Junco de las Heras, Joaquin Murcia y Borja Velasco

Programación II - Grupo 210

PRÁCTICA Nº2: MEMORIA

1. EVALUACIÓN DE EXPRESIONES POSFIJO

El TAD pila diseñado en el ejercicio P2_E2 establece una pila de enteros. Para permitir la evaluación de expresiones posfijo debemos ampliar su implementación, haciendo uso de las siguientes funciones privadas (tanto su primitiva como su implementación pertenece a *stack_elestack.c*):

- **char caracter_leer (char *str, int i);**
  Recibe una cadena de caracteres y un entero como argumento de entrada. Retorna el caracter que corresponde a la posición i en la cadena de caracteres recibida.

- **Boolean caracter_esOperando (char c);**
  Recibe un carácter como parámetro de entrada. Retorna TRUE en caso de ser un operando, FALSE en caso opuesto.

- **Boolean caracter_esOperador (char c);**
  Recibe un carácter como parámetro de entrada. Retorna TRUE en caso de ser un operador, FALSE en caso opuesto.

- **int operarExpresion (int op1, int op2, char c);**
  Recibe dos operandos de tipo int y un char que indica la operación a realizar. Retorna el resultado de la operación.

Todo ello debe pertenecer a la función pública *status stack_evaluarPosfijo (char *str)*, cuya implementación se encuentra en *stack_elestack.c* y su primitiva en *stack_elestack.h*.

2. EXPLICACIÓN DE LAS DECISIONES DEL DISEÑO

Dividiremos las explicaciones en los distintos proyectos resueltos en la práctica, centrándonos principalmente en los aspectos que hemos tenido que modificar tras las entregas intermedias para dejar así el código completamente correcto:

- **P2_E1 y P2_E2**, destacamos en primer lugar la implementación tomada en la función *stack_ini*. En el proceso de diseño e implementación tomamos la decisión de no inicializar cada uno de los elementos de la pila a la hora de inicializarla. Reservamos memoria para los mismos a la hora de hacer un *stack_push*.

```
Stack * stack_ini() {
    Stack *s = NULL;
    int i;

    s = (Stack*) malloc(sizeof (Stack));
    if (s == NULL){
        fprintf(stderr, "%s", strerror(errno));
        return NULL;
    }

    for (i = 0; i < MAXSTACK; i++) {
        s->item[i] = NULL;
    }

    s->top = -1;
    return s;
}
```

Figura 1. Implementación *stack_ini* en stack.c

```
Status stack_push(Stack *s, const EleStack *e) {
    EleStack* copia = NULL;
    if (!s || !e) return ERROR;

    if (stack_isFull(s) == TRUE) return ERROR;

    copia = EleStack_copy(e);
    if (!copia) return ERROR;

    s->item[s->top + 1] = copia;
    s->top++;

    return OK;
}
```

Figura 2. Implementación *stack_push* en stack.c

Del mismo modo ocurre en los respectivos *elestack.c*, en el caso del *p2_e1.c* son nodos mientras que en *p2_e2.c* son enteros.

```c
/*Inicializa un EleStack*/
EleStack * EleStack_ini() {
    EleStack *es = NULL;

    es = (EleStack *) malloc(sizeof (EleStack));
    if (!es) {
        fprintf(stderr, "%s", strerror(errno));
        return NULL;
    }

    es->e = NULL;
    return es;
}
```

Figura 3. Ejemplo de implementación Ele*Stack_ini* en *elestack.c* de tipo int

```c
/*Es el setter*/
Status EleStack_setInfo(EleStack *es, void *p) {
    if (!es || !p)return ERROR;

    /*Hace free en caso de que haya algo guardado*/
    free(es->e);
    es->e = NULL;


    es->e = (int *) malloc(sizeof (int));
    if (!es->e) {
        fprintf(stderr, "%s", strerror(errno));
        return ERROR;
    }

    *(es->e) = *(int*) p;

    return OK;
}
```

Figura 4. Ejemplo de implementación *EleStack_setInfo* en *elestack.c* de tipo int

- **P2_E3**, el contenido didáctico principal de esta práctica es la generalización de la implementación del TAD Pila, con el objetivo de poder utilizar pilas con diferentes tipos de elementos en un mismo programa. Encontramos tres funciones dependientes del tiipo de elemento. De estas tres pasamos sus punteros a la estructura de la pila, quedando esta así:

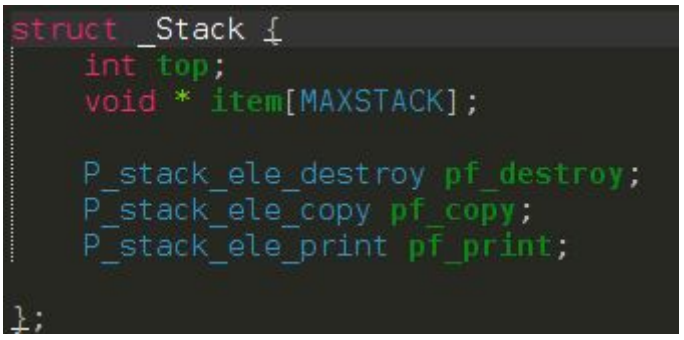

Figura 5. Estructura del TAD Pila generalizado para poder ser utilizado con distintos tipos de elementos sin cambiar su implementación

Nuestra principal decisión en este ejercicio (además de la señalada arriba) ha sido cambiar el archivo node.c y emplear punteros a void, para evitar un conflicto de tipos entre stack_fp y node.

Una de las cosas que más nos llamó la atención y de la que tardamos en percatarnos fue que, al tener que incluir las funciones antes indicadas en la EdD, debíamos modificar algunas cosas. Dichas funciones, a pesar de encontrarse en el .c del elemento concreto, y no de la pila, tenían que utilizar argumentos genéricos (es decir, punteros a void en lugar de, por ejemplo, a node). Lo ilustramos con un ejemplo:

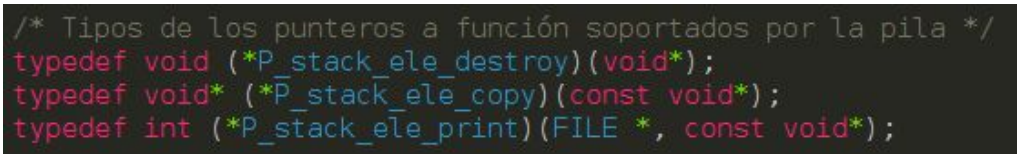

Figura 6. Definición de tipos de las funciones que se incluyen en la EdD del TAD Pila.

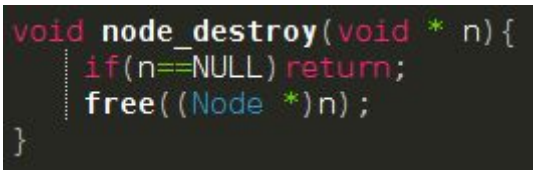

Figura 7. Implementación modificada de la función node_destroy

Como vemos, el tipo *P_stack_ele_destroy recibe como argumento un puntero a void, mientras que la función original node_destroy recibía uno a Node. En consecuencia, y para poder generalizar el TAD Pila, se debe tener esto en cuenta al implementar estas funciones en cualquier elemento que queramos utilizar en conjunto con el TAD Pila.

En este caso, es tan simple como modificar el argumento y hacer un casting a Node a la hora de utilizar la función free.

La repetición de los ejercicios 1 y 2 con este formato no nos supuso gran complicación.

La única decisión relevante fue la de diseñar las funciones destroy, copy y print par el tipo de dato int en el mismo main, y no en un .c y .h separado, resultándonos indiferente una opción de la otra por no tener mayor complicación ni efectos positivos ni negativos en la práctica.

Destacar el conocimiento, ahora, de la función atoi, que nos permite transformar una cadena de caracteres en entero. Esto fue utilizado para obtener un entero de la cadena pasada como argumento mediante argv.x

- **P2_E4**, En la función del DFS lo que hacemos es una pila de ids, los que utilizamos para poder acceder en la matriz de adyacencia y un array de colores, es decir, para saber si un id (AKA nodo) ha sido visitado o no.

  El algoritmo es básicamente el que nos dan en pseudocódigo, meto el id del nodo en la pila y luego mientras haya algo en la pila mete a todos los nodos no visitados de los que se puede acceder desde ese nodo y que no hayan sido visitados aún a la pila hasta encontrar al nodo que queremos buscar (en caso contrario devolvemos NULL).

```
while(stack_isEmpty(s)==FALSE){
    u = stack_pop(s);

    if(color[*u] == BLANCO){
        color[*u] = NEGRO;

        for(i=0;i<numNodes;i++){
            /*Si no es un vecino continua el bucle*/
            if(g->connections[*u][i] == FALSE)continue;

    /*Si es vecino y es el id buscado acaba, devolviendo el nodo*/
            if(i==destinoId){
                free(color);
                stack_destroy(s);
                intDestroy(u);

                return graph_getNode(g, to_id);
            }

    /*No es el nodo buscado, si no se ha visitado (Etiqueta BLANCA)
    * se mete en la pila */
            if(color[i] == BLANCO){
                if(stack_push(s, &i) == ERROR){
                    stack_destroy(s);
                    free(color);
                    return NULL;
                }
            }
        }


    }

    intDestroy(u);
```

Figura 7. Implementación del algoritmo de búsqueda en profundidad.

La función *graph_printCaminoDFS* es una función que prepara el array y las variables para hacer la función recursiva. También llama a la función de si hay camino (si no hay camino debe acabar y no mostrar nada).

```c
void graph_printCaminoDFS (FILE *f, Graph *g, int from_id, int to_id){
    int origenId = find_node_index(g, from_id), destinoId = find_node_index(g, to_id);
    int numNodos = graph_getNumberOfNodes(g), *color = NULL, i;
    Node *n = NULL;

    if(origenId<0||origenId>=numNodos||destinoId<0||destinoId>=numNodos)return;

    /*Comprueba si hay conectividad (si no lo hay n sera NULL y la funcion acabara)*/
    n=graph_findDeepSearch(g, from_id, to_id);
    if(!n){
        fprintf(f, "NO HAY CONECTIVIDAD");
        return;
    }

    node_destroy(n);

    color=(int *)malloc(numNodos*sizeof(int));
    if(!color){
        fprintf(stderr, "%s", strerror(errno));
        return;
    }

    /*Inicializa el array de visitados*/
    for(i=0;i<numNodos;i++){
        color[i]=BLANCO;
    }

    /*Para mostrar el camino de izq a derecha correctamente lo que hace es un DFS del destino hasta el origen.
     Como es un grafo dirigido, en vez de mirar la conectividad con connect[i][j] buscara connect[j][i]*/
    camino_recursivo(f, g, destinoId, origenId, color);
    node_print(f, g->nodes[destinoId]);

    free(color);
```

Figura 8. Código de *graph_printCaminoDFS*.

En la función recursiva, para poder mostrar el camino del derecho, lo que se hace es una especie de DFS desde el nodo del destino hasta el nodo origen. Una vez encontrado el nodo origen las funciones recursivas deben acabar (se sabe si una función tiene que acabar porque devuelve la anterior TRUE, si la anterior llamada devuelve FALSE debe seguir explorando). Mientras vuelve (True) la función va imprimiendo los nodos.

```c
Bool camino_recursivo(FILE *f, Graph *g, int u, int destinoId, int *color){
    int i, numNodos = graph_getNumberOfNodes(g);

    /*Si ya se ha visitado volver*/
    if(color[u] == NEGRO)return FALSE;
    color[u]=NEGRO;

    for(i=0;i<numNodos;i++){
        /*Mira conexiones de i a u porque estamos yendo en sentido contrario, a ver si existen
         conexiones de un i al nodo u en el que estamos*/
        if(g->connections[i][u] == FALSE)continue;

    /*Si es el destino lo imprime y acaba las llamadas*/
        if(i==destinoId){
            node_print(f, g->nodes[i]);
            return TRUE;
        }

    /*Recursion, si encuentra alguna de esas llamadas al id entonces la funcion devuelve TRUE para
        * acabar*/
        if(camino_recursivo(f, g, i, destinoId, color) == TRUE){
            node_print(f, g->nodes[i]);
            return TRUE;
        }
    }

    return FALSE;
}
```

Figura 9. Código de la función recursiva..

Por último, remarcar la importancia de la función *mainDestroy*, implementada en *p2_e1.c*, *p2_e2.c*, *p2_e3_Node.c*, *p2_e3_Int.c* y *p2_e4.c*. Una llamada a dicha función destruye los recursos de memoria dinámica empleados en el main. Sin embargo, debemos ser cautelosos a la hora de programar e ir inicializando nuestros punteros a NULL al comienzo del código y después de hacer destroys parciales (a mitad de código, por ejemplo).

```
int mainDestroy(int FLAG, FILE *f, Graph *g, Node *n){
    fclose(f);
    graph_destroy(g);
    node_destroy(n);

    return FLAG;
}
```

Figura 10. Ejemplo de implementación de mainDestroy en *p2_e4.c*

3. CONCLUSIONES FINALES

La práctica nº 2 ha supuesto un crecimiento abismal en términos de conocimiento y madurez. Frente al agobio y la presión generada por la práctica nº 1, hemos sido capaces de afrontar la práctica nº 2 con mayor sosiego.

El trabajo con mayor constancia ha marcado una seria diferencia. Tratamos siempre de participar de forma conjunta en los distintos proyectos integrados en la práctica, lo cual ha permitido una mayor tranquilidad en el grupo, permitiéndonos profundizar en mayor medida en los contenidos de la asignatura empleados (TAD Pila y sus diversas implementaciones).

El P2_E4 fue el único problema que presentó mayor dificultad, en especial la creación de la función recursiva para mostrar el camino del DFS. Sin embargo, logramos llegar a una implementación que simplificó mucho el algoritmo, permitiendo finalmente resolverlo como explicamos en el apartado de explicación de las decisiones del diseño.

En conclusión, gracias a la práctica nº 2 hemos crecido como programadores y como grupos. Hemos adquirido mayor fluidez a la hora de plantear los proyectos e implementarlos y en la administración del tiempo de trabajo, que nos permite llegar a la futura práctica nº 3 con una mayor confianza y motivación.