**TDA Memoria Heap**

Este programa se encarga de la simulación de la asignación y liberación de memoria en un proceso.

Las estructuras de datos usadas para la implementación del TDA son cuatro:

* Un bloque de memoria representado por un arreglo de char.
* Una lista enlazada con información sobre los bloques libres.
* Una lista enlazada con información sobre los bloques ocupados.
* Un mapa de bits que indica que bloques se encuentran libres y cuáles ocupados. Se representa con un arreglo de char, el cual posee la octava parte del tamaño de la memoria.

**int inicializarMemoria(int max)**

Crea e inicializa un espacio de memoria de *m* bytes, siendo *m* el menor múltiplo de 8 mayor o igual que max. Retorna 0 si la creación fue exitosa, 1 si no hay suficiente memoria, 2 si la memoria ya fue inicializada, 3 en caso de ocurrir cualquier otro error.

Además, esta función se encarga de inicializar las listas de bloques libres y ocupados y el mapa de bits de forma tal que toda la memoria este disponible.

* int max: tamaño con el que se desea inicializar la memoria.
* No tiene parámetros de salida.

**void\* asignar(int cant)**

Reserva cant bytes de memoria contigua y devuelve el puntero al primer byte. Retorna NULL en caso de no poder encontrar cant bytes contiguos. Actualiza las listas de bloques libres y ocupados, así como también el mapa de bits.

* int cant: el número de bytes de memoria que se desea reservar.
* El parámetro de salida es un puntero al primer byte del bloque de memoria si este existe, es NULL en caso contrario.

**void liberar(void\* bloque)**

Libera el bloque de memoria apuntado por bloque. Además actualiza las estructuras auxiliares. En el caso de que existan bloques contiguos de memoria libre, estos son fusionados para asegurar un mayor éxito en las futuras invocaciones a asignar.

* void\* bloque: puntero al primer byte del bloque de memoria a liberar.
* No tiene parámetros de salida.

**int consistente()**

Cheque la consistencia de la memoria. Retorna verdadero (<>0) si las listas de bloques y el mapa de bits representan la misma información, falso (0) en caso de detectar algún error.

* El parámetro de salida es un entero que representa la consistencia de la memoria, distinto de 1 si la memoria es consistente, 0 si existe algún error.

Funciones auxiliares

**void actualizarMapaBits(int posEnMemoria,int tam)**

Esta función se encarga de actualizar el mapa de bits a la hora de la asignación y liberación de bloques de memoria. Se calcula un número como potencia (2,posEnMemoria%8), el cual al hacer un EXOR con la celda del mapa de bits en la que se encuentra posEnMemoria (se calcula como posEnMemoria/8) hace un toogle en el bit correspondiente. Esta operación se hace un número de veces igual al tamaño del bloque. El parámetro posEnMemoria se va incrementando en uno en cada repetición de la operación.

* int posEnMemoria: representa la posición de inicio del bloque de memoria a actualizar en el mapa de bits.
* int tam: representa el tamaño del bloque de memoria.
* No tiene parámetros de salida.

Ej: void actualizarMapBits(0,3)

Se debe indicar que los primeros 3 bytes de la memoria están ocupados, asumiendo que la memoria esta libre y tiene 8 bytes.

* Mapa de bits antes de actualizarlo: 0000 0000
* Mapa de bits EXOR potencia(2,0)(1, en binario 1000 0000): 0000 0001
* Mapa de bits EXOR potencia(2,1)(2, en binario 0100 0000): 0000 0011
* Mapa de bits EXOR potencia(2,2)(4, en binario 0010 0000): 0000 0111

(En cada cálculo como se ve, posEnMemoria se incrementa en uno, por el número calculado cada vez es menor).

**void ubicar(tLista \*lista, tElemento x)**

Esta función se encarga de insertar ordenadamente un tElemento (bloque de memoria) en la lista pasada por parámetro. El valor que se tiene en cuenta para ordenar es el puntero al primer byte de cada bloque, es decir, la posición del comienzo del mismo. De esta manera, en la lista los bloques quedan ordenados de acuerdo a la posición que ocupan en la memoria.

* tLista\* lista: lista en la cual se debe insertar ordenadamente.
* tElemento x: elemento a insertar en la lista, representa un bloque de memoria.
* No tiene parámetros de salida.

**void fusión()**

La tarea de este método es fusionar bloques libres de memoria consecutivos. Decidimos hacer esta función como auxiliar de la operación liberar() para que el código quede más claro. Se podría haber hecho la fusión en el momento en que se inserta el bloque en la lista de bloques libres, pero esto traía algunas desventajas:

* el código quedaba engorroso, se debían hacer comparaciones con el bloque anterior y el siguiente para ver si eran consecutivos, y se complicaba entender cómo funcionaba el código.
* teníamos que definir una nueva operación ubicar() que se encargue además de insertar ordenadamente, fusionar los bloques consecutivos, por lo que no podíamos volver a reusar la operación ubicar() definida previamente.