Programación II, 2016-2017

Escuela Politécnica Superior, UAM

Práctica 3: TAD Cola y TAD

PARTE 2A: PREGUNTAS SOBRE LA PRÁCTICA

Ejercicio #1:

Suponed que en lugar de la implementación del TAD Pila elaborado para la práctica 2 con memoria estática, se desea hacer una implementación basada en listas. Indicad cómo influye el uso de esta nueva implementación desde los puntos de vista sintáctico, semántico y uso de memoria:

Desde el punto de vista del uso de memoria, en la Pila de la práctica 2 teníamos un array de punteros a void de tamaño MAXSTACK, por lo que el tamaño máximo de esa pila era MAXSTACK, mientras que en una pila implementada con esta lista tendría tamaño ilimitado. Además, se usaría solo la memoria necesaria, ya que en la pila de la práctica 2 teníamos MAXSTACK punteros a void, y no todos ellos estaban siendo utilizados, lo que es un malgasto de memoria.

Ejercicio #2:

Suponed que se disponen de dos implementaciones diferentes del TAD lista (no ordenada). La primera de ellas es la que se ha elaborado en la presente práctica con memoria dinámica. La segunda es una implementación con memoria estática, donde se emplea un array para almacenar las direcciones a los elementos que contienen la lista, y una referencia (puntero) a la posición de cabeza de la misma. Si se desea eliminar el elemento que se encuentra justo en la mitad de la lista, indique los pasos que debe realizarse en cada caso y calcule el número de accesos a memoria en cada uno de los mismos. Si esta operación tuviera que repetirse con mucha asiduidad, ¿por cuál de las dos implementaciones optaría?

Optaríamos por la implementación con memoria dinámica, ya que, aunque haya que recorrer media lista, con sus correspondientes accesos a memoria, la acción de

eliminar el elemento deseado se limita a cambiar unos pocos punteros, mientras que en la implementación del array, una vez eliminado el elemento del medio, habría que mover todas las direcciones de memoria posteriores al elemento eliminado una "casilla" del array hacia la izquierda, lo cual sería muy costoso.

Ejercicio #3:

Basándote en el análisis anterior, discute las diferencias (incluyendo ventajas y desventajas) de la definición del TAD Grafo de la P1 frente a las dos propuestas indicadas en el último ejercicio, el P3_E4, independientemente de que hayas implementado alguna de estas propuestas en la práctica.

Respecto a las ventajas, vemos una ventaja similar a la descrita en las preguntas 1 y 2, ya que tenemos un array estático (nodos limitados) y si quisiéramos eliminar un nodo tendríamos que mover todos los que estuvieran por encima de él hacia la izquierda una posición. También, la matriz de adyacencia (MAX_NODES x MAX_NODES enteros) se ve optimizada con listas conteniendo los nodos a los que llegan los enlaces salientes y nodos de los que llegan enlaces de cada nodo del grafo. Además, el manejo de la memoria es mucho mas eficiente en el caso del uso de memoria dinámica, pues no se desperdicia memoria.

Respecto a las desventajas, tenemos que definir más TADs (NodeConnections, List), en especial en la opción 2 al ser bastante abstracto es más complicado escribir y detectar errores en el código. Referenciar punteros es más costoso a la hora de escribir el código a simplemente manejar memoria estática.

PARTE 2B: MEMORIA SOBRE LA PRÁCTICA

1. Decisiones de diseño

Ejercicio #1:

La implementación del TAD Cola ha resultado ser la parte más sencilla de la práctica. En ella simplemente nos adaptamos a lo que nos pedía la especificación de la interfaz.

Las decisiones de diseño vinieron dadas por lo visto en las clases teóricas y por la práctica anterior, en la que creamos el TAD Pila general, al igual que hemos creado el TAD Cola aquí.

Ejercicio #2:

En este ejercicio se nos pedía implementar el ejercicio "p2_e3.c" pero con el TAD Cola en vez de con el TAD Pila. En este caso hemos implementado una versión del algoritmo de *Búsqueda en Anchura*, que es el que se realiza con una estructura de tipo *FIFO (First in first out)* como la cola.

Además, añadimos el campo ***int*** *fatherId;* en el TAD Nodo, así como la creación de primitivas para acceder a él para que el *backtracking* resultara más sencillo.

El pseudocódigo de la función que devuelve el camino (sin control de errores) está en la página siguiente.

u, v, src son nodos.
Q y Path son colas.
V[G] se refiere a los vertices de G

```
Queue graph_path (Graph G, int srcId, int destId)
{
      for u ∈ V[G] do
      {
            color[u] ← WHITE;
            father[u] ← NULL;
      }

      src ← graph_getNode(G, srcId);
      color[src] ← GREY;
      father[src] ← NULL;

      Q ← queue_ini();
      Q ← enqueue(src);

      while !empty(Q) do
      {
            u ← dequeue(Q);
            for v ∈ adjacency[u] do
            {
                  if color[v] = WHITE do
                  {
                        color[v] ← GREY;
                        father[v] ← u;
                        Q ← enqueue(v);
                  }
            }
            color[u] ← BLACK;

      }

      //Backtracking

      Path ← queue_ini();

      u ← graph_getNode(G, destId);
      Path ← enqueue(u);

      while father[u] ≠  0
      {
            u ← father[u];
            Path ← enqueue(u);
      }

      return Path;
}
```

PARTE 2B: MEMORIA SOBRE LA PRÁCTICA

Ejercicio #3:

Para la implementación del TAD Lista seguimos la especificación de cada función en la interfaz y las recomendaciones de la clase de teoría. Decidimos enumerar la lista a partir del 1, y no a partir del 0, como se ve reflejado en la funcion de list_get.

Implementamos la lista de forma general guardando en la estructura de lista las funciones con las que opera la misma. Además, tuvimos que definir una nueva estructura de datos llamada nodeList, para la que creamos la función de inicializar con el objetivo de ahorrar código.

Esta estructura se compone de un void*element y un nodeList* next; y serán los ítems que compongan la lista.

La mayor dificultad surgió en la función list_insertInOrder, cuyo pseudocódigo (sin control de errores) se adjunta en la página siguiente.

Por último, para realizar el main seguimos las instrucciones del PDF y fuimos leyendo de él y añadiendo los nodos uno a uno en la lista.

```
List list_insertInOrder(List list, const void* pElem)
 {
     u ← first[list];

     if u = NULL do
     {
         list ← insertFirst(pElem);
         return list;
     }

     if data[u] > pElem do
     {
         list ← insertFirst(pElem);
         return list;
     }

     v = next[u];

     while forever
     {
         if v = NULL do
         {
             aux ← nodeList_ini();
             data[aux] ← copy(pElem);
             next[aux] ← v;
             next[u] ← aux;
             return list;
         }

         if data[v] > pElem do
         {
             aux ← nodeList_ini();
             data[aux] ← copy(pElem);
             next[aux] ← v;
             next[u] ← aux;
             return list;
         }

         u ← v;
         v ← next[v];
     }
 }
```

PARTE 2B: MEMORIA SOBRE LA PRÁCTICA

Ejercicio #4:

Finalmente, en el cuarto ejercicio hemos implementado de nuevo graph.c, siguiendo las indicaciones de la práctica para crear un TAD Graph usando la estructura de datos de lista. Para ello hemos hecho uso de las primitivas de lista. El nuevo archivo fuente se llama graph-list.c.

Escogimos la opción B debido a que a nuestro parecer es más general y proporciona un mejor manejo de memoria al no haber arrays de memoria estática.

La interfaz de graph no se ha visto modificada debido a que sólo hemos de cambiar la implementación.

Para nuestra opción nos creamos una nueva estructura de datos llamada nodeConnetions, y se modificó la estructura de Graph.

```
struct _nodeConnections{
        int id;
        list connections;
};
```

```
struct _Graph {
    List * nodes;
    List* out_connections;
    List* in_connections;
    int num_nodes;
    int num_edges;
};
```

Por otro lado hemos tenido que crear las funciones para que la lista trabajase con la estructura nodeConnections. Por esta modificación nuestra función graph_print imprime una lista de adyacencia en vez de una matriz de adyacencia.

Las funciones se hallan implementadas en functions.c y declaradas en functions.h.

Finalmente, el main que usamos para probarlo es el mismo que el de p1_e4.c, es decir, graph_test.c, proporcionado por los profesores de PROG_II.

Por tanto, las únicas funciones que hemos tenido que modificar han sido aquellas que requerían el uso de memoria estática.

2. Informe del uso de memoria.

Se adjuntan los reportes de valgrind sin la salida del programa por motivos de extensión.

Ejercicio #1:

```
valgrind --leak-check=full ./p3_e1 "nodos.txt"
  Memcheck, a memory error detector
  Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
  Using Valgrind-3.12.0.SVN and LibVEX; rerun with -h for copyright info
  Command: ./p3_e1 nodos.txt

  HEAP SUMMARY:
      in use at exit: 0 bytes in 0 blocks
    total heap usage: 19 allocs, 19 frees, 11,976 bytes allocated

  All heap blocks were freed -- no leaks are possible

  For counts of detected and suppressed errors, rerun with: -v
  ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Ejercicio#2:

```
valgrind --leak-check=full ./p3_e2 "g1.txt" "1" "3"
  Memcheck, a memory error detector
  Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
  Using Valgrind-3.12.0.SVN and LibVEX; rerun with -h for copyright info
  Command: ./p3_e2 g1.txt 1 3

  HEAP SUMMARY:
      in use at exit: 0 bytes in 0 blocks
    total heap usage: 21 allocs, 21 frees, 67,151,048 bytes allocated

  All heap blocks were freed -- no leaks are possible

  For counts of detected and suppressed errors, rerun with: -v
  ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Se adjuntan los reportes de valgrind sin la salida del programa por motivos de extensión.

Ejercicio #3:

```
valgrind --leak-check=full ./p3_e3 "datos.txt"
  Memcheck, a memory error detector
  Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
  Using Valgrind-3.12.0.SVN and LibVEX; rerun with -h for copyright info
  Command: ./p3_e3 datos.txt

  HEAP SUMMARY:
      in use at exit: 0 bytes in 0 blocks
    total heap usage: 65 allocs, 65 frees, 6,280 bytes allocated

  All heap blocks were freed -- no leaks are possible

  For counts of detected and suppressed errors, rerun with: -v
  ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Ejercicio#4:

```
valgrind --leak-check=full ./p3_e4 "g1.txt"
  Memcheck, a memory error detector
  Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
  Using Valgrind-3.12.0.SVN and LibVEX; rerun with -h for copyright info
  Command: ./p3_e4 g1.txt
  HEAP SUMMARY:
      in use at exit: 0 bytes in 0 blocks
    total heap usage: 62 allocs, 62 frees, 7,628 bytes allocated

  All heap blocks were freed -- no leaks are possible

  For counts of detected and suppressed errors, rerun with: -v
  ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

3. 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 y la referenciación de punteros, muy presente en las clases teóricas.

La parte más fácil de la práctica puede ser la creación propia del TAD Cola y la más difícil sin duda la implementación del TAD Grafo con la estructura de lista tanto para las conexiones como para los nodos. Sin embargo, el trabajo habría sido bastante menor si hubiéramos elegido la opción A en el ejercicio 4. Por otro lado, la implementación de ciertas funciones del TAD Lista tuvieron un mayor nivel de complejidad en comparación con el TAD Cola.

Como aspectos nuevos de programación podemos recoger tanto el uso de un nuevo TAD como estructuras de datos, como la creación de funciones propias para una nueva estructura de datos.