Primer Parcial

Primer Cuatrimestre 2020

Normas generales

- El parcial es INDIVIDUAL, de modalidad remota tipo take-home.
- Durante el periodo que dure el take-home se dispondrá de la **lista de correo** para consultas. Las mismas serán respondidas prioritariamente durante el horario de clases. Además usaremos el canal de Anuncios en *Zulip* para darles información actualizada y aclaraciones sobre el examen. Les pedimos que lo revisen con regularidad.
- Una vez terminada la evaluación se deberá completar un formulario con el *hash* del *commit* del repositorio de entrega.
- **Tienen hasta las 23:59hs del 20/10 para completar el take-home.** Se solicita completar el formulario con tiempo a fin de evitar inconvenientes de conectividad.
- Si por algún motivo tienen problemas con *gitlab*, es fundamental que obtengan el hash del commit en su repositorio local y lo envien en el form de entrega. Esto certifica que realizaron la entrega en tiempo y forma. No será considerada ninguna entrega cuyo hash no haya sido enviado, o el hash no corresponda con un commit de su repositorio.
- El día 27/10 se enviará por mail los resultados parciales de la evaluación, adjuntando una lista de todos los alumnos que rendirán el coloquio complementario.
- En el caso que deban rendir el coloquio complementario, su evaluador asignado se contactará con ustedes uno por medio de *Zulip* y realizarán una llamada. Utilizando una aplicación de code-share online, el evaluador presentará la solución del alumno y realizará preguntas sobre la misma.
- Una vez terminada la llamada, el evaluador completará una planilla con las respuestas del alumno.
- Es condición necesaria para rendir el coloquio complementario, contar con conexión a internet y una computadora donde poder escribir código y compartirlo. Además, deben contar con un micrófono adecuado para comunicarse. **De no contar con cualquiera de estos recursos les pedimos que nos informen a la brevedad.**

Enunciado

Los siguientes ejercicios están basados en las estructuras de datos presentadas en el TP1 y complementan los ejercicios realizados en el TP2. Todos los ejercicios deben ser resueltos en lenguaje ensamblador, ya sea utilizando instrucciones de enteros para la primera parte, como instrucciones vectoriales para la segunda parte.

Programación en enteros

Ejercicio 1

Implementar la función `int docSimilar(document_t* a, document_t* b, void* bitmap)` que toma dos punteros a documentos y un puntero a un mapa de bits. El mapa de bits es un vector de bytes, donde cada bit representa uno de los campos del documento. Esto significa que al primer campo del documento le corresponde el bit menos significativo del primer byte; al segundo campo le corresponde el segundo bit menos significativo del primer byte; y por ejemplo, al noveno campo le corresponde el bit menos significativo del segundo byte. Si el bit correspondiente al campo en el documento es 0, entonces ese campo no será tenido en cuenta en la comparación. Si el bit es 1, entonces el campo será tenido en cuenta en la comparación.

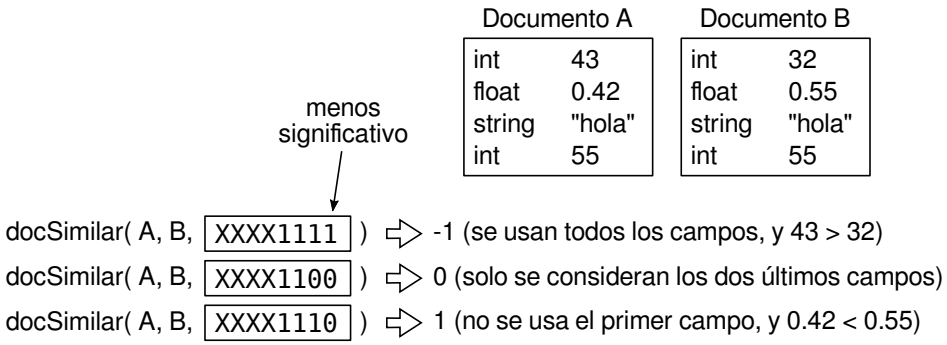
La comparación de documentos sigue las mismas reglas que `docCmp`:

1. Compara el tamaño de los documentos en cantidad de datos.
2. Si tienen la misma cantidad de datos. Compara los tipos de los datos uno a uno en orden de aparición en los documentos.
3. Si tienen los mismos tipos de datos. Compara los datos uno a uno en orden de aparición en los documentos, usando para esto la función de comparación asociada a cada tipo.
4. Si pasa todas estas comparaciones entonces los documentos son iguales.

El resultado de la función se debe reflejar según los siguientes casos.

$$\begin{array}{rl} 0 & \text{si son iguales} \\ 1 & \text{si } \texttt{a} < \texttt{b} \\ -1 & \text{si } \texttt{b} < \texttt{a} \end{array}$$

Ejemplo:



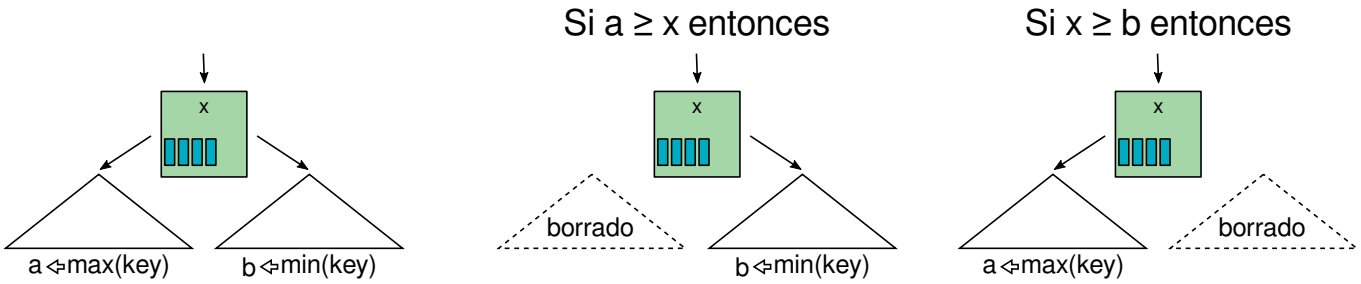
Nota: Suponer por simplicidad que los documentos tienen como máximo 8 campos. Es decir, que solo utilizan un byte en el mapa de bits.

Ejercicio 2

Implementar la función `int treeTrim(tree_t* tree, list_t** key)` que recorre el arbol pasado por parámetro y si existe un subarbol tal que no cumpla el invariante de ser arbol de busqueda, entonces borra el subarbol completo desde el nodo encontrado y agrega todos los *key* del subarbol a una lista. La lista debe ser retornada sobre el doble puntero pasado por parámetro. Los *values* dentro de los nodos del arbol deben ser borrados. Además, la función debe retornar `1` si algún nodo fue borrado y `0` en caso contrario. La cantidad de nodos totales en el arbol también debe ser actualizada para reflejar los nodos que fueron borrados.

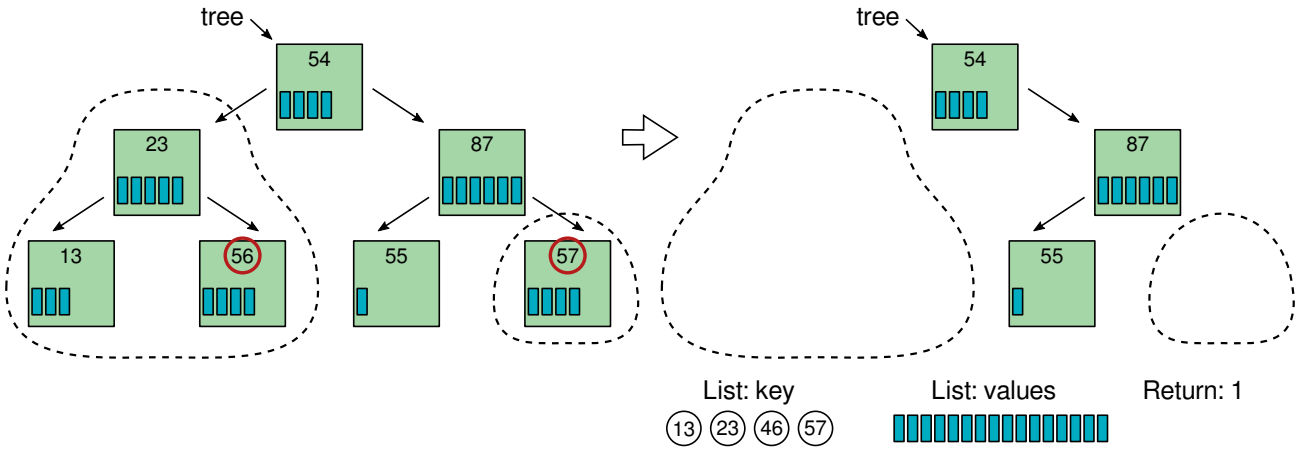
La función se debe encargar de crear la lista en cualquier caso, ya sea que deba retornar algún valor o no. Los datos en la lista a retornar pueden ser los datos originales o copias de los mismos.

Para determinar si un subarbol debe ser borrado, se debe buscar el máximo *key* del lado izquierdo (`a`) y el minimo *key* del lado derecho (`b`). Si se cumple la condición descripta en la figura, cada subarbol es borrado.



Nota: El arbol no contendrá *key*s repetidos.

Ejemplo:



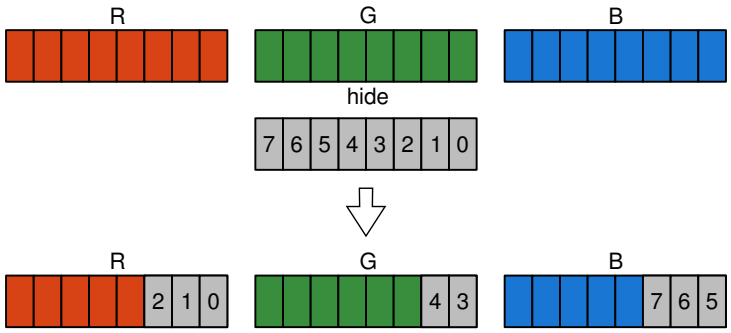
Programación vectorial

Ejercicio 3

Considerar una imagen de `n`×`m` pixeles de 32 bits almacenados en memoria en el orden B (blue), G (green), R (red), A (alpha), y una segunda imagen del mismo tamaño pero de 8 bits en escala de grises.

Se pide implementar la función `hide`. La misma guarda cada pixel de la imagen en escala en grises, en la parte menos significativa de las 3 componentes de color la imagen de 32 bits. Los 8 bits de la imagen en escala de grises (`T`) se almacenan de la siguiente forma:

- `R[2:0]` ← T[2:0]
- `G[1:0]` ← T[4:3]
- `R[2:0]` ← T[7:5]

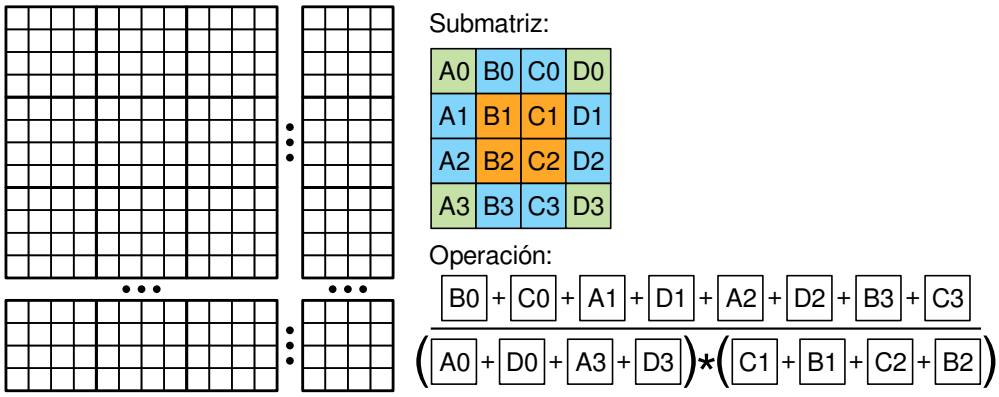


Donde los rangos indicados corresponden a bits de cada uno de los componente. `R`, `G` y `B`, corresponden a las componentes de color de la imagen de 32 bits.

La tarea de `hide` se realiza para toda la imagen, pixel a pixel. La solución debe procesar al menos 2 pixeles simultaneamente. Suponer además que `n` y `m` son valores múltiplo de 8 y que la aridad de la función es:
`void hide(uint32_t* imagen, uint8_t* T, int m, int n)`.
La operación debe modificar la imagen original pasada como parámetro.

Ejercicio 4

Considerar una matriz de `n`×`n` con `n` múltiplo de 8, de números en punto flotante de 32. Dividir la matriz en submatrices de 4×4 y para cada una realizar la siguente operación:



El resultado debe ser almacenado en punto flotante de 64 bits, en una nueva matriz de `n/4`×`n/4` que debe ser creada por la función. La aridad de la función es: `double* addings(float* matriz, int n)`.

Criterio de Aprobación

El parcial debe ser completado en su totalidad. Para todos los ejercicios se debe presentar una solución. Los ejercicios de codificar funciones deben poder ser ejecutados y resolver el problema planteado correctamente. Para aprobar se puede tener hasta un ejercicio resuelto de forma incorrecta.