Práctica 2

Formato y fecha de entrega

Hay que entregar la Práctica antes del día **22 de mayo a las 23:59**. Tendréis que entregar un fichero en formato **ZIP de nombre logincampus_pr2** en minúsculas (donde *logincampus* es el nombre de usuario con el que hacéis login en el Campus), que contenga:

- Workspace CodeLite entero, con todos los ficheros que se piden en el enunciado.

Para reducir la medida de los ficheros, hay que eliminar lo que genera el compilador. Podéis utilizar la opción “clean” del workspace o eliminarlos directamente.

Hay que hacer la entrega en el apartado de entregas de AC del aula de teoría.

Presentación

Esta práctica culmina el proyecto empezado en la práctica 1.

Esta última práctica tiene **dos versiones**, una que contempla la totalidad de los contenidos de la asignatura, y una que sólo contempla los más importantes de cara a las futuras asignaturas. En el enunciado se refiere a estas dos versiones como **versión simplificada** y versión **completa**. Con la versión simplificada se puede obtener hasta un 80% de la nota de la PR2, con la versión completa hasta un 100%.

Competencias

Transversales

- Capacidad de comunicación en lengua extranjera.

Específicas

- Capacidad de diseñar y construir aplicaciones informáticas mediante técnicas de desarrollo, integración y reutilización.
- Conocimientos básicos sobre el uso y la programación de los ordenadores, sistemas operativos, bases de datos y programas informáticos con aplicación a la ingeniería

Objetivos

- Analizar un enunciado y extraer los requerimientos tanto de tipos de datos como funcionales (algoritmos)
- Analizar, entender y modificar adecuadamente código existente
- Saber utilizar tipos de datos abstractos
- Saber utilizar punteros

Recursos

Para realizar esta actividad tenéis a vuestra disposición los siguientes recursos:

Básicos

- Materiales en formato web de la asignatura
- Laboratorio de C

Complementarios

- Internet: Una forma efectiva de encontrar información ante cualquier duda sobre el lenguaje C es buscarla a través de un buscador.

Criterios de valoración

Cada ejercicio trae asociada la puntuación sobre el total de la actividad. Se valorará tanto la corrección de las respuestas como su completitud.

- Los ejercicios en lenguaje algorítmico, tienen que respetar el formato.
- Los ejercicios en lenguaje C, tienen que compilar para ser evaluados. En tal caso, se valorará:
    - Que funcionen
    - Que se respeten los criterios de estilo (Ved la Guía de estilo de programación en C que tenéis en los materiales del laboratorio)
    - Que el código esté comentado (preferiblemente en inglés)
    - Que las estructuras utilizadas sean las correctas

Descripción del proyecto

Junto con el enunciado se os facilita un nuevo workspace Codelite basado en la solución de la práctica 1 con código y ficheros adicionales. Este workspace será la base para esta segunda práctica.

En esta práctica trabajaremos con una estructura más compleja que contendrá un resumen de información de una sección, con el TAD pila, los punteros y la memoria dinámica.

En esta práctica utilizaremos:

- Un algoritmo de inserción ordenada basado en un algoritmo de busca para ordenar una tabla.
- Una estructura para almacenar la composición de una sección de la biblioteca, sus subsecciones y los libros que componen cada una de ellas.
- Una pila para simular la pila de libros devueltos a la biblioteca que están pendientes de ser colocados en el estante correspondiente.
- La memoria dinámica para no tener la restricción de número máximo de libros.

NOTA IMPORTANTE TANTO PARA VERSIÓN SIMPLIFICADA COMO COMPLETA

Hay en el código que os proporcionamos con el enunciado una serie de marcas para poder implementar la versión completa en el mismo proyecto que la versión simplificada. Son:

#ifdef SIMPLE_VERSION

#endif

#ifdef COMPLETE_VERSION

#endif

Mientras hacéis la versión simplificada no tenéis que escribir ninguna línea de código dentro del bloque delimitado con #ifdef COMPLETE_VERSION y #endif.

Si implementáis la versión completa, podéis elegir antes de compilar y ejecutar qué versión queréis. El archivo data.h contiene estas líneas de código:

/* Uncomment the practice version you want to run */

#define SIMPLE_VERSION

//#define COMPLETO_VERSION

Para ejecutar los fragmentos que corresponden a la versión completa hará falta que comentéis poniendo // ante la primera línea y sacándolo de la segunda.

[15%] Ejercicio 1: Ordenación de una tabla

En este ejercicio se pide que implementáis las siguientes acciones y funciones que encontraréis en **books.c**:

a) La función:

tError bookTable_sortedAdd(tBookTable *tabBook, tBook book)

- Dada una **tabla de libros**
- Ordenada según el criterio de la función ***book_cmp***
  - Por los campos sección, subsección, autor e ISBN)
  - De menor a mayor, y un libro,
- Si hay bastante espacio en la tabla,
  - Desplace una posición hacia el final todos los libros mayores que él
  - E inserte el libro en la posición pertinente

b) La acción:

void bookTable_sort(tBookTable tabBook, tBookTable *result)

- Dada una **tabla de libros**
- Nos **devuelva** una **tabla con los mismos libros**
- Ordenada según el criterio de la función ***book_cmp***
  - Por los campos sección, subsección, autor e ISBN
  - De menor a mayor.
- Se tendrá que usar:
  - Un algoritmo de recorrido por la tabla original
  - Y la función anterior para ir insertando de uno en uno cada libro de manera ordenada en la tabla de resultado, que primero habremos vaciado.

[5%] Ejercicio 2: Definir una estructura compleja

- En este ejercicio definiremos un tipo de datos que guardará la información de una sección de la biblioteca.
- Para **cada sección** nos interesará saber:
    - La **lista de subsecciones** que la forman
        - Y de **cada subsección:**
            - La **lista de libros** que contiene
- Completa la implementación de los tipos de datos
    - **tSubInfo**
    - **tSectionInfo**
- Que encontrarás en el fichero **data.h** para que permita guardar esta información:

tSubInfo:

- **id**: Identificador de la subsección (un carácter).
- **subBooks**: Vector con las posiciones que ocupan los libros de esta subsección dentro de la tabla de libros.
- **totSubBooks**: cantidad total de libros de la subsección

tSectionInfo:

- **section:** Estructura de tipo tSection
- **secSubs**: Vector con todas las subsecciones (tipos tSubInfo) de la sección (pueden haber como máximo 10) que tengan algún libro
- **totSecSubs**: cantidad total de subsecciones de la sección
- **totSecBooks**: cantidad total de libros de la sección

Nota: La cantidad máxima de libros dentro de cada subsección será la cantidad máxima de libros que podemos tener, definida por la constante **MAX_BOOKS**.

[25%] Ejercicio 3: Llenar la estructura de información

Utilizando el tipo **tSectionInfo** del ejercicio anterior, en este ejercicio definiremos los métodos para la manipulación de este tipo.

Las funciones siguientes las encontrarás en el fichero **info.c**.

a) Completa el código de la función:

tError si_getSectionInfo(tBookTable tabB, tSectionTable tabS, char sectionId, tSectionInfo *si)

que dada

- una tabla de libros
- una tabla de secciones
- un identificador de sección
- nos devuelva:
    - en un parámetro de salida de tipo **tSectionInfo**
        - la composición de la sección en subsecciones
        - y los libros contenidos en cada una.
- Esta función devolverá:
    - **OK**: Si todo ha ido bien y se devuelve la información pedida.
    - **ERR_INVALID_DATA**: Si la sección no existe en la tabla de secciones
    - **ERR_ENTRY_NOT_FOUND**: Si no hay libros en esta sección.

b) Completa el código de la función:

tBook si_getBook(tBookTable tabB, tSectionInfo si, unsigned int nSub, unsigned int nBook)

- Dada:
    - una tabla de libros,
    - una estructura de información de sección
    - y dos enteros
        - nSub
        - nBook
- Devuelva:
    - El **libro** número **nBook**
    - De la **subsección** número **nSub** de la sección
    - Según la ordenación que hemos establecido al construir la información de la sección.

c) Completa el código de la acción:

void si_listSectionInfo(tBookTable tabB, tSectionInfo si)

- Dada:
  - Una tabla de libros
  - Una estructura de información de sección
- Escriba por pantalla la composición de la **sección**:
  - Identificador y nombre
  - Total de subsecciones
  - Total de libros
  - Y para cada subsección
    - el identificador
    - el total de libros
    - y la lista de libros
- Es recomendable:
  - Usar la función **si_getBook**
  - Y las acciones **getSectionStr** y **getBookStr**
- Esta función:
  - Os permitirá comprobar mediante las opciones del menú
    - 1) Load - que carga la tabla books
    - Y después la 4) Manage Sections
    - Seguida de 3) Section Info
  - Que la construcción realizada en el apartado **a)** es correcta.

[25%] Ejercicio 4: Tipos Abstractos de datos

En este ejercicio definiremos:

- una **pila** para guardar elementos de tipos libro
- y los métodos básicos para acceder a esta pila.

Completa en los ficheros **stack.h** y **stack.c** el código de los apartados de este ejercicio.

Se pide:

a) Define en stack.h el tipo de datos **tBookStack** que represente una pila de elementos del tipo **tBook**.

Nota: La cantidad máxima de elementos en la pila será el número máximo de libros que puede guardar nuestra aplicación, identificado por la constante **MAX_BOOKS**.

b) Implementa en stack.c la acción:

void bookStack_create(tBookStack *stack)

que dado un parámetro de entrada/salida de tipo **tBookStack**, lo inicialice a una pila vacía.

c) Implementa la función:

tBoolean bookStack_empty(tBookStack stack)

que dada una pila nos indique si está vacía (TRUE) o si contiene elementos (FALSE).

d) Implementa la función:

tError bookStack_push(tBookStack *stack, tBook newElement)

- Dado:
  - un parámetro de entrada/salida de tipo **tBookStack**,
  - y un parámetro de entrada de tipo **tBook**,
- Añada este último libro a la pila (encima de todos).
- Esta función devolverá:
  - un valor **OK** en caso de que se haya podido añadir el nuevo elemento a la pila,
  - o un valor **ERR_MEMORY** en caso de que no haya bastante espacio en la pila para añadir el nuevo elemento.

e) Implementa la función:

tError bookStack_pop (tBookStack *stack, tBook *elemento)

- Dado:
  - Un parámetro de entrada/salida de tipo **tBookStack**
  - Y un parámetro de salida de tipo **tBook**
- Nos devuelva en este parámetro de salida
  - El primer elemento de la pila (el de encima) sacándolo de la pila.
- Esta función devolverá:
  - Un valor **OK** en caso de que se haya podido sacar el elemento de la pila
  - O un valor **ERROR** en caso de que la pila esté vacía y no se haya podido sacar ningún elemento.

[10%] Ejercicio 5: Navegación por el TAD pila

Construiremos acciones que trabajen sobre la pila que hemos definido. En particular se pide lo siguiente **en stack.c**:

- Implementa la acción
  - ***void bookStack_transfer(tBookStack *stack_dest, tBookStack *stack);***
  - Dados dos pilas (con o sin elementos)
  - Las junte en una sola
  - Pasando todos los elementos de la **segunda** pila a la **primera**
  - En orden invertido,
    - Esto es lo que podemos conseguir usando los únicos movimientos posibles con pilas
    - Sacar el elemento superior o poner el elemento superior

- Implementa la función que:
  - ***tError bookStack_search(tBookStack *stack, char *ISBN, tBook *book);***
  - Dada una pila de libros
  - Busque el libro con el **ISBN** dado
  - Si lo encuentra:
    - Lo elimine de la pila
    - Y lo saque como parámetro de salida.
  - Si no encuentra el libro:

        - Tendrá que devolver el valor **ERR_ENTRY_NOT_FOUND**.
    - Podéis utilizar la acción del apartado anterior en parte de la implementación.

FIN VERSIÓN SIMPLIFICADA

VERSIÓN COMPLETA

[20%] Ejercicio 6: Vector de punteros y memoria dinámica

Durante toda la práctica hemos sido utilizando estructuras con memoria estática para guardar todos nuestros datos. Esto nos ha obligado a poner un límite en el número de registros que se podían guardar en las tablas. En este ejercicio nos proponemos eliminar esta restricción con el uso de punteros.

Se pide:

a) Modifica el campo **subBooks** de la estructura **tSubInfo** definida en el ejercicio 2 y todas las acciones y funciones que trabajan con esta estructura (**si_getSectionInfo, si_getBook, si_listSectionInfo**) para que se adapten a este nuevo tipo de datos:

- **subBooks**: será un vector con punteros a los elementos **tBook** de la tabla de libros que corresponden a libros de esta subsección.
- Identificamos el final del vector con un NULL.
- **Nota:** Un vector de punteros se declara como cualquiera otro tipo de vector, y cada uno de sus elementos es un puntero. Por ejemplo aquí se define un vector de punteros a int.

```
int* vector[3];
int a=5;
vector[0]=&a;
vector[1]=NULL;
```

b) Modifica la definición de **tBookTable** del fichero **data.h** para que utilice memoria dinámica.

c) Modifica el método **bookTable_init** para que se adapte a la nueva definición del apartado b).

d) Modifica los métodos **bookTable_add** y **bookTable_sortedAdd** para que utilicen memoria dinámica. En caso de que no haya bastante espacio de memoria, estos métodos devolverán el error **ERR_MEMORY.**

e) Modifica el método **bookTable_del** para que utilice memoria dinámica.

f) Completa el código de las acciones **bookTable_release y appData_release**, para que reciban una tabla y eliminen la memoria que esté utilizando.

Nota: En este caso, como los tests están pensados para poder pasar en ambas versiones, no se liberará la memoria de las tablas que se utilizan. Este hecho está previsto y no hace falta que modifiques los tests para solucionarlo. Al finalizar la ejecución la memoria quedará liberada. Además, **los tests para comprobar la máxima capacidad de la tabla fallarán**, puesto que en este

caso las tablas no tienen esta limitación.