**Memoria P3\_E1**

En el primer ejercicio no hemos tomado ninguna medida demasiado personal para la implementación del TAD Queue. La complejidad se basaba sobretodo en la aritmética modular necesaria a la hora de modificar los punteros head y end.

En la mayoría de las primitivas copiamos el puntero a void que se nos da: el original sigue siendo independiente de la cola y el usuario deberá encargarse de liberar la memoria correspondiente. El otro puntero a void pasa a formar parte de la cola y se liberará cuando se extraiga o, en su defecto, cuando se destruya la cola. Hemos optado por hacer esta copia porque en los posteriores ejercicios (e incluso en este mismo) nos será de gran utilidad.

**Memoria P3\_E2**

En este ejercicio hemos reciclado completamente la búsqueda de anchura que se hizo en la práctica anterior. Hemos cambiado todas las pilas por colas y hemos cambiado el nombre de las primitivas, de tal forma que ahora no utilizamos un algoritmo de búsqueda en anchura, sino de búsqueda en profundidad, ya que primero se explorarán los últimos nodos visitados. Básicamente no hemos hecho ningún otro cambio, únicamente hemos solucionado nuevos errores de memoria.

**Memoria P3\_E3**

A la hora de implementar la lista, destacamos el haber introducido en la estructura de la lista un entero que muestre el tamaño. Gracias a este entero la función list\_size tiene un coste O(1) y de otra forma sería de coste O(n). Por el resto, la solución el problema es muy estándar, al igual que el resto de la definición del TAD. Quizá se puede destacar la primitiva list\_insertInOrder, donde no se ha aprovechado ninguna otra primitiva como list\_extractLast o list\_extractFirst, y por lo tanto se ha conseguido una mayor optimización de la misma.

**Memoria P3\_E4**

Hemos optado por cambiar el grafo original por la opción A de las ofrecidas. Aunque no es tan óptima como la opción B, hemos considerado que lo más importante del problema es sustituir la matriz de adyacencia, ya que aumentaba cuadráticamente respecto al número de MAX\_NODES. También hemos decidido implementar un entero en la estructura, numNodes, para saber el número de nodos que hay en el grafo, ya que de nuevo facilita muchísimas cuentas.

Adicionalmente para este ejercicio nos hemos tomado la libertad de introducir una nueva función en el TAD List llamada list\_isElement(List \*l, void \*ele) la cual devuelve TRUE si el elemento pasado por referencia se encuentra en algún lugar de la lista (también pasada como argumento a la función), o FALSE en caso contrario. La misión de esta función es facilitarnos la implementación de la función del TAD Graph "graph\_areConnected" que nos comprueba si dos nodos están conectados.

**Informe del uso de memoria**

Lo primero que hay que destacar es que el Valgrind nos marca 0 errores de memoria en todos los ejercicios. Partiendo de ahí, ahora explicamos los errores solucionados. Gran parte de ellos eran del archivo principal, donde hemos tenido que revisar no muy exhaustivamente los pasos realizados. Después están los errores en primitivas, menos comunes pero también más complicados de vislumbrar.

**Conclusiones**

En conjunto con la práctica anterior, hemos visto pilas, colas y listas, y ahora podemos compararlas. Las listas tienen la ventaja de englobar colas y pilas, y además servir para otros fines distintos. Sin embargo, las pilas y colas nos ofrecen una mayor velocidad de respuesta, ya que ellas no necesitan recurrencia para llegar a ciertos elementos.

Hemos visto que las colas pueden servir para realizar el algoritmo de búsqueda en anchura, y podemos decir lo análogo de las pilas y la búsqueda en profundidad. Lo que podemos concluir es que la mayoría de problemas se pueden solucionar tanto con pilas como con colas, solamente que hay que utilizar una perspectiva distinta. Por supuesto, eso también implica que todo problema de colas o pilas se puede realizar con listas aunque, por lo dicho antes, es mejor no escoger listas si se puede.

Respecto al nuevo formato del grafo, concluimos que la memoria dinámica en muchos casos tiene más ventajas que inconvenientes, pero que cuando cada nuevo elemento insertado aumenta exponencialmente la memoria, la memoria dinámica no es recomendable, es imprescindible. De hecho, en nuestro caso hemos optado por no hacer una lista con los nodos del grafo (solo con las adyacencias), ya que no ahorramos tanta memoria y se hace mucho más fácil la escritura de las primitivas del TAD.