El formato BCD o *Binary Coded Decimal* se usa para representar números enteros positivos en base 10. Cada cifra decimal del número se codifica en binario en 4 bits. Por ejemplo el número 60219 se codifica como 0b0110 0000 0010 0001 1001, en donde el 0110 codifica el 6, el 0000 el 0, el 0010 el 2, el 0001 el 1 y el 1001 el 9. La ventaja es que al anotar la secuencia de bits en hexadecimal se lee exactamente el mismo número representado en BCD, es decir 0x60219. Observe que las secuencias 0b1010 … 0b1111 o 0xa … 0xf no pueden ocurrir y que 0x60219 representa en BCD el número 60219 mientras que el mismo 0x60219 representa en binario el número 393753 ($9 + 1*16+2*16^2+6*16^4$).

Programe la función *sumaBcd* que retorna la suma de 2 números en formato BCD, almacenados en enteros sin signo de 64 bits, es decir *unsigned long long*. El número más grande representable puede tener hasta 16 cifras decimales. Si el resultado de la suma tiene 17 cifras decimales se produce desborde y Ud. debe retornar 0xffffffffffffffff. El encabezado de la función *sumaBcd* es:

```
typedef unsigned long long Bcd;
Bcd sumaBcd(Bcd x, Bcd y);
```

Ejemplo de uso:

```
Bcd a= sumaBcd(0x60219, 0x1); // a es 0x60220
Bcd b= sumaBcd(0x199305, 0x9781); // b es 0x209086
Bcd c= sumaBcd(0x9999999999999999, 0x1); // c es 0xfff...ffff
```

Observe que no sirve sumar directamente los números en BCD con el operador + de C porque 0x60219 + 0x1 es 0x6021A, que es una secuencia de bits inválida en BCD. Por lo tanto Ud. necesita separar las cifras decimales de a 4 bits y sumar exactamente como aprendió a hacerlo en enseñanza básica.

Restricciones:

- Ud. no puede usar los operadores de multiplicación, división o módulo (* / %). Use los operadores de bits eficientemente.
- No se permite convertir los números a binario, sumarlos con + y convertir el resultado a BCD.
- Se descontará medio punto por no usar el estilo de indentación de Kernighan como se explica en esta sección de los apuntes.
- El estándar de C no especifica el resultado para desplazamientos mayores o iguales al tamaño del operando. Sanitize rechaza el desplazamiento *x<<nbits* cuando *nbits* es mayor o superior a la cantidad de bits de *x*.

Instrucciones

Baje *t1.zip* de U-cursos y descomprímalo. El directorio *T1* contiene los archivos (a) *test-suma.c* que prueba si su tarea funciona y compara su eficiencia con la solución del profesor, (b) *prof.ref-x86_64 y prof.ref-aarch64* con los binarios ejecutables de la solución del profesor, (c) *suma.h* que incluye el encabezado de la función pedida, y (d) *Makefile* que le servirá para compilar y ejecutar su tarea. Ud. debe programar la función *sumaBcd* en el archivo *suma.c*.

Pruebe su tarea bajo Debian 11 de 64 bits nativo o virtualizado con VirtualBox, Vmware, QEmu o WSL 2. **Ejecute el comando *make* sin parámetros**. Le mostrará las opciones que tiene para compilar su tarea. Estos son los requerimientos para aprobar su tarea:

- *make run* debe felicitarlo por aprobar este modo de ejecución. Su solución no debe ser 80% más lenta que la solución del profesor.
- *make run-g* debe felicitarlo.
- *make run-san* debe felicitarlo y no reportar ningún problema como por ejemplo desplazamientos indefinidos.

Cuando pruebe su tarea con *make run* asegúrese que su computador esté configurado en modo alto rendimiento y que no estén corriendo otros procesos intensivos en uso de CPU al mismo tiempo. De otro modo podría no lograr la eficiencia solicitada.

Entrega

Ud. solo debe entregar por medio de U-cursos el archivo *suma.zip* generado por el comando *make zip*. **A continuación es muy importante que descargue de U-cursos el mismo archivo que subió, luego descargue nuevamente los archivos adjuntos y vuelva a probar la tarea tal cual como la entregó**. Esto es para evitar que Ud. reciba un 1.0 en su tarea porque entregó los archivos equivocados. Créame, sucede a menudo por ahorrarse esta verificación. Se descontará medio punto por día de atraso. No se consideran los días de receso, sábados, domingos o festivos.

Parte a.- Programe la función:

```
void eliminar(char *str, char *pat);
```

Esta función elimina del string *str* todas las apariciones del patrón *pat*. Ejemplos:

```
char s1[]= "las palas van"; // 13 caracteres
eliminar(s1, "las"); // s1 es " pa van" (7 caracteres)
char s2[]= "11101101";
eliminar(s2, "10"); // s2 es "11101"
```

Restricciones: Ud. no puede usar el operador de subindicación [ ], ni su equivalente **(p+i)*. Use ++ -- *p+i* o *p-i*. Por razones de eficiencia, Ud. no puede usar *malloc* o declarar un arreglo para pedir memoria adicional. Use múltiples punteros para direccionar distintas partes del string.

Parte b.- Programe la función:

```
char *eliminados(char *str, char *pat);
```

Esta función entrega un nuevo string en donde se han eliminado del string *str* todas las apariciones del patrón *pat*. Ejemplos:

```
char *s1= eliminados("las palas van", "las");
          // s1 es " pa van" (7 caracteres)
char *s2= eliminados("11101101", "10");
          // s2 es "11101"
```

Restricciones: Ud. no puede usar el operador de subindicación [ ], ni su equivalente **(p+i)*. Use ++ o *p+i*. Para recorrer el string use aritmética de punteros. Use *malloc* para pedir memoria para el string resultante. Debe pedir exactamente la cantidad de bytes que necesita el resultado, no más. Para el primer ejemplo debe pedir 8 bytes, para el segundo 6 bytes. Si pide más memoria que la que necesita, el test de uso de memoria podría agotar la memoria de su computador haciendo que se ponga muy lento antes de que el programa falle.

Instrucciones

Baje *t2.zip* de U-cursos y descomprímalo. El directorio *T2* contiene los archivos (a) *test-elim.c* que prueba si su tarea funciona y compara su eficiencia con la solución del profesor, (b) *prof.ref-x86_64 y prof.ref-aarch64* con los binarios ejecutables de la solución del profesor, (c) *elim.h* que incluye los encabezados de las funcion pedidas, y (d) *Makefile* que le servirá para compilar y ejecutar su tarea. **Ejecute en un terminal el comando *make*** para recibir instrucciones adicionales. Estos son los requerimientos para aprobar su tarea.

- *make run* debe felicitarlo por aprobar este modo de ejecución. Su solución no debe ser 80% más lenta que la solución del profesor.
- *make run-g* debe felicitarlo.
- *make run-san* debe felicitarlo y no reportar ningún problema como por ejemplo goteras de memoria.

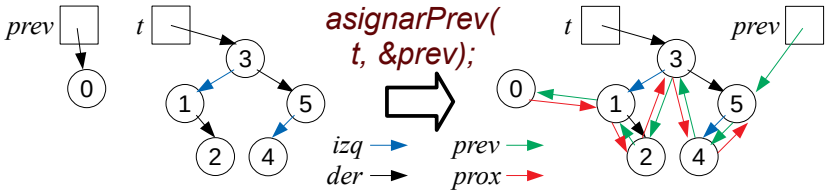
Cuando pruebe su tarea con *make run* asegúrese que su computador esté configurado en modo alto rendimiento y que no estén corriendo otros procesos intensivos en uso de CPU al mismo tiempo. De otro modo podría no lograr la eficiencia solicitada.

Entrega

Ud. solo debe entregar por medio de U-cursos el archivo *elim.zip* generado por el comando *make zip*. **A continuación es muy importante que descargue de U-cursos el mismo archivo que subió, luego descargue nuevamente los archivos adjuntos y vuelva a probar la tarea tal cual como la entregó**. Esto es para evitar que Ud. reciba un 1.0 en su tarea porque entregó los archivos equivocados. Créame, sucede a menudo por ahorrarse esta verificación. Se descontará medio punto por día de atraso. No se consideran los días de receso, sábados, domingos o festivos.

En un *recorrido en orden* de un árbol binario, se visita recursivamente primero el subárbol izquierdo, luego se visita la raíz y finalmente se visita recursivamente el subárbol derecho. Considere que se está visitando un nodo T al recorrer un árbol binario en orden. Se define como *previo* a T el nodo que se visitó anteriormente, y como *próximo* el nodo que se visitará a continuación. Estudie el lado derecho de la figura de ejemplo. Programe en el archivo *prev.c* la función *asignarPrev* que asigna los campos *prev* y *prox* agregados a la estructura de los nodos de un árbol *t*. El encabezado de la función se muestra a la derecha. El parámetro **pprev* es de entrada y salida. El nodo previo del primer nodo visitado (el nodo 1 en el ejemplo) debe ser el nodo apuntado inicialmente por **pprev* (nodo 0) y el nodo próximo del último nodo en ser visitado (nodo 5) debe ser NULL. En **pprev* debe quedar finalmente la dirección del último nodo visitado (nodo 5). En el siguiente ejemplo de uso las variables *t* y *prev* son de tipo Nodo *.

```
typedef struct nodo {
  int x;
  struct nodo *izq, *der;
  struct nodo *prev, *prox;
} Nodo;
void asignarPrev(Nodo t,
                 Nodo **pprev);
```



Restricción: Su solución debe tomar tiempo linealmente proporcional al número de nodos en el árbol *t*.

Ayuda: Cuando visite el nodo T, su nodo previo es **pprev*. Asigne NULL a su nodo próximo por ahora. Si el nodo previo a T no es NULL, T es el nodo próximo del nodo previo a T. Antes de continuar el recorrido, asigne T a **pprev*.

Instrucciones

Descargue *t3.zip* de U-cursos y descomprímalo. El directorio *T3* contiene los archivos (a) *test-prev.c* que prueba si su tarea funciona y compara su eficiencia con la solución del profesor, (b) *prof.ref-x86_64 y prof.ref-aarch64* con los binarios ejecutables de la solución del profesor, (c) *prev.h* que incluye los encabezados de las funcion pedidas, (d) *Makefile* que le servirá para compilar y ejecutar su tarea, y (e) *prev.cbf* para que pueda probar su tarea con *codeblocks*. **Ejecute en un terminal el comando *make*** para recibir instrucciones adicionales. Estos son los requerimientos para aprobar su tarea.

- *make run* debe felicitarlo por aprobar este modo de ejecución. Su solución no debe ser 80% más lenta que la solución del profesor.
- *make run-g* debe felicitarlo.
- *make run-san* debe felicitarlo y no reportar ningún problema como por ejemplo *heap-buffer-overflow*.

Cuando pruebe su tarea con *make run* asegúrese que su computador esté configurado en modo alto rendimiento y que no estén corriendo otros procesos intensivos en uso de CPU al mismo tiempo. De otro modo podría no lograr la eficiencia solicitada.

Entrega

Ud. solo debe entregar por medio de U-cursos el archivo *prev.zip* generado por el comando *make zip*. **A continuación es muy importante que descargue de U-cursos el mismo archivo que subió, luego descargue nuevamente los archivos adjuntos y vuelva a probar la tarea tal cual como la entregó**. Esto es para evitar que Ud. reciba un 1.0 en su tarea porque entregó los archivos equivocados. Créame, sucede a menudo por ahorrarse esta verificación. Se descontará medio punto por día de atraso. No se consideran los días de receso, sábados, domingos o festivos.

El archivo de texto *dicc.txt* que viene a continuación almacena un diccionario representado como una tabla de hashing cerrado.

```
celular:aparato portatil de un sistema de telefonia celular

casa:edificacion construida para ser habitada

posada:establecimiento economico de hospedaje para viajeros
alimento:sustancia ingerida por un ser vivo
```

Todas las líneas son de 80 caracteres más el fin de línea (caracter '\n'). Se usa el caracter ':' para separar la llave de su valor asociado. Una línea que contiene solo espacios en blanco no contiene ninguna definición.

Escriba en el archivo *definir.c* un programa que agregue una llave al diccionario. El siguiente es un ejemplo de uso:

```
$ make definir.bin
$ ./definir.bin dicc.txt bolsillo "bolsa pequena"
```

En donde el primer parámetro es el nombre del archivo que contiene el diccionario, el segundo es la llave definida y el tercero es la definición. En el archivo *definir.c*, Ud. deberá programar la función:

*int main(int argc, char *argv[ ]);*

Requerimientos

- Abra el archivo del diccionario en modo "r+" para poder leerlo y escribirlo. Es decir: *fopen(..., "r+")*.
- Calcule el tamaño y el número de líneas en el archivo usando *fseek y ftell*.
- **Debe ser eficiente.** Use *hash_string(llave)* módulo número de líneas del archivo para calcular el número de línea en donde debe ir la llave definida. Luego use fseek y fread para leer esa línea. Si esa línea está desocupada reescriba esa misma línea para agregar la definición en esa misma línea. Si no, lea las líneas que vienen a continuación hasta encontrar una línea en blanco y reescríbala para agregar la defición en esa línea. El archivo es circular: si llega al final, continúe desde el comienzo.
- Está prohibido leer el archivo completo, salvo si el archivo está lleno.

Debe diagnosticar las situaciones de error en la salida estándar de errores. En caso de error termine el programa de inmediato. Estos son los errores que debe diagnosticar:

- No se puede abrir el archivo. Debe diagnosticar este error con *perror*.
- La llave ya estaba definida en el diccionario.

El test de prueba incluido verifica que Ud. diagnostique estos errores con exactamente el mismo mensaje que entrega la solución de referencia incluida en formato binario. Puede probar la solución de referencia con un diccionario y llave específica. Por ejemplo:

```
$ ./prof.ref-$(arch) dicc.txt bolsillo "bolsa pequena"
```

Vea el diccionario resultante con el comando: `less dicc.txt`

Cada ejecución de *definir* modifica el diccionario. Para restaurarlo ejecute el comando: `bash mk-dicc.sh`

Instrucciones

Descargue *t4.zip* de U-cursos y descomprímalo. Ejecute el comando *make* sin parámetros en el directorio *T4* para recibir instrucciones acerca del archivo en donde debe programar su solución (*T4/definir.c*), cómo compilar y probar su solución, los requisitos que debe cumplir para aprobar la tarea (*make run-san*, *make run-g* y *make run*) y cómo entregar su tarea por U-cursos (*make zip*).

Entrega

Ud. solo debe entregar por medio de U-cursos el archivo *definir.zip* generado por el comando *make zip*. **A continuación es muy importante que descargue de U-cursos el mismo archivo que subió, luego descargue nuevamente los archivos adjuntos y vuelva a probar la tarea tal cual como la entregó**. Esto es para evitar que Ud. reciba un 1.0 en su tarea porque entregó los archivos equivocados. Créame, sucede a menudo por ahorrarse esta verificación. Se descontará medio punto por día de atraso. No se consideran los días de receso, sábados, domingos o festivos.