Sistemas Operativos 2020-2, Práctica 02: Apuntadores y Estructuras en C.

Luis Enrique Serrano Gutiérrez (luis@ciencias.unam.mx)
Juan Alberto Camacho Bolaños (juancamacho@ciencias.unam.mx)
Ricchy Alaín Pérez Chevanier (alain.chevanier@ciencias.unam.mx)

Fecha de Entrega: 02 de Marzo de 2020

1 Objetivo

El objetivo de esta práctica es que el estudiante ejercite el uso de estructuras y apuntadores en C, pues posteriormente será necesario que éste maneje con fluidez dichas entidades para poder resolver problemas dentro de Pintos.

2 Introducción

2.1 Docker

Para esta práctica es necesario que tengas instalada la instancia de docker de la práctica 01, de nuevo proveemos un script con atajos para compilar y ejecutar dentro del contenedor de docker.

A continuación una lista de todos los comandos disponibles en el script `docker-exec`:

- `./docker-exec compile` compila todos los archivos `.c` por medio del comando `make`
- `./docker-exec run` una vez que todo compila sin errores puedes ejecutar el binario generado con este comando, que en el fondo ejecuta el comando `make check`.
- `./docker-exec clean` elemina los archivos objeto y ejecutable por medio del comando `make clean`
- `./docker-exec "any other command"` ejecuta la instrucción *any other command* dentro del contenedor de docker, por ejemplo puedes ejecutar `./docker-exec "make"` compila el código.

2.2 Descripción del código

Para todos los problemas propuestos proporcionamos infraestructura de código y pruebas unitarias para poder implementar y probar tus soluciones. Dicha infraestructura se encuenta dentro del directorio ***src***. A continuación damos una breve explicación de algunos archivos de relevancia contenidos en dicho directorio:

- **list_node.h**: Este archivo contiene la definición de la estructura que representa a un nodo de una lista simplemente ligada, así como también las firmas de algunas funciones auxiliares que actúan sobre listas.
- **problemas.h**: Este archivo contiene las firmas de las funciones que deberás implementar para poder resolver los problemas, estas firmas son inmutables pues las pruebas unitarias las suponen.

- **soluciones.c**: Este archivo debe de contener las implementaciones de tus soluciones a los para los problemas, por default tiene el esqueleto de las funciones que debes de implementar.
- **main.c**: Este archivo como su nombre sugiere contiene el método main, que ejecuta todas las pruebas unitarias de las sobre tus soluciones para los problemas.

2.3 Salida de las pruebas

Cuando ejecutes `./docker-exec run` y suponiendo que pasas todas las pruebas, deberás obtener una salida como la siguiente:

```
STARTING UNIT TEST:

TESTS FOR LIST_CREATE_COPY:
Empty List: PASS
Single node: PASS
Huge list: PASS
counter example: PASS

TESTS FOR HEAP SORT:
Single element array: PASS
Complete Tree: PASS
Incomplete Tree: PASS
Worst case runtime: PASS
Linear Execution: PASS
With Repetitions: PASS
Huge array: PASS

TESTS FOR INSERT_NODE FUNCTION:
Empty list, empty new_node: PASS
Empty list, single new_node: PASS
Empty list, multi-node list: PASS
List, empty list: PASS
List, single-node list, result contains new_node: PASS
List, single-node list: PASS
List, multi-node list: PASS
List, single-node list, first: PASS
List, multi-node list, first: PASS
List, single-node list, last: PASS
List, multi-node list, last: PASS

TESTS FOR EGYPCIAN MULTIPLICATION:
Multiplication by zero left: PASS
Multiplication by zero rigth: PASS
Mutiplication functional test case: PASS
Mutiplication robustness test case: PASS

TESTS FOR HAS_CYCLE FUNCTION:
Empty list: PASS
Single node, no cycle: PASS
Single node, cycle: PASS
Huge list, no  cycle: PASS
Huge list, cycle: PASS

TESTS FOR ANAGRAMS FUNCTION:
Null Strings: PASS
```

```
Empty Strings: PASS
Empty Strings, String: PASS
String, Empty Strings: PASS
No anagrams: PASS Anagrams: PASS
Big anagrams: PASS
Big strings, no anagrams: PASS
Same weigth, different strings:
PASS Same characters, different strings: PASS

TESTS FOR CALC_MAX_SUM:
Functional test: PASS
Negative sum: PASS
All Positive sum: PASS
No elements sum: PASS
Null pointer sum: PASS

Overall test [ PASS / TOTAL ]: [ 46 / 46 ]
```

3 Ejercicios

3.1 Programación

3.1.1 Problema 1, Copiar Listas:

Consideremos la siguiente estructura que define un nodo de una lista simplemente ligada:

```
struct list_node {
        int val;
        struct list_node* next;
        struct list_node* rand;
};
```

Como es usual el apuntador next apunta al siguiente elemento de la lista, sin embargo el apuntador rand apunta a cualquier elemento de la lista elegido aleatoriamente, el apuntador rand no es nulo para ningún nodo.

Problema: El objetivo es crear un función que genere una copia de una lista definida por una secuencia de nodos formada por representantes de la estructura struct list_node, en donde se preserve la relación que existe entre cada nodo y el apuntador rand que este tiene. Por ejemplo si en la lista original el nodo 5 está apuntando por medio de rand al nodo 2, entonces se debe de cumplir esa misma relación en la copia creada.

Hint: Combina la lista original con la copia generada de tal forma que en una pasada por la nueva lista combinada puedas generar las relaciones para los apuntadores rand de la lista copia y luego seperar cada lista.

Nota: Existe una solución para el problema que utiliza tiempo $O(n)$ y que utiliza memoria constante adicional a la ocupada por la replica de los nodos de la lista original.

3.1.2 Problema 2, Heap Sort:

Utilizar algoritmos de ordenación y la estructura de datos heap es de gran relevancia para este curso, por lo que esperamos que por medio de este ejercicio los conozcan.

Problema: Implementar el algoritmo de ordenación Heap Sort.

Hint: Puedes utilizar un arreglo de enteros para representar el heap y con un poco de aritmética puedes manejarlo como si este fuera un árbol binario. Más aún puedes utilizar el mismo arreglo que

recibes como heap y el heap en cuestión sea del tipo max heap.[1]

Pregunta: Este algoritmo es inestable[2]. Supón que utilizas un max-heap para representar una priority queue, dicha queue representa la lista de espera de clientes para acceder a un spa, cada cliente compra un boleto para acceder al spa y entre más caro es el boleto mayor prioridad tienen el cliente en la lista de espera. ¿crees que la inestabilidad del algoritmo pueda eventualmente traer algún comportamiento no aceptable para algún cliente?. Contesta brevemente esta pregunta como parte de tu reporte.

Nota: Este algoritmo tiene tiempo de ejecución $O(n\,log\,(n))$.

3.1.3 Problema 3, Inserción de elementos en una Lista Simplemente Ligada:

Supongamos que L es una lista simplemente ligada definida como en el ejercicio 1, pero para la cual el apuntador `rand` es nulo para cada nodo, y supongamos que sus elementos están ordenados ascendentemente.

Problema: Implementa una función que inserte un nodo N en la lista L tal que L con el nuevo nodo siga estando ordenada.

Hint: El problema tiene una solución trivial, pero no pierdas de vista nunca los detalles, sobre todo el hecho de que vas a insertar un nodo y no un entero.

Nota: Si no estás seguro por qué tu implementación no pasa alguna de las pruebas pregunta un poco más por medio del clasroom de google al resto de los compañeros. Dado que L ya está ordenado tu algoritmo no debería ser más lento que $O(n)$.

3.1.4 Problema 4, Multiplicación Egipcia:

La Multiplicación Antigua Egipcia es un método sistemático para multiplicar dos números que no requiere una tabla de multiplicación, sólo la habilidad de multiplicar por dos y de sumar. Este método descompone unos de los multiplicandos en una suma de potencias de dos y crea una tabla de duplicaciones del segundo multiplicando.

Este método tiene tres fases: descomposition, tabla y resultado.

La descomposición de un número N consiste en encontrar las potencias de dos que lo conforman. Los egipcios sabían empíricamente que cualquier potencia de dos sólo aparence una vez en la descomposición de potencias de dos de cualquier número.

Ejemplo considera el número $N = 13$, la descomposición de N es 8, 4, 1, pues $N = 8 + 4 + 1$.

Después de descomponer el primer multiplicando N, es necesario construir una tabla de potencias de dos multiplicando al segundo.

Por ejemplo la tabla generada por la multiplicación $N\,x\,M$ con $N = 13$ y $M = 238$ es la siguiente:

Potencia de Dos	M x Potencia de Dos
1*	238
2	476
4*	952
8*	1904
N = 13	N x M = 3094

El resultado final es obtenido al sumar los números de la columna de la derecha para los cuales la potencia de dos de la columna izquierda forma parte de la descomposición de N (en nuestro caso denotados por una marca *), i.e. $13 \times 238 = 238 + 952 + 1904 = 3094$.

Problema: Realizar un programa en C que simule la tabla generada por el método de la multiplicación Egipcia, en particular debe de regresar un arreglo que contenga la descomposición de la

[1] En wikipedia puedes encontrar de manera concisa la descripción del algoritmo. Aunque será visto durante las sesiones de laboratorio. http://en.wikipedia.org/wiki/Binary_heap

[2] Averigua qué significa inestable si desconoces el término

tabla (como la de arriba) pero sin las potencias de dos que no participan en las descomposición de N. Considera que N y M son mayores o iguales que cero pero menores o iguales que 1,000,000,000.

Para el ejemplo descrito debes de regresar el siguiente arreglo

[5, 238, 952, 1904, 3094]

Nota que siempre la primer entrada representa la longitud del arreglo y que la última debe de contener el resultado de la multiplicación, las pruebas unitarias utilizan ese valor es para poder saber donde empieza y termina el arreglo. Si cualquiera de los dos multiplicandos es cero el resultado debe de ser en vez de un arreglo una referencia nula.

Hint: Puedes basarte en la multiplicación bitwise que hicimos durante las sesiones de laboratorio.

Nota: No puedes utilizar el operador de multiplicación en ningún paso de tu solución, quizás podrías resolver este problema de forma muy eficiente utilizando sólo operaciones bitwise. Asegurate de que los tipos de datos que utilices tengan espacio suficiente para poder mantener el resultado de la multiplicación. Puedes procesar primero la descomposición de N para saber cuál será el tamaño del arreglo que tiene que devolver tu solución.

3.1.5 Problema 5, Ciclos en Listas Simplemente Ligadas: (Opcional 1 Pto Extra)

Sea L una lista simplemente ligadad definida por la estructura del ejercicio 1, en donde el apuntador `rand` no es necesariamente nulo para cada nodo. Decimos que L tiene un ciclo si y sólo si exite un nodo x de la lista que en cualquier recorrido secuancial aparece más de una vez. En general el problema es que el último nodo en vez a apuntar a NULL apunta a un nodo dentro de la lista, dicho lo anterior L no tiene último elemento en realidad.

Problema: Hacer un programa que tome a L y que decida si tiene un ciclo.

Hint: Evidentemente para saber si L tiene un ciclo es necesario recorrer L; la pregunta es ¿cuántas referencias a nodos de L utilizarías para recorrer L y responder el problema en cuestión, avanzarían a la misma velocidad?

Nota: Existe un algoritmo que en tiempo $O(n)$ decide esta propiedad. Muchos problemas dentro de pintos pueden ser fácilmente resueltos utilizando listas simplemente ligadas y algoritmos sobre ellas. Recibirás medio punto adicional si encuentras la solución lineal y es correcta.

3.1.6 Problema 6, Anagramas: (Opcional 1 Pto Extra)

Decimos que un enunciado es un anagrama de otro, si tiene las mismas letras pero en distinto orden. Sean s_1 y s_2 dos cadenas de caracteres que contiene únicamente caractéres del código ascii.

Problema: Realizar un programa que decida si s_1 y s_2 son anagramas.

Hint: La solución a este problema es muy sencilla, no pierdas de vista el hecho de que s_1 y s_2 sólo contienen caracteres del código ascii.

Nota: Existe una solución para el problema que utiliza tiempo $\mathcal{O}(n)$. Recibirás medio punto adicional si encuentras la solución lineal y es correcta.

3.1.7 Problema 7, Suma máxima de un arreglo: (Opcional 1 Pto Extra)

Dado un arreglo de números, queremos encontrar una subsecuencia del mismo tal que su suma sea máxima, por ejemplo si un arreglo contiene solamente números positivos entonces la subsecuencia de su suma máxima es el arreglo mismo. Por ejemplo para el arreglo $[-2, 1, -3, 4, -1, 2, 1, -5, 4]$ la sub-

secuencia de suma máxima es $[4, -1, 2, 1]$ con suma igual a 6.

Problema: Realizar un programa que dado un arreglo de enteros encuentre la subsecuencia de suma máxima, valor de returno debe de ser precisamente el valor de la suma.

Hint: Utiliza programación dinámica para resolver el problema.

Nota: Existe una solución para el problema que utiliza tiempo $O(n)$. Recibirás medio punto adicional si encuentras la solución lineal y es correcta.

4 Evaluación

- Dado que el objetivo de esta práctica es que resuelvas los primeros cuatro problemas, entonces sólo puedes recibirás puntos por los ejercicios adicionales si tienes una solución no trivial para estos.
- Como podrás ver cada una de tus soluciones será puesta a prueba automáticamente por una serie de pruebas; pero ésto no significa que no vamos a revisar tu código, lo haremos y parte de tu calificación estará relacionada con cómo realizaste tu implementación, por ejemplo si las pruebas del problema 3 pasan y tu solución no es suficientemente general entonces no tendrá puntos en este ejercicio, i.e. no es válido hacer soluciones que sólo resuelvan los casos que estresan las pruebas unitarias.
- Puedes modificar las pruebas del archivo `main.c`, de hecho es recomendado si quieres ir dado solución a los problemas en un orden distinto al de las pruebas, pero cuando nosotros califiquemos sustituiremos este archivo por una copia del original; entonces asegurate de que tu implementación pase las pruebas como fueron dadas originalmente. Si hay alguna prueba que no pases y que impida la ejecución completa de las pruebas por favor indica cuál es en tu reporte y nosotros la comentamos para que eso no impida que te evaluemos correctamente.