Algoritmos y programación II

2013

TP 1

Federico L. Amura

Análisis y diseño

El programa a desarrollar, simplificando, es un interpretador de comandos y procesador de base de datos, por lo tanto, es lógico empezar con esa división en el diseño.
El interpretador de comandos va a ser la entrada de la información y la encargada de llamar al comando correspondiente con los parámetros que sean necesarios, además también va a encargarse de la salida correspondiente según la devolución de cada comando, viéndolo de otro modo, esta seria la parte de entrada/salida del programa. Implementarlo de este modo aporta la facilidad de tener toda la parte de comunicación en un solo lugar, trayendo la ventaja de que si en un futuro quisiera modificarse la parte de comunicación con el usuario, por ejemplo agregando una interfaz grafica o pasando el programa a otro idioma, solamente se tiene que modificar esta parte.
El procesador de base de datos simplemente recibirá los comandos con sus parámetros para luego efectuar las operaciones necesarias con el fin de resolver y devolver un valor que indica como fueron el resultado de las operaciones. Este enfoque le deja todo el manejo de la información a esta parte, llegado a haber un cambio en la estructura del almacenamiento de la información o como debe esta procesarse, las demás partes permanecerían intactas ya que solo se modificaría esta. No se opto por dividir mas esta estructura, por ejemplo separar los comandos de administración de los de alumnos o separar las estructuras, ya que la separación extra podría llegar a provocar desorden y de esta manera los comandos están todos al mismo nivel pudiendo reutilizarse sin problemas y manteniendo un formato constante de la información del programa sin intermediarios innecesarios.

Interpretador de comandos

En este caso la entrada es el stdin, se realiza la entrada de los comandos utilizando un formato especifico, dato clave para el funcionamiento correcto.
Previo al desarrollo de esta parte hay que tomar en consideración como son los comandos para adaptar la entrada a lo que pueda llegar a ingresar. En el caso de los comandos que están incluidos tienen un máximo de 4 parámetros, para estos y el comando hay que ver cual es la máxima longitud que pueden tener de manera de preparar las cadenas con el tamaño adecuado para que venga lo que venga pueda almacenarlo sin problemas. En la tabla se pueden ver todas las longitudes e incluso ya se va empezando a definir como van a ser los comandos.

comando	longitud	parametro1	longitud	parametro2	longitud	parametro3	longitud	parametro4	longitud
agregar_curso	13	idc	10	descripcion	80	materia	10	vacantes	3
inscribir	9	padron	5	idc	10				
eliminar_curso	14	idc	10						
desinscribir	12	padron	5	idc	10				
listar_inscriptos	17	idc	10						
listar_cursos	13								
listar_cursos_materia	21	materia	7						
sesion_iniciar	14	padron	5						
sesion_inscribir	16	idc	10						
sesion_ver	10								
sesion_deshacer	15								
sesion_aplicar	14								
maximo	21		10		80		10		3

Sabiendo que cada linea equivale a un comando, y que en cada una viene primero escrito el comando, un espacio, y después los parámetros separados por comas, se desarrolla un proceso que primero toma el primer carácter de la entrada mediante fgetc, esto lo hace para verificar que de hecho no estemos en el EOF. De no estar en el EOF, consigue toda la linea con getline y ahora con toda la linea ya cargada en un vector de caracteres se procede a separarla en el comando y los parámetros. Esta acción se desarrolla con whiles, el primero siendo el correspondiente al comando y luego otro ciclo juntando los parámetros, la complejidad en esta parte es un poco superior dado que hay que distinguir no solo en que parámetro nos encontramos si no que puede ser que solo haya algunos parámetros y no todos. Acá se define la estructura “lectura” con el fin de guardar esta

información y permitir su uso. Todas las cadenas se almacenan con el formato de char* para facilitar esta parte, luego el comando hará la transformación en el tipo de dato que corresponda si hace falta.
Luego de tener la lectura en el formato que queremos pasamos a la parte del proceso, primero se tiene que ver cual es el comando ingresado comparándolo con cadenas de caracteres definidas y después se lo llama con la cantidad de parámetros que entraron y agregando los que hagan falta.
Se procesa el contenido, tarea a cargo de la otra parte del programa y este genera una devolución que vuelve a esta parte del programa. De acuerdo a esta devolución se imprime el resultado del proceso.
Luego se repite el ciclo con la siguiente linea hasta que termina la entrada.
Esta parte del programa tiene un orden de crecimiento de n, siendo n el largo de la entrada, ya que debe recorrerla toda elemento a elemento procesando cada carácter y luego llama a cada comando que corresponde.
Esta parte posee algunos pocos comandos a explicar.

void iniciar_parser(lecturaformateada* leer):

Este comando lo único que hace es preparar la estructura necesaria para el ingreso de los comandos.
Realiza un malloc por cada array de la estructura.
Es de orden constante.

void reiniciar_parser(lecturaformateada* leer);

Con esta instrucción vaciamos la estructura que utilizamos para leer.
Simplemente le asigna al inicio de cada cadena de caracteres el carácter nulo, de manera de indicar que ahí se termina el contenido y lo siguiente es basura.
Es de orden constante.

void finalizar_parser(lecturaformateada* leer)

Este proceso simplemente finaliza la estructura adecuadamente para evitar perdidas de memoria al salir del programa.
Realiza un free por cada cadena de caracteres utilizada.
Es de orden constante.

bool conseguir_comando(lecturaformateada* leer);

Es el proceso que realiza básicamente todo el trabajo de esta parte que ya fue explicado.
Es de orden n porque a pesar de que tiene un par de whiles anidados, uno es simplemente para controlar las asignaciones y en definitiva lo que realiza es un numero definido de operaciones para cada carácter de la cadena que ingreso.

Procesador

Recibiendo el comando a ejecutar, con sus parámetros en el formato correcto, esta parte simplemente ejecuta todas las operaciones necesarias para conseguir el objetivo deseado. Básicamente se implementa cada comando por separado y así debe analizarse.

base_t* inicializar_base_de_datos();

Este proceso es necesario para el desarrollo ya que en la base de datos que se inicia es donde se almacena la información, en este caso se crea una lista de cursos.
Es de orden constante ya que no recibe información y solo se encarga de inicializar la base.

void cerrar_base_de_datos(base_t* basedatos);

Este proceso cierra la base de datos, eliminando todo lo que tiene adentro y liberando la memoria. Se podría decir que este proceso seria de orden n^2 ya que destruye elemento a elemento y cada elemento puede a su vez tener una lista dentro.

sesion_t* sesion_crear();

Con este comando se crea la estructura que almacena la información de la sesión activa habilitando todos los comandos que hagan uso de esta. No se la inicializa aquí. Es de orden constante.

void sesion_destruir(sesion_t* sesion);

Este proceso destruye la estructura de la sesión. Es de orden de crecimiento n si la sesion no se cerro ya que debe destruir cada elemento de la lista que incluye esta estructura, si la sesion fue cerrada, es de orden constante.

char* obtener_padron(sesion_t* sesion);

Con este proceso se recibe el padrón de la sesión activa, se implemento con el fin de poder recuperar el padrón en el interprete de comandos una vez iniciada la sesión ya que la estructura esta en otra parte del programa. Es de orden constante.

int agregar_curso(char* idc,char* descripcion,char* materia,char* vacantes, base_t* basedatos);

Este comando agrega un curso a la base de datos con la información dada en los parámetros. Primero verifica si el curso ya existía con la función existe_curso, de no existir lo crea con la función crear_curso y lo agrega a la lista de la base de datos.
Es de orden de crecimiento n, siendo n la cantidad de cursos en la base, por la función existe_curso que tiene que recorrer todos los cursos.

int inscribir(char* padron, char* idc, base_t* basedatos);

Con este comando se inscribe al padrón en el curso idc.
Primero verifica que el curso exista y su ubicación con existe_curso, lo saca de la base de datos, verifica si el alumno no estaba ya inscripto y de no estarlo lo agrega, luego vuelve a agregar el curso a la base de datos.
Es de orden de crecimiento n, ya que debe recorrer toda la base de datos para buscar el curso y

luego recorrer todos los anotados en ese curso pero de maneras separadas.

int eliminar_curso(char* idc,base_t* basedatos);

Este comando elimina un curso de la base de datos.
Primero verifica que exista en la base y donde, luego lo busca con un iterador, lo saca de la base y lo destruye adecuadamente.
Es de orden de crecimiento n por tener que recorrer toda la base.

int desinscribir(char* padron, char* idc, base_t* basedatos);

Con este comando se puede desinscribir a un alumno de un curso.
Primero consigue el curso con el comando conseguir_curso, luego busca con un iterador al alumno que tiene que borrar y lo elimina de manera acorde.
Es de orden n por tener que recorrer los cursos y los inscriptos.

int listar_inscriptos(char* idc, base_t* basedatos);

Este comando imprime todos los inscriptos en el curso con el estado de su inscripción.
Luego de verificar que el curso exista, lo saca de la base, consigue el numero de vacantes y mediante un iterador interno imprime todos los inscriptos siendo todos regulares hasta que el numero de vacantes llega a cero y pasan a ser condicionales, antes de terminar vuelve a poner el curso en la base.
El orden de este comando es n por tener que recorrer los cursos, y luego los inscriptos.

void listar_cursos(char* filtro,base_t* basedatos);

Con este comando se imprimen los cursos disponibles, se puede especificar que solo imprima algunos cursos mediante el filtro.
Primero crea un iterador a la lista de cursos, con este iterador va recorriendo la lista e imprimiendo con la información correspondiente y una vez recorridos todos destruye el iterador.
Como tiene que recorrer todos los cursos, es de orden n.

int sesion_iniciar(sesion_t* sesion,char* padron);

Mediante este comando se inicia la sesión.
Primero verifica que no haya una sesión ya abierta, esto lo hace viendo si el padrón de la estructura sesión es distinto de “00000” ya que ese padrón es invalido se toma como el caso de no haber sesión iniciada, de no haberla, inicia la sesión.
Es de orden constante.

int sesion_inscribir(char* idc,base_t* basedatos, sesion_t* sesion);

Con este proceso anotamos el padrón de la sesión activa en el curso especificado.
Lo primero que hace es verificar que de hecho haya una sesión iniciada. Luego verifica si el curso existe, esto además, nos da su posición en la base de datos. Además se verifica si no estaba ya programada una inscripción a ese curso dentro de la sesión. Luego se consigue el curso al que se quiere anotar, se verifica que no este anotado en el curso y de no estarlo se pasa a crear la inscripción y almacenarla en la estructura sesión adecuadamente.
Dado que tiene que recorrer las listas de las inscripciones programadas, la lista de cursos y la lista de inscriptos al curso, el orden de crecimiento es n.

void sesion_ver(sesion_t* sesion);

Este proceso nos muestra el estado de la sesión junto con las inscripciones hechas.
Luego de verificar que hay una sesión en curso, el proceso para a imprimir el padrón, seguido de las inscripciones que se hicieron en esta sesión.
Es de orden n ya que debe procesar todas las inscripciones dentro de la sesión.

int sesion_deshacer(sesion_t* sesion);

Con este comando deshacemos la ultima inscripción que hicimos en la sesión activa.
Viendo que la sesión haya sido iniciada, se verifica además que haya inscripciones, de no ser así devuelve el valor correspondiente. De haber inscripciones crea dos iteradores y los hace avanzar hasta que el primero llega al fin de la lista, quedando el otro en el ultimo elemento, este se deshace y se destruyen los iteradores adecuadamente.
Por recorrer todas las inscripciones, es de orden n.

int sesion_aplicar(sesion_t* sesion,base_t* basedatos);

Este proceso aplica todas las inscripciones que hicimos en la sesión y luego cierra la sesión.
Primero verifica que haya una sesión activa y luego, por cada inscripción hecha en la sesión busca el curso y lo inscribe. Luego cierra la sesión.
Es de orden n^2 ya que por cada inscripcion que procesa tiene que buscar el curso correspondiente en la base de datos para realizar la inscripción.

curso_t* crear_curso(char* idc,char* descripcion,char* materia,char* vacantes);

Este comando crea y llena la estructura de los cursos con el fin de ser agregados a la base de datos.
El proceso es muy simple, se va pidiendo la memoria para cada elemento, luego de verificar que la memoria se haya conseguido correctamente se procede a llenarla con la información que recibe como parámetros.
Es de orden constante.

void destruir_curso(curso_t* curso);

Este proceso destruye la estructura curso adecuadamente para no perder memoria.
Se van haciendo los frees correspondientes de manera acorde.
Es de orden constante.

int existe_curso(char* idc,lista_t* lista);

Este comando tiene una especie de doble función, no solo nos dice si el curso existe ya en la base de datos si no que además nos dice en que posición, lo cual es útil después si queremos ir a conseguirlo con un iterador.
En primer lugar crea un iterador en la lista de cursos, luego lo va adelantando mientras aumenta un contador hasta que encuentra en el elemento actual el mismo idc que se busca, en ese momento guarda el contador y sale de la función destruyendo el iterador y devolviendo el valor del contador, que contiene la cantidad de avances que tuvo que hacer o -1 en caso de no haber encontrado el curso.
Es de orden n ya que debe recorrer la lista hasta encontrar el curso o en su totalidad.

int alumno_inscripto(char* padron,lista_t* lista);

Esta función es similar a existe_curso pero buscando un padrón en la lista de inscriptos a un curso. Su funcionamiento es similar, pero no pueden juntarse dado que las estructuras que se manejan son distintas.
También es de orden n por tener que recorrer una lista.

curso_t* conseguir_curso(char* idc, lista_t* lista);

Con este proceso se puede conseguir un curso de la lista, útil para ciertos otros procedimientos donde hay que realizar cambios sobre algún curso.
Luego de ver que el curso existe y tener su posición llega hasta ahí con un iterador, lo consigue y lo devuelve luego de eliminar el iterador correspondiente.

bool imprimir_inscripto(char* padron, int* estado);

Este pequeño proceso nos sirve para imprimir un inscripto de manera acorde junto con su estado. Primero imprime el padrón y luego verificando que haya mas vacantes imprime si esta como regular o condicional. Esta función esta hecha para utilizarse con un iterador interno.
Es de orden constante.

Consideraciones extra

La información guardada se mantiene en forma de listas. Se opto por las listas en lugar de pilas o colas ya que estas pueden comportarse como cualquiera de las otras dos simplemente con un poco de ingenio, además estas poseen mas herramientas disponibles, como los iteradores, y fue la estructura mas probada lo cual hace que sea mas confiable y con menos posibilidades de encontrarnos con algún error en su implementación.
El código fuente se incluye en otro archivo separado generado automáticamente, en este archivo también esta la implementación de la lista por si llegara a ser necesario.