Programación II, 2016-2017
Escuela Politécnica Superior, UAM

Práctica 2: TAD Pila

PARTE 2A: PREGUNTAS SOBRE LA PRÁCTICA

Ejercicio #1:

Indicad qué modificaciones habría que hacer a la pila del ejercicio 3 para que la cima de la pila fuera de tipo puntero, en particular especificad qué ficheros habría que modificar y cuáles serían estos cambios:

En stack_fp.h no habría que modificar nada ya que el prototipo de las funciones seguiría siendo el mismo.

En stack_fp.c es el único sitio en el que habría que realizar modificaciones, debido a que la modificación realizada es en la estructura de la pila, a la que solo tenemos acceso dentro de stack_fp.

En esta, el top sería de tipo Element** en vez de tipo int. Además habría que cambiar todas las funciones que hagan uso del top.
En stack_ini inicializaríamos el top a NULL.
En stack_destroy iríamos bajando el top a medida que liberamos hacia adonde apunta, para terminar liberando el puntero a Stack.
En stack_push copiamos el elemento a donde apunta el top, y avanzamos el top una posición. Si inicialmente apunta a NULL hacemos que apunte al elemento en la posición cero.
En stack_top devolvemos la dirección a la que apunta el doble puntero.
En stack_pop retrocedemos el puntero una posición y devolvemos la dirección hacia donde apuntaba en un inicio. Si la pila fuera a quedarse vacía, igualamos el top a NULL.
En stack_isEmpty y stack_isFull comprobamos si la dirección del primer o último elemento (respectivamente) coinciden con la dirección a la que apunta el top en cada caso. Si es así devolvemos TRUE.

En p2_e3.c no habría que modificar nada ya que el prototipo de las funciones seguiría siendo el mismo, y su uso no variaría. Además, el cambio solo afecta a la estructura de la pila, a la que no podemos acceder desde p2_e3.c.

En el resto de archivos tampoco habría que modificar nada debido a que el único archivo que tiene acceso a la estructura de Stack (que es donde se realiza el cambio) es stack_fp.c.

PARTE 2A: PREGUNTAS SOBRE LA PRÁCTICA

Ejercicio #2:

Una aplicación habitual del TAD PILA es para evaluar expresiones posfijo. Describe en detalle qué TADs habría que definir/modificar para poder adaptar la pila de enteros (P2_E1) en una que permita evaluar expresiones posfijo. Realiza el mismo ejercicio pero con la pila general (P2_E4).

Para poder evaluar expresiones posfijo con la pila de enteros, necesitaríamos poner un switch al ir leyendo los caracteres de la expresión, en el que se mire si el carácter es +, -, *, /, ^. En caso de que sea un número, le hace push en la pila, y en caso de que lea un operador, en el switch programaremos que se realice la operación que representa el carácter de ese operador con los dos últimos elementos de la pila. Definiríamos por tanto los TADs usados en el ejercicio p2_e1.

Es decir, si la pila tiene 2,3 y lee un *, hará dos pop de la pila, los multiplicará y guardará un 6 en la pila.
Por último, debemos programar que haga un último pop para devolver el resultado, y comprobar que la pila ha quedado vacía.

En el caso de tener la pila del ejercicio 4, tendríamos que pasarle las funciones de imprimir, destruir y copiar enteros, y ya tendríamos nuestra pila como en el caso 1, por lo que solo habría que repetir el proceso especificado en el párrafo anterior.

PARTE 2B: MEMORIA SOBRE LA PRÁCTICA

Ejercicio #1:

Durante el ejercicio #1 tuvimos que implementar el TAD Pila y el TAD Elemento como envoltura de un puntero a entero.

Para ellos definimos los prototipos de las funciones en stack.h y element.h e implementamos la Estructura de Datos y las primitivas en stack.c y element_int.c

Tras ellos hicimos un programa que creara una pila con un número de elementos introducidos por un argumento de entrada. Después el programa divide la pila en una pila par y una impar. Por último la imprime por pantalla.

Además creamos la función:

```
void stack_printCheck(Stack* stack) {
   switch (stack_isEmpty(stack)) {

      case TRUE:
         printf("(no llena, vacía)");
         break;

      default:
         switch (stack_isFull(stack)) {
            case TRUE:
               printf("(llena, no vacía)");
               break;

            default:
               printf("(no llena, no vacía)");
         }
   }
}
```

Que imprime por pantalla el estado de la pila que pasamos como argumento, con la finalidad de ahorrar código.

Ejercicio #2:

En este ejercicio modificamos la estructura y primitivas del TAD Element, ya que lo hicimos como envoltura de un puntero a Node. La interfaz, como es lógico, no se ve afectada.

Reutilizamos la función read_graph_from_file de la práctica 1 para leer un grafo desde un archivo de texto. Una vez leído el grafo, vamos nodo por nodo comprobando sus conexiones entrantes y salientes y añadiendolas a una pila en ese orden.
Finalmente imprimimos la pila resultante.

PARTE 2B: MEMORIA SOBRE LA PRÁCTICA

Ejercicio #3:

En este ejercicio debíamos crear una función que recorriese un grafo a partir de un nodo dado. Para ello, modificamos la estructura del nodo en el fichero node.c añadiendo un campo de tipo Color (una enumeración de int definida en node.h, análogo a lo hecho en types.h). Modificamos también las funciones de node_equals y node_ini, inicializando los nodos con un color inicial 'WHITE'. Además incluimos dos funciones destinadas a leer y escribir el color de un nodo (añadiendo su prototipo a la interfaz para hacerlas públicas).

Una vez modificada la estructura, implementamos la función graph_discover_from_node siguiendo el pseudocódigo dado. Decidimos que el retorno de la función fuera el grafo con el color de los nodos modifcados.

Para implementar la función de si existe o no camino, decidimos que tuviera un retorno de tipo Bool y argumentos de entrada de tipo Graph* e int (para las Ids). Simplemente llamamos a la función de recorrer el grafo desde un nodo dado, y después comprobamos el color del nodo destino. Si el color es 'BLACK' entonces es que ha sido descubierto y visitado desde el nodo origen y por tanto existe un camino.

En la función opcional para devolver un camino, decidimos implementar la información del nodo padre con una tabla de IDS, siendo una fila la ID del nodo y la fila inferior la ID del nodo padre (desde el que está conectado). Para devolver el camino usamos dos pilas, una auxiliar y la pila con el camino que vamos a devolver. En la función simplemente vamos recorriendo el grafo como hacemos en la función graph_discover_from_node, pero anotando esta vez quien es el nodo padre de cada nodo descubierto (usamos -2 para la id del nodo padre del nodo inicial). Una vez se encuentra el nodo destino, lo añadimos a la pila del camino, seguido del nodo padre, y así con cada nodo hasta llegar al inicial, que se añade fuera del bucle.

Como ayuda, hemos implementado una función que nos permite encontrar el indice de un id dado en la tabla de IDs.

PARTE 2B: MEMORIA SOBRE LA PRÁCTICA

Ejercicio #4:

Finalmente, en el cuarto ejercicio hemos implementado de nuevo Stack.c, siguiendo las indicaciones de la práctica para crear un TAD Stack más general que los anteriores. Para ello hemos hecho uso de los punteros a funciones, de forma que la propia pila almacena información de que función tiene que usar dependiendo del tipo de dato que almacene.

La generalización se basa en guardar una serie de items de tipo punteros a void*, y en la inicialización de la pila pasaremos como argumento como ha de imprimir, copiar y destruir los elementos que guardemos en ella.

Hemos modificado la interfaz de la pila añadiendo los typedef necesarios, y hemos creado la interfaz functions.h para poder usar las funciones dadas en el PDF de la práctica bajo el nombre de functions.c. Además, no hemos hecho uso del TAD Element.

En el ejercicio nos pedían realizar un main que repitiera lo mismo que en el segundo ejercicio de la práctica pero que simultáneamente añadiera ids de nodos a una pila de punteros a enteros y nodos a una pila de punteros a nodo.

Para ello hemos hecho uso del mismo TAD Stack, pero inicializando cada pila con unas funciones distintas. El algoritmo viene a ser el mismo que en el p2_e2.c, adaptándonos a la nueva implementación del TAD.

Conclusiones finales:

La práctica ha resultado beneficiosa para la comprensión de la abstracción de los TADs, y de cómo si queremos hacer un cambio, solo tendremos que modificar un fichero por lo general.

Se ha visto reforzado el uso de la memoria así como el control de errores, muy presente en las clases teóricas.

La parte más fácil de la práctica puede ser la creación propia de los TADs, y la más difícil sin duda la implementación del algoritmo que devuelve el camino buscado. Sin embargo, el trabajo habría sido menos si hubiéramos modificado la estructura del Nodo añadiendo un puntero a puntero a Nodo que apunte al nodo padre del mismo.

Como aspectos nuevos de programación podemos recoger tanto el uso de punteros a funciones, como el uso de los argumentos de entrada de la función main (argc y argv[]) y el uso de flags a la hora de compilar el programa.