cual contiene la plantilla de las rutinas que debes programar. El archivo contiene la instrumentación de las operaciones vacías, éstas retornas valores sin sentido. Tu trabajo es programarlas para que retornen los valores correctos. Las rutinas son las siguientes:

- `filter_string` la cual recibe una cadena (texto) y debe retornar una nueva cadena donde todos los símbolos de espacio sean sustituidos por el símbolo `'~'` y todos los saltos de línea sean sustituidos por el símbolo `'|'`.

- `unfilter_string` la cual recibe una cadena y retorna una nueva revirtiendo los cambios efectuados por `filter_string`.

- `split_by_char` (se te provee resuelta) la cual recibe un texto filtrado y lo retorna particionado por carácter. Cada carácter del resultado queda almacenado en un arreglo de cadenas.

- `split_two_by_two` la cual recibe un texto filtrado y lo retorna particionado por pares de caracteres. Cada pareja de caracteres del resultado queda almacenado en un arreglo de cadenas.

- `split_three_by_three` la cual recibe un texto filtrado y lo retorna particionado por tripletas de caracteres. Cada tripleta de caracteres del resultado queda almacenada en un arreglo de cadenas.

- `build_freq_table` la cual recibe un arreglo de cadenas y retorna la tabla de frecuencia (representada mediante el tipo `map`) de aparición de cada cadena en el arreglo.

- `init_heap` la cual recibe como parámetro una tabla de frecuencias y debe construir un `Heap` cuyo contenido sean raíces unitarias a árboles de Huffman que almacenen la cadena dada por la clave de la tabla y la frecuencia dada como valor en la tabla (revisa nuevamente el tipo `HeapInfo` descrito anteriormente).

- `build_optimal_tree` la cual recibe un `Heap` previamente iniciado con árboles de Huffman y las frecuencias asociadas. Aquí debes construir el árbol óptimo de Huffman y retornar su raíz. Para efectos de la evaluación y poder comparar la igualdad de tu árbol con el mío, sigue la siguiente convención: Cuando extraigas la raíz con menor frecuencia del heap, ésta será la raíz del sub árbol izquierdo del nuevo árbol que se formará. De la misma manera, al sacar la segunda raíz menor será la raíz del sub árbol derecho del nuevo árbol. Recuerda que ese proceso lo efectuarás hasta que el heap almacene un solo árbol y que por cada iteración extraerás los dos árboles con menores frecuencias. **Ayuda: No olvides que el árbol con menor frecuencia se encuentra en la raíz (tope) del heap.**

- `build_path_table` la cual recibe un árbol de Huffman y debe construir una tabla en donde se mapearán las cadenas albergadas en cada hoja del árbol con una cadena que conforme el camino en bits (ceros y unos) para llegar hasta esa hoja. La convención a seguir aquí es utilizar un `'0'` cuando se haga un desplazamiento a un sub árbol izquierdo y un `'1'` cuando se haga un desplazamiento a un sub árbol derecho. La tabla se instrumenta con una instancia de `map<string, string>`.

- `encode` Con las operaciones anteriores, ya solo queda utilizarlas para codificar el texto. Esta operación recibe como argumentos una cadena con el texto y una función de partición para el texto. Aquí deberás filtrar el texto, partirlo y luego codificarlo llamando las operaciones definidas anteriormente en un orden correcto. No debes olvidar añadir el fin de archivo para efectos de una correcta

recuperación del texto original. También recuerda que este fin de archivo no debe pasar por el proceso de partición del texto; piensa bien en cual será la línea idónea para agregarlo. Debe retornar una tupla (revisa `make_tuple`) cuyo primer elemento es el arreglo de bits y el segundo elemento es la raíz del árbol de Huffman.

- `decode` es inversa a la anterior. Recibe el texto codificcado y la raíz del árbol de Huffman utilizado para codificarlo. Debe retornar el texto descifrado (no olvides utilizar `unfilter_string` al final y que en el texto recuperado no deberá estar el fin de archivo).

# 3. Evaluación

La fecha de entrega de este trabajo es desde el 23/10/2018 hasta el 28/10/2018.

Tienes permitido enviar tu práctica máximo una vez por día. Es decir, desde el día de inicio hasta el día final de la práctica tienes disponibles 3 intentos. Estos no son acumulativos. Si llegas a los 3 días de finalizar la práctica, cada día que no envíes será un intento que desperdiciarás. Entonces te quedarían solamente 2 intentos disponibles.

Para evaluarte debes enviar el archivo `Huffman.H` a la dirección:

`alejandro.j.mujic4@gmail.com`

El "subject" debe ser **exactamente** el texto **"AYDA-LAB-01"** sin las comillas. Si fallas con el subject entonces probablemente tu trabajo no será evaluado, pues el mecanismo automatizado de filtrado no podrá detectar tu trabajo. No comprimas el archivo y no envíes nada adicional a este.

El único contenido que debe aparecer en el correo es tu número de cédula y tu nombre separados por espacio como se muestra en el siguiente ejemplo:

```
V01XXXXXXX Alejandro Mujica
```

**Por favor, no uses otros medios para enviar la solución ni escribas otros comentarios adicionales en el email**.

**Atención:** si tu programa no compila, entonces el evaluador no compila. Si una de tus rutinas se cae, entonces el evaluador se cae. Si una de tus rutinas cae en un lazo infinito o demora demasiado, entonces el evaluador cae en un lazo infinito o se demora demasiado. Por esa razón, en todos estos casos será imposible darte una nota, lo que simplemente se traduce en que tienes cero en el intento en el cual ocurra una de las circunstancias mencionadas.

No envíes programas que no compilan o se caen. Hacen perder los tiempos de red, de cpu, el tuyo y el mío. Si estás al tanto de que una rutina se te cae y no la puedes corregir, entonces trata de aislar la falla. Si no logras implementar una rutina, entonces haz que dé un valor de retorno el cual, aunque estará incorrecto y no se te evaluará, le permitirá al evaluador proseguir con otras rutinas. De este modo no tumbarás al evaluador y eventualmente éste podría darte nota para algunos casos (o todos si tienes suerte). Si no logras aislar una falla, entonces deja la rutina tal como te fue dada, pero asegúrate de que dé un valor de retorno. De este modo, otras rutinas podrán ser evaluadas.

**La primera indicación dada en este documento fue que lo leyeras completamente antes de trabajar o formular preguntas. Cualquier falta a las normas de evaluación aquí mencionadas, serán consideradas como un "no leyó el documento", por lo tanto, no habrá manera de que tu práctica sea evaluada.**

# 4. Recomendaciones

1. Haz un boceto de tu estructura de datos y cómo esperas utilizarla. Por cada rutina, plantea qué es lo que vas hacer y cómo vas a resolver el problema. Esta es una situación que amerita una estrategia de diseño y desarrollo.

2. La distribución de este trabajo contiene un pequeño test. No asumas que tu implementación es correcta por el hecho de pasar el test. Construye tus casos de prueba, verifica condiciones frontera y manejo de alta escala.

3. Este es un problema en el cual los refinamientos sucesivos son aconsejables. La recomendación general es que obtengas una correcta versión operativa lo más simplemente posible. Luego, si lo prefieres, puedes optar por mejorar el rendimiento.

4. Ten cuidado con el manejo de memoria. Asegúrate de no dejar "leaks" en caso de que utilices memoria dinámica. Todo `new` que hagas debe tener su `delete` en algún lugar. valgrind **y** DDD **son buenos amigos**.

5. Usa el grupo para plantear tus dudas de comprensión. Pero de ninguna manera compartas código, pues es considerado **plagio**. Tampoco hagas preguntas en privado, cualquier pregunta que tengas, es posible que otro también la tenga, si cada uno pregunta por separado, entonces tendré que dar varias veces la misma respuesta.

6. No incluyas headers en tu archivo `Huffman.H` (algo como `# include ...`), pues puedes hacer fallar la compilación. Si requieres un header especial, el cual piensas no estaría dentro del evaluador, exprésalo en el grupo para así incluirlo.