**INF-134 Estructuras de Datos, 2020-1**

# Tarea 2

Profesores: Diego Arroyuelo B., Roberto Díaz U.
darroyue@inf.utfsm.cl, roberto.diazu@usm.cl

Ayudantes:
Anastasiia Fedorova anastasiia.fedorova@sansano.usm.cl,
Gonzalo Fernandez gonzalo.fernandezc@sansano.usm.cl,
Tomás Guttman tomas.guttman@sansano.usm.cl,
Héctor Larrañaga hector.larranaga@sansano.usm.cl,
Martín Salinas martin.salinass@sansano.usm.cl.

Fecha de entrega: 10 de julio, 2020.
Plazo máximo de entrega: 3 días.

## 1. Reglas del Juego

La presente tarea debe hacerse en grupos de 3 personas. Toda excepción a esta regla debe ser conversada con los ayudantes (usando los correos indicados en el encabezado de esta tarea). No se permiten de ninguna manera grupos de más de 3 personas. Debe usarse el lenguaje de programación C. Alternativamente, pueden usar C++, aunque en ese caso no se pueden usar las librerías `stl`. En cualquiera de los dos casos, las estructuras de datos usadas en esta tarea deben ser implementadas por usted mismo. Al evaluarlas, las tareas serán compiladas usando el compilador `gcc` (o `g++` en el caso de C++), usando la línea de comando `gcc archivo.c -o output -Wall`. Alternativamente, se aceptan variantes o implementaciones particulares de `gcc`, como el usado por MinGW (que está asociado a la IDE code::blocks). Se deben seguir los tutoriales que están en Aula USM, cualquier alternativa explicada allí es válida. Recordar que una única tarea en el semestre puede tener nota menor a 30. El no cumplimiento de esta regla implica reprobar el curso.

## 2. Objetivos

Entender y familiarizarse con el uso de estructuras de datos lineales (listas, pilas y colas) y árboles binarios de búsqueda.

## 3. Problema 1: Administrador de Memoria Dinámica

Se desea implementar un administrador de memoria dinámica. Se tiene una memoria de $M$ bytes, inicialmente disponible como un único bloque de memoria contigua. A lo largo de la ejecución, existirán requerimientos de bloques de memoria contigua de un tamaño variable $m$ bytes (operación `malloc`), y además se liberarán bloques de memoria asignados anteriormente (operación `free`).

Se pide administrar los bloques de memoria usando listas enlazadas (también puede usar listas doblemente enlazadas). En particular, se necesitan dos listas:

- Lista de bloques disponibles ($L_1$): cada nodo de la lista almacenará un bloque de memoria disponible, representado por dos números enteros. Estos indican el byte de comienzo y el byte de final ocupados

por el bloque. Inicialmente esta lista contiene un único nodo, que representa el bloque de memoria $[1..M]$. Los nodos de la lista se mantienen ordenados por el byte de comienzo de los bloques libres.

- Lista de bloques asignados ($L_2$): esta lista se mantiene de manera similar a la anterior, salvo que no es necesario mantener el orden de los nodos. Inicialmente, la lista está vacía (no hay bloques asignados).

Dado un requerimiento de memoria de $m$ bytes, se debe recorrer la lista $L_1$ hasta encontrar el primer bloque de tamaño $m'$ capaz de satisfacer el requerimiento (estrategia conocida como "first-fit"). Es decir, $m \leq m'$. En caso de que $m < m'$, se debe agregar el bloque de tamaño $m$ a la lista $L_2$, y se debe modificar el tamaño del bloque disponible en la lista $L_1$ (el tamaño ahora es $m' - m$ bytes).

La liberación de memoria, por otro lado, se hará indicando el byte de comienzo del bloque a liberar. Dicho bloque debe ser buscado en la lista $L_2$, será eliminado de la misma, y asignado a la lista $L_1$ (en la posición adecuada). Se debe tener en cuenta que si existen bloques libres contiguos en $L_1$, los mismos deben ser unificados en un único gran bloque. Por ejemplo, supongamos que la lista $L_1$ contiene los bloques $[i..j]$ y $[k..l]$, y luego se libera el bloque $[j+1..l-1]$ de $L_2$. Entonces, los tres nodos contiguos de $L_1$ $[i..j]$, $[j+1..l-1]$ $[k..l]$ de la lista de bloques disponibles deben unificarse en un único bloque disponible $[i..l]$.

## 3.1. Entrada de Datos

La entrada de datos se hará mediante el archivo `input1.dat`, el cual tiene el siguiente formato:

```
M
N
OP1
...
OPN
```

En donde:

- `M` es la cantidad de bytes total de la memoria.
- `N` es la cantidad de operaciones que se harán sobre la memoria dinámica.
- `OPi` son las operaciones sobre la memoria dinámica, las cuales pueden ser de dos tipos:
  - `malloc m`: solicita la asignación de un bloque contiguo de `m` bytes.
  - `free b`: libera el bloque de memoria que comienza en el byte `b`. Se asume que los valores de `b` serán tales que el bloque correspondiente ya ha sido asignado.

Un posible ejemplo es el siguiente:

```
100
9
malloc 10
malloc 20
malloc 10
malloc 40
malloc 21
free 31
malloc 5
malloc 10
malloc 5
```

### 3.2. Salida de Datos

La salida de datos se hará mediante el archivo `output1.dat`. Por cada una de las operaciones del archivo `input2.dat`, este archivo tendrá una línea que indique alguna de las siguientes opciones:

- `Bloque de m bytes asignado a partir del byte B`: indica que una operación "`malloc m`" ha sido exitosa. El valor `B` indica el byte de comienzo del bloque asignado.
- `Bloque de m bytes NO puede ser asignado`: cuando una operación "`malloc m`" no puede ser satisfecha, dado que no hay un bloque disponible de tamaño suficiente.
- `Bloque de m bytes liberado`: indica que una operación "`free b`" liberó un bloque de $m$ bytes.

Al finalizar la ejecución, el programa debería indicar si quedaron bloques de memoria sin asignar, y su tamaño, mostrando alguno de los siguientes mensajes:

- En caso de que se haya liberado toda la memoria pedida, se debe imprimir el mensaje `Toda la memoria dinámica pedida fue liberada`
- En caso de que no se haya liberado toda la memoria pedida, si quedaron `N` bloques de memoria que totalizan `B` bytes sin liberar debe imprimir el mensaje `Quedaron N bloques sin liberar (B bytes)` (obviamente, reemplazando `N` y `B` por los valores enteros correspondientes).

De acuerdo a esto, la salida correspondiente al ejemplo visto anteriormente es:

```
Bloque de 10 bytes asignado a partir del byte 1
Bloque de 20 bytes asignado a partir del byte 11
Bloque de 10 bytes asignado a partir del byte 31
Bloque de 40 bytes asignado a partir del byte 41
Bloque de 21 bytes NO puede ser asignado
Bloque de 10 bytes liberado
Bloque de 5 bytes asignado a partir del byte 31
Bloque de 10 bytes asignado a partir del byte 81
Bloque de 5 bytes asignado a partir del byte 36
Quedaron 6 bloques sin liberar  (90 bytes)
```

## 4. Problema 2: Árboles Binarios de Búsqueda con Operación Sucesor

Sea $S = \{x_1, x_2, \ldots, x_n\}$ un conjunto de $n$ elementos enteros tal que $1 \leq x_1 < x_2 < \cdots < x_n < u$, para algún valor de $u$ específico. Luego, dado un elemento $x \in [1..u)$, de define la operación

$$\mathsf{sucesor}(S, x) = \text{mín}\,\{y \in S \cup \{u\} \mid x < y\}.$$

Como ejemplo, considere el conjunto $S = \{3, 6, 8, 9, 10, 11, 15, 19, 21\}$, para $u = 30$. Entonces:

- $\mathsf{sucesor}(S, 5) = 6$,
- $\mathsf{sucesor}(S, 10) = 11$,
- $\mathsf{sucesor}(S, 17) = 19$,
- $\mathsf{sucesor}(S, 20) = 21$,
- $\mathsf{sucesor}(S, 22) = 30$.

En la presente tarea, deberá implementar la función sucesor de forma eficiente, suponiendo que el conjunto $S$ ha sido almacenado en un árbol binario de búsqueda. Además, el árbol deberá soportar la inserción y eliminación de elementos, así como implementar algunos de los recorridos que hemos definido sobre árboles binarios de búsqueda.

## 4.1. Entrada de Datos

La entrada de datos será a través del archivo `input.txt`, con el siguiente formato. El archivo comienza con una línea que contiene un único número entero positivo `u` ($1 \leq$ `u` $\leq$ 10,000,000). Luego, sigue un número de líneas, cada línea conteniendo una operación a realizar sobre el conjunto de datos (que inicialmente está vacío). La operaciones válidas son las siguientes:

- `INSERTAR N`: agrega el entero `N` al conjunto, sólo si éste no pertenece al conjunto. En otro caso, no tiene efecto.
- `BORRAR N`: elimina el entero `N` del conjunto, sólo si éste pertenece al conjunto. En otro caso, no tiene efecto. En caso de que el elemento a borrar tenga dos hijos no nulos en el árbol, aplique la política que lo reemplaza por su predecesor.
- `PREORDEN`: imprime los elementos del conjunto, siguiendo un recorrido preorden del árbol.
- `SUCESOR X`: imprime el sucesor del número `X`.

El orden en que pueden aparecer esas operaciones dentro del archivo es arbitrario. El archivo finaliza con EOF.

Un ejemplo de entrada válida es:

```
30
SUCESOR 15
INSERTAR 10
INSERTAR 6
INSERTAR 19
SUCESOR 7
PREORDEN
INSERTAR 8
INSERTAR 11
PREORDEN
SUCESOR 8
SUCESOR 11
INSERTAR 9
INSERTAR 15
SUCESOR 8
SUCESOR 11
INSERTAR 21
BORRAR 19
INSERTAR 3
SUCESOR 5
SUCESOR 10
SUCESOR 17
SUCESOR 20
SUCESOR 22
PREORDEN
```

## 4.2. Salida de Datos

La salida de datos se realizará a través de la salida standard (pantalla). Se deben imprimir los resultados de las operaciones `SUCESOR` y `PREORDER` de acuerdo al orden en que aparecen en el archivo `input.txt`. En particular, por cada operación `SUCESOR` en el archivo `input.txt`, se debe imprimir una línea que contiene el resultado de la operación aplicada sobre el conjunto actual. Por cada operación `PREORDEN` en el archivo

`input.txt`, se debe imprimir una línea que contiene todos los elementos del conjunto en un recorrido preorden del árbol. Dentro de la línea, los elementos deben estar separados por un único espacio.

La salida correspondiente a la entrada de ejemplo de la sección anterior es:

```
30
10
10 6 19
10 6 8 19 11
10
19
9
15
6
11
21
21
30
10 6 3 8 9 15 11 21
```

# 5. Entrega de la Tarea

La entrega de la tarea debe realizarse enviando un archivo comprimido llamado

`tarea2-apellido1-apellido2-apellido3.tar.gz`

(reemplazando sus apellidos según corresponda) a la página aula.usm del curso, a más tardar el día 10 de julio, 2020, a las 23:59:00 hs (Chile Continental), el cual contenga:

- Los archivos con los códigos fuentes necesarios para el funcionamiento de la tarea. ¡Los archivos deben compilar!
- **nombres.txt**, Nombre, ROL, Paralelo y qué programó cada integrante del grupo.
- **README.txt**, Instrucciones de compilación en caso de ser necesarias, y la forma de compilación que usó (debe ser alguna de las indicadas en los tutoriales entregados en Aula USM).

# 6. Restricciones y Consideraciones

- Por cada día de atraso en la entrega de la tarea se descontarán 10 puntos en la nota.
- El plazo máximo de entrega es 3 días después de la fecha original de entrega.
- **Las tareas que no compilen no serán revisadas y serán calificadas con nota 0**.
- Debe usar **obligatoriamente** alguna de las formas de compilación indicada en los tutoriales entregados en Aula USM.
- Por cada *Warning* en la compilación se descontarán 5 puntos.
- Si se detecta **COPIA** la nota automáticamente sera 0 (CERO), para todos los grupos involucrados. El incidente será reportado al jefe de carrera.
- La prolijidad, orden y legibilidad del código fuente es obligatoria. Habrá descuentos si alguno de estos items no se cumple.

# 7. Consejos de Programación

El código fuente del programa debe estar estructurado adecuadamente en archivos (separados de ser necesario). Si el código fuente está desordenado, se pueden descontar hasta 20 puntos de la nota.

Cada función programada debe tener comentarios de la siguiente forma:

```
/*****
*   TipoFunción NombreFunción
******
*   Resumen Función
******
*   Input:
*       tipoParámetro NombreParámetro : Descripción Parámetro
*        .......
******
*   Returns:
*       TipoRetorno, Descripción retorno
*****/
```

**Por cada comentario faltante, se restarán 5 puntos.**

Por último, la indentación (1 TAB o 4 espacios), es muy importante. Por **cada bloque mal indentado, se quitarán 10 puntos**.