PROG II – Memoria de la Práctica 2

Pareja 08
Javier Barranco & Martín Sánchez

Pregunta 1

Evaluación de posfijos:

Para evaluar expresiones posfijo es necesario utilizar los TADs Pila y EleStack usados en el ejercicio P02_E02. No obstante, tendremos que añadir una función, al TAD EleStack, capaz de pasar los caracteres leídos a elementos que guardan enteros y pueden ser almacenados en la pila.

Como la expresión posfijo viene dada en una cadena de caracteres tendremos que usar la función atoi() para pasar estos caracteres a enteros.

La pila tendrá la siguiente estructura:

```
Struct _Stack {
        EleStack *item[MAXSTACK];
        int top;
}
```

El elemento tendrá la siguiente estructura:

```
Struct _EleStack {
        int *info;
}
```

La función que añadimos al TAD EleStack es la siguiente:

EleStack* EleStack_LeerItem(CadenaCaracteres S, Bool* Operador) - Dada una cadena de caracteres esta función leerá el siguiente elemento de la cadena de caracteres. Si el carácter leído es un numero la función devolverá un EleStack con la información del entero leído (para esto habrá que pasar el carácter a un entero usando atoi()). Si el carácter leído es un operador la función pondrá el valor de Operador a TRUE y devolverá NULL (de esta manera podemos hacer el doble pop cuando leamos un operador). Por ultimo, si leemos el carácter EOS (End of String) devolveremos NULL y dejaremos Operador a FALSE (así sabremos cuando hemos terminado de leer la cadena de caracteres).

Usaremos las funciones pila_ini, pila_push, pila_pop y pila_destroy para manejar la pila durante la evaluación de posfijos. Las funciones pila_Llena y pila_Vacia las usaremos para hacer comprobaciones de error.

Pregunta 2

Decisiones de diseño e implementación en cada uno de los ejercicios intermedios del 1 al 4.

· Ejercicio 1

En este ejercicio hemos optado por realizar la pila utilizando un entero como tope, el cual se inicializa a -1 y apunta al índice en el cual se halla el último elemento de la pila. Además, la pila guarda punteros a elementos, los cuales tienen una estructura de envoltura de nodo. Para las funciones de la pila nos hemos ceñido a lo visto en clase, mientras que elestack consiste en llamadas a funciones de nodo y los usos básicos que se dará al elemento en general.

· Ejercicio 2

En este ejercicio se nos planteaba la tarea de implementar la pila ya creada en el apartado anterior, la cual consistía en punteros a elementos. Estos elementos son un TAD auxiliar que la pila usa para poder

copiarlos, imprimirlos y operar con ellos en general. Así pues en el primer ejercicio se implementó el archivo "elestack-node.c" como elemento, el cual permite trabajar con nodos.

Sin embargo, en este apartado lo que se buscaba era crear una pila de enteros, así pues nuestra solución consistió en modificar exclusivamente el archivo .c utilizado para guardar la estructura de elemento, por lo tanto, el resto de archivos (elestack.h, stack_elestack.c, stack_elestack.h) no sufrieron ninguna modificación aprovechando lo ya escrito. Se creó pues un archivo "elestack-int" el cual permite envolver un entero en una estructura de elemento, con la cual la pila puede trabajar.

En conclusión, gracias a la abstracción realizada en la implementación de la pila (trabaja con la estructura "elemento", la cual guarda en el ejercicio 1 nodos, y en este, enteros), pudimos implementar una estructura elemento ajena a la pila.

Específicamente, nuestro archivo "elestack-int.c" consiste en una estructura de elemento que encapsula un entero. En particular, hemos optado por no desarrollar otro archivo .c que contenga las operaciones de copia, comparación e impresión de enteros (como hemos hecho con "elestack-node"), sino que hemos declarado unas funciones privadas que realizan dichas acciones, para poder así evitar modificar el código en exceso y ganar en abstracción.

Así pues, se planteó el uso de una estructura entero como análoga a la estructura nodo, sin embargo, debido a que los enteros son un tipo de datos propio de C, trabajar con ellos directamente resultó más sencillo. Por ello, en "p2_e2.c" lo que se declara como información del elemento es un puntero a nodo que guarda la dirección de un entero, y como lo que se guarda es una copia no hace falta reservar memoria en el main.

· Ejercicio 3

En este apartado lo que se propone es desarrollar una implementación de pila capaz de funcionar ajena al TAD elemento. Lo que se pretende es; por un lado simplificar el uso de la pila, pues no hará falta crear un elemento, ponerle una información y luego meterlo en la pila, sino que se podrá trabajar directamente con la pila; por otro lado, la manera en la que se implementaron las pilas anteriores de nodos y enteros, éstas no eran compatibles en un mismo programa, pues dependen de la estructura elemento, la cual solo puede ser de un tipo.

Así pues, lo que se hace para poder realizar esta pila mejorada es, modificar su estructura para que pueda guardar tres punteros a funciones que realizan las únicas operaciones que la pila necesita para poder destruir, copiar e imprimir sus elementos, en el resto de acciones la pila es independiente al tipo de su contenido. Cuando se inicializa la pila, se le pasa como argumento estas funciones, las cuales siempre trabajan con punteros a void. Dentro de la implementación de la pila, el resto apenas varía. En vez de usar elementos se utilizan punteros a void, y para las operaciones de destrucción, copia e impresión se llama a las funciones que la pila tiene guardadas como punteros.

Por lo tanto, la dificultad del ejercicio radica en que dichas funciones clave han de poder adaptarse a cualquier tipo de dato que se quiera guardar en la pila. Es por eso por lo que se emplean punteros a void y mediante castings podemos trabajar con ellos.

Así pues, nuestro primer pensamiento fue que, para poder pasar funciones capaces de destruir, copiar e imprimir nodos o enteros utilizando como argumentos y retornos punteros a void, sería necesario modificar los prototipos de las funciones "node_destroy", "node_copy", etc. Sin embargo esto implicaría subordinar los TADs nodo y entero a la estructura de nuestra pila, lo cual no tiene sentido y viola la mentalidad que buscamos.

Por lo tanto, lo que decidimos hacer fue que, si se necesita trabajar con la pila usando un tipo de dato que tenga ya implementadas las funciones de destruir, copiar e imprimir (como es el caso del nodo o del entero) pero sin el prototipo adecuado, lo que hacemos es declarar unas funciones auxiliares en el programa principal que tengan los prototipos necesarios para la pila (usando retornos y argumento como punteros a void en vez de nodos o enteros) y que mediante castings llamen a las funciones con el tipo de argumento adecuado y que devuelvan punteros a void.

Ejemplo:

```
void * fp_node_copy(const void * src) {
        return (void*) node_copy((Node*) src);
}
```

En conclusión, nuestra implementación para este ejercicio consiste en el uso de funciones auxiliares con el prototipo adecuado para guardarlas en la pila, gracias a lo cual no nos vemos obligados a modificar ninguno de los módulos ya implementados.

· Ejercicio 4

Este ejercicio consistió en darle una aplicación a la pila, usándola para realizar una búsqueda en profundidad a través de un grafo, para hallar así el camino entre dos nodos. Lo que se proponía era implementar una función "graph_findDeepSearch" (DFS para abreviar) capaz de, dados un grafo y dos ids uno destino y otro origen, encontrar un camino entre los nodos y, adicionalmente imprimir dicho camino. Para ello se sugería modificar la estructura de nodo, añadiéndole una etiqueta y un identificador de su antecesor, en el caso del grafo era necesario añadir una función primitiva nueva, capaz de realizar la DFS.

Nuestro primer planteamiento para decidir si existe un camino entre dos nodos fue crear en la función "graph_findDeepSearch" una pila que almacenara nodos y mediante el algoritmo propuesto, ir modificándolos hasta decidir si hay o no un camino. Sin embargo, este planteamiento conlleva ciertas dificultades. En primer lugar, en la pila se guardarían copias de los nodos del grafo, por lo que modificar los nodos de la pila no repercute en los del grafo. Por otro lado, cada vez que se quiera guardar un nodo habría que copiarlo, insertarlo y destruirlo, y en particular esta llevaría más coste de memoria que la solución proporcionada.

Por lo tanto, pensamos que en vez de guardar nodos, podríamos guardar sus identificadores. Sin embargo, esto nos llevaría a tener que buscar cada nodo cada vez que queramos acceder a él. Así pues, al final optamos por guardar sus índices, pues hallándonos en la estructura del grafo, podemos acceder a ellos en la función y además cada índice representa un nodo al que podemos acceder directamente y en tiempo O(1).

Así pues, para decidir si hay o no camino utilizamos una pila de índices. Por otro lado, a lo largo de función vamos modificando el campo AntId de cada nodo visitado, guardando en él, no una id, que tardaríamos en acceder al nodo, sino el índice del antecesor. De esta manera, para realizar la función capaz de imprimir el camino (la cual la implementamos como primitiva de grafo para ganar velocidad en acceso usando índices) tardamos O(1) en acceder al nodo previo.

En particular, nuestra solución consiste en una nueva primitiva en "graph.c" que recibe un FILE y un nodo destino que es el obtenido tras llamar a la función de DFS. Esta función se basa en la recursividad para ir imprimiendo los nodos en el orden correcto, como se accede mediante índices, el tiempo de ejecución será de O(1).

```
int graph_print_camino(FILE *pf, const Graph *g, const Node * pn) {
        if(pf == NULL || pn == NULL || g == NULL) return -1;

        Node * pnext = NULL;
        int ind;
         int nbytes = 0;

        // Capturamos el antId de pn
        ind = node_getAntId(pn);

        // Si hay un nodo antecesor...
        if(ind > 0) {
                // Si existe antecesor tras DFS, antId es su indice
                pnext = g->nodes[ind];
                nbytes = graph_print_camino(pf, g, pnext);
                nbytes += fprintf(pf, "\n");
        }

 nbytes += node_print(pf, pn);
 return nbytes;
 }
```

CONCLUSIONES

En definitiva, esta práctica ha resultado, a nuestro parecer, un claro ejemplo de la manera en la que va evolucionando una idea. Partimos del concepto de pila que guarda elementos, sin embargo, esto no solo es ineficiente sino que también resulta inviable usar varias pilas distintas en un mismo programa. Por lo tanto se evoluciona a una estructura de pila generalizada, capaz de tratar con cualquier tipo de dato, independiente a este. Finalmente, esta pila se usa de manera concisa, dándole una aplicación útil e interesante, como es realizar una DFS.

Los beneficios que nos ha aportado esta práctica han sido, por un lado experimentar de primera mano como la abstracción es capaz de simplificar y resolver problemas eficientemente y de manera modular, hemos aprendido a trabajar con punteros a funciones, hemos practicado la recursividad y, a mi parecer, lo más importante ha sido ver que para resolver un problema lo mejor es buscar aquella solución que aunque en principio parezca poco ortodoxa, sea más eficiente.

Las principales dificultades radicaron más que en el código, en la toma de decisiones: cómo implementamos la pila de enteros, cómo pasamos una función con este prototipo, ¿modificamos los archivos ya hechos? Etc. Estas cuestiones nos brindaron la oportunidad de recapacitar y adoptar una autonomía necesaria para llevar a cabo nuestro proyecto de la manera que consideramos mejor.

Lo más sencillo fue la implementación de las funciones de la pila, mientras que lo más complicado resultó ser tomar decisiones esenciales, que nos permitieron avanzar.

En conclusión, esta práctica ha resultado un gran avance para nuestro grupo y esperamos seguir evolucionando y aprendiendo más con cada una de las siguientes prácticas.