Universidad Abierta Interamericana

Campus Virtual UAIOnline

PROGRAMACIÓN I

LECTURA

UNIDAD 2

ESTRUCTURAS DINÁMICAS 1

LISTAS

Autor de contenidos:
Nicolás Battaglia

PRESENTACIÓN

En Programación Estructurada usted ha estudiado las estructuras de memoria de tamaño estático. Esos conocimientos son fundamentales para el estudio propuesto en esta asignatura ya que a partir de ellos, avanzará en el conocimiento inicial y uso básico de las **estructuras dinámicas de memoria** (profundizará su estudio en Programación II).

Como su nombre lo indica, estas estructuras son de tamaño dinámico. Algunas permiten colocar dentro de ellas datos de distintos tipos y facilitan, en general, su manejo dentro de un programa.

Las conocemos como **cola**, **lista**, **pila**, **árbol** y **arrays** de registro en sus distintas formas. Veremos en esta unidad cada una de las estructura por separado y deberá realizar dos Trabajos Prácticos para ejercitar su uso.

Recuerde que al finalizar esta Unidad deberá realizar la Propuesta de Integración del Módulo I en la que se evaluará su competencia para **optimizar el manejo de la información en la memoria.** Esta propuesta constituye su Primera Evaluación Parcial de la asignatura cuya realización es obligatoria. Consulte en el Cronograma de la Asignatura las fechas previstas para su realización y entrega.

Acorde con lo expresado hasta aquí, esperamos que usted a través del estudio de esta unidad, adquiera capacidad para:

- Generar una estructura dinámica de memoria.
- Recorrer una estructura.
- Modificarla insertando o eliminando un nodo.

Comprobar el estado de capacidad de la estructura (llena o vacía o cuántas posiciones están ocupadas).

CARACTERÍSTICAS

En Ciencias de la Computación, una **lista enlazada** es una de las estructuras de datos fundamentales, y puede ser usada para implementar otras estructuras de datos. Consiste en una secuencia de nodos, en los que se guardan campos de datos arbitrarios y una o dos referencias, enlaces o punteros al nodo anterior o posterior. El principal beneficio de las listas enlazadas respecto a los vectores convencionales es que el orden de los elementos enlazados puede ser diferente al orden de almacenamiento en la memoria o el disco, permitiendo que el orden de recorrido de la lista sea diferente al de almacenamiento.

Una lista enlazada es un tipo de dato autorreferenciado porque contienen un puntero o enlace (en inglés *link*, del mismo significado) a otro dato del mismo tipo. Las listas enlazadas permiten inserciones y eliminación de nodos en cualquier punto de la lista en tiempo constante (suponiendo que dicho punto está previamente identificado o localizado), pero no permiten un acceso aleatorio. Existen diferentes tipos de listas enlazadas: listas enlazadas simples, listas doblemente enlazadas, listas enlazadas circulares y listas enlazadas doblemente circulares.

Tipos de listas enlazadas

Listas simples enlazadas

La lista enlazada básica es la **lista enlazada simple** la cual tiene un enlace por nodo. Este enlace apunta al siguiente nodo (o indica que tiene la dirección en memoria del siguiente nodo) en la lista, o al valor NULL o a la lista vacía, si es el último nodo.

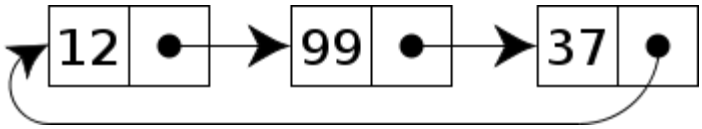
Listas doblemente enlazadas

Un tipo de lista enlazada más sofisticado es la **lista doblemente enlazada** o **lista enlazadas de dos vías**. Cada nodo tiene dos enlaces: uno apunta al nodo anterior, o apunta al valor NULL si es el primer nodo; y otro que apunta al nodo siguiente, o apunta al valor NULL si es el último nodo.

En algún lenguaje de muy bajo nivel, XOR-Linking ofrece una vía para implementar listas doblemente enlazadas, usando una sola palabra para ambos enlaces, aunque esta técnica no se suele utilizar.

Listas enlazadas circulares

En una lista enlazada circular, el primer y el último nodo están unidos juntos. Esto se puede hacer tanto para listas enlazadas simples como para las doblemente enlazadas. Para recorrer una lista enlazada circular podemos empezar por cualquier nodo y seguir la lista en cualquier dirección hasta que se regrese hasta el nodo original. Desde otro punto de vista, las listas enlazadas circulares pueden ser vistas como listas sin comienzo ni fin. Este tipo de listas es el más usado para dirigir buffers para “ingerir” datos, y para visitar todos los nodos de una lista a partir de uno dado.



Una lista enlazada circular que contiene tres valores enteros

Listas enlazadas simples circulares

Cada nodo tiene un enlace, similar al de las *listas enlazadas simples*, excepto que el siguiente nodo del último apunta al primero. Como en una lista enlazada simple, los nuevos nodos pueden ser solo eficientemente insertados después de uno que ya tengamos referenciado. Por esta razón, es usual quedarse con una referencia solamente al último elemento en una lista enlazada circular simple, esto nos permite rápidas inserciones al principio, y también permite accesos al primer nodo desde el puntero del último nodo.

Listas enlazadas doblemente circulares

En una lista enlazada doblemente circular, cada nodo tiene dos enlaces, similares a los de la lista doblemente enlazada, excepto que el enlace anterior del primer nodo apunta al último y el enlace siguiente del último nodo, apunta al primero. Como en

una lista doblemente enlazada, las inserciones y eliminaciones pueden ser hechas desde cualquier punto con acceso a algún nodo cercano. Aunque estructuralmente una lista circular doblemente enlazada no tiene ni principio ni fin, un puntero de acceso externo puede establecer el nodo apuntado que está en la cabeza o al nodo cola, y así mantener el orden tan bien como en una lista doblemente enlazada.

Ventajas

Como muchas opciones en programación y desarrollo, no existe un único método correcto para resolver un problema. Una estructura de lista enlazada puede trabajar bien en un caso pero causar problemas en otros. He aquí una lista con algunas de las ventajas más comunes que implican las estructuras de tipo lista. En general, teniendo una colección dinámica donde los elementos están siendo añadidos y eliminados frecuentemente e importa la localización de los nuevos elementos introducidos se incrementa el beneficio de las listas enlazadas.

Listas enlazadas vs. vectores o matrices

Las listas enlazadas poseen muchas ventajas sobre los vectores. Los elementos se pueden insertar en una lista indefinidamente mientras que un vector tarde o temprano se llenará o necesitará ser redimensionado, una costosa operación que incluso puede no ser posible si la memoria se encuentra fragmentada.

En algunos casos se pueden lograr ahorros de memoria almacenando la misma 'cola' de elementos entre dos o más listas – es decir, la lista acaba en la misma secuencia de elementos. De este modo, uno puede añadir nuevos elementos al frente de la lista manteniendo una referencia tanto al nuevo como a los viejos elementos - un ejemplo simple de una estructura de datos persistente.

Por otra parte, los vectores permiten acceso aleatorio mientras que las **listas enlazadas** sólo permiten acceso secuencial a los elementos. Las listas enlazadas simples, de hecho, solo pueden ser recorridas en una dirección. Esto hace que las listas sean inadecuadas para aquellos casos en los que es útil buscar un elemento por su índice rápidamente, como el heapsort. El acceso secuencial en los vectores también es más rápido que en las listas enlazadas.

Otra desventaja de las listas enlazadas es el almacenamiento extra necesario para las referencias, que a menudos las hacen poco prácticas para listas de pequeños datos como caracteres o valores booleanos.

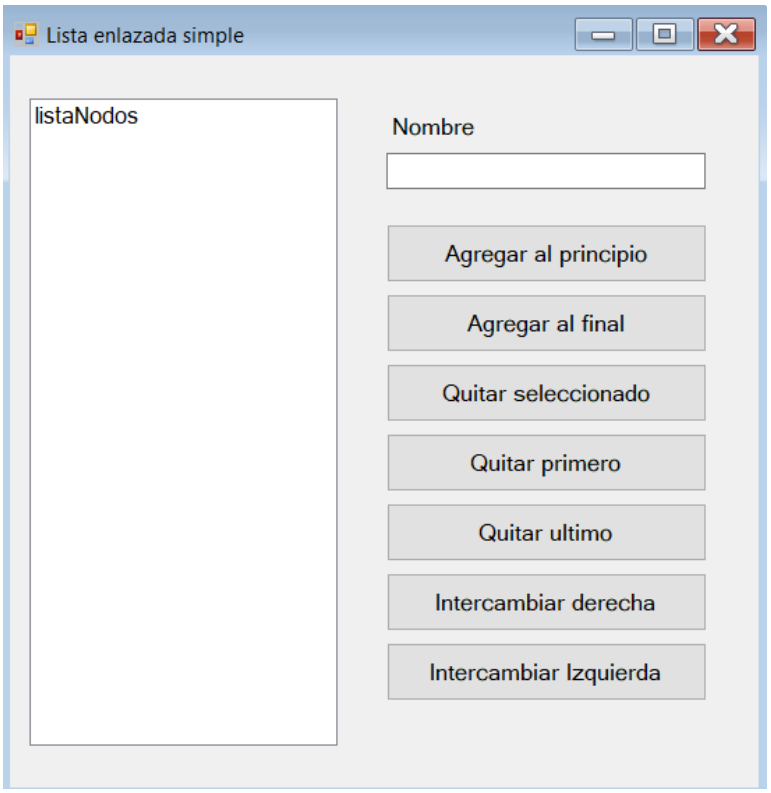
También puede resultar lento y abusivo el asignar memoria para cada nuevo elemento. Existe una variedad de listas enlazadas que contemplan los problemas anteriores para resolver los mismos. Un buen ejemplo que muestra los pros y contras del uso de vectores sobre listas enlazadas es la implementación de un programa que resuelva el problema de Josephus. Este problema consiste en un grupo de personas dispuestas en forma de círculo. Se empieza a partir de una persona predeterminadas y se cuenta *n* veces, la persona n-ésima se saca del círculo y se vuelve a cerrar el grupo. Este proceso se repite hasta que queda una sola persona, que es la que gana. Este ejemplo muestra las fuerzas y debilidades de las listas enlazadas frente a los vectores, ya que viendo a la gente como nodos conectados entre sí en una lista circular se observa como es más fácil suprimir estos nodos. Sin embargo, se ve como la lista perderá utilidad cuando haya que encontrar a la siguiente persona a borrar. Por otro lado, en un vector el suprimir los nodos será costoso ya que no se puede quitar un elemento sin reorganizar el resto. Pero en la búsqueda de la n-ésima persona tan sólo basta con indicar el índice n para acceder a él resultando mucho más eficiente.

Doblemente enlazadas vs. simplemente enlazadas

Las listas doblemente enlazadas requieren más espacio por nodo y sus operaciones básicas resultan más costosas pero ofrecen una mayor facilidad para manipular ya que permiten el acceso secuencial a lista en ambas direcciones. En particular, uno puede insertar o bo rrar un nodo en un número fijo de operaciones dando únicamente la dirección de dicho nodo (Las listas simples requieren la dirección del nodo anterior para insertar o suprimir correctamente). Algunos algoritmos requieren el acceso en ambas direcciones.

EJEMPLO

En el ejemplo Listas.rar, tendrá acceso a un ejemplo de listas enlazadas simples, desarrollado en C#. La interfaz de usuario es la siguiente:



En el ejemplo, se han creado dos tipos abstractos de datos en representación de la lista enlazada simple y en representación de los nodos que la componen.

El TDA del nodo, se llama NodoSimple y tiene la siguiente estructura:

```
public class NodoSimple
    {
        public int Numero;
        public string Nombre;
        public NodoSimple Siguiente;


        public override string ToString()
        {
            return string.Format("{0} {1}", Numero, Nombre);
        }
    }
```

En la misma vemos dos atributos que representan el nombre del nodo y un numero que lo identificará. Además, vemos también un atributo Siguiente, que representa el siguiente nodo en la lista.

El TDA de la lista en sí misma, se llama ListaEnlazadaSimpley tiene la siguiente estructura:

```
class ListaEnlazadaSimple
    {

        public NodoSimple NodoInicial = null;
        public void AgregarAlPrincipio(string nombre)
        {
            NodoSimple nodo = new NodoSimple();
            nodo.Numero = ProximoNumero();
            nodo.Nombre = nombre;

            if (NodoInicial == null)
                NodoInicial = nodo;
            else
            {
                //si hay elementos en la lista, tenemos que agregarlo entre el
inicial y el siguiente

                NodoSimple aux = NodoInicial;
                NodoInicial = nodo;
                NodoInicial.Siguiente = aux;

            }

        }
        public void AgregarAlFinal(string nombre)
        {
            NodoSimple nodo = new NodoSimple();
            nodo.Numero = ProximoNumero();
            nodo.Nombre = nombre;

            //necesito buscar el último para agregarlo al final
            NodoSimple ultimo = BuscarUltimo(NodoInicial);
            ultimo.Siguiente= nodo;
        }
        private int BuscarMaximo(NodoSimple nodo, int numero)
        {
            int max = nodo.Numero > numero ? nodo.Numero : numero;
            if (nodo.Siguiente!=null) //no es el ultimo
            {

                return BuscarMaximo(nodo.Siguiente, max);

            }
            else
            {
                return max;
            }


        }
        private int ProximoNumero()
        {
```

```
        if (NodoInicial == null) return 1;
        int max= BuscarMaximo(NodoInicial, NodoInicial.Numero);

        //busco el maximo y le sumo uno

            return max + 1;

    }
    private NodoSimple BuscarUltimo(NodoSimple nodo)
    {
        //la lista este vacia
        if (nodo == null) return null;

        //que no sea el ultimo
        if (nodo.Siguiente != null)
            return BuscarUltimo(nodo.Siguiente);
        else
            return nodo;

    }
    public void QuitarPrimero()
    {
        //par aquitar el primero, necesito usar una variable temporal
        //la lista este vacia
        if (NodoInicial != null)
        {
            NodoInicial = NodoInicial.Siguiente;
        }
    }
    public void QuitarUltimo()
    {
        //necesitamos buscar el anteultimo
        NodoSimple anteultimo = BuscarAnteultimo(NodoInicial);


        if (anteultimo != null)
           anteultimo.Siguiente = null;
        else
            NodoInicial = null;

    }
    private NodoSimple BuscarAnteultimo(NodoSimple nodo)
    {
        if (nodo == null) return null; //lista vacia



        if (nodo.Siguiente == null) return null; //hay un solo elemento

        if (nodo.Siguiente.Siguiente != null)
                return BuscarAnteultimo(nodo.Siguiente);
            else
                return nodo;

    }
```

```
        private NodoSimple BuscarAnterior(NodoSimple nodo, int numero)
        {


            if (nodo.Siguiente != null && nodo.Siguiente.Numero == numero)
                return nodo;
            if (nodo.Siguiente!=null)
                return BuscarAnterior(nodo.Siguiente, numero);
            return null; //es el ultimo...


        }
        public void QuitarPosicion(int numero)
        {
            if (NodoInicial!=null)
            {
                //si el primero es el que se quiere borrar
                if (NodoInicial.Numero == numero)
                {
                    QuitarPrimero();
                }
                else
                {
                    // si el ultimo es elq ue se quiere borrar
                    NodoSimple ultimo = BuscarUltimo(NodoInicial);
                    if (ultimo != null && ultimo.Numero == numero)
                        QuitarUltimo();
                    else
                    { //si se quiere borrar un nodo intermedio
                        NodoSimple nodoAnteriorAlElegido =
BuscarAnterior(NodoInicial, numero);
                        if (nodoAnteriorAlElegido != null)
                            nodoAnteriorAlElegido.Siguiente =
nodoAnteriorAlElegido.Siguiente.Siguiente;
                    }
                }
            }


        }
        public void IntercambiarDerecha(int numero)
        {

        }
        public void IntercambiarIzquierda(int numero)
        {

        }
        public void Intercambiar(int numero1, int numero2)
        {

        }
    }
```

Como vemos, además de tener una referencia al nodo inicial, tiene todas las funciones que podría tener una lista de este tipo, las cuales se resumen a continuación:

- Agregar nodo al principio (AgregarAlPrincipio)
- Agregar nodo al final (AgregarAlFinal)
- Buscar nodo con el número más alto (BuscarMaximo)
- Buscar el próximo número disponible (ProximoNumero)
- Buscar el último nodo de la lista (BuscarUltimo)
    - Se considera el último, aquel nodo que si siguiente es NULL
- Quitar el primer nodo de la lista (QuitarPrimero)
- Quitar el último nodo de la lista (QuitarUltimo)
- Buscar el anteúltimo nodo de la lista (BuscarAnteultimo)
- Obtener el nodo predecesor de un nodo determinado (BuscarAnterior)
- Quitar un nodo de una posición determinada (QuitarPosicion)
- Intercambiar un nodo determinado con el de su derecha (IntercambiarDerecha)
- Intercambiar un nodo determinado con el de su izquierda (IntercambiarIzquierda)
- Intercambiar un nodo con otro nodo a partir de los números que los identifican (IntercambiarIzquierda)

EN EL EJEMPLO, LOS ÚLTIMOS 3 ITEMS NO ESTÁN IMPLEMENTADOS, SE ESPERA QUE LOS PUEDA COMPLETAR EN EL TP DE LA UNIDAD.