UNIVERSIDAD NACIONAL DEL LITORAL
FACULTAD DE INGENIERÍA Y CIENCIAS HÍDRICAS

Tecnicatura en diseño y programación de videojuegos

UNLVIRTUAL

Programación de videojuegos I

Unidad 2
Manejo de listas enlazadas, pilas y colas

Docentes
Germán Gaona
Mauro Walter

CONTENIDOS

INTRODUCCIÓN

Las listas enlazadas forman parte del grupo de estructuras de datos dinámicas, lo cual significa que es posible agregar elementos a medida que se necesitan. Sin embargo, esto conlleva un problema, pues las listas se pueden recorrer únicamente de manera secuencial. Una *lista enlazada* es un conjunto de elementos llamados *nodos*, que contienen un dato y a su vez un enlace a otro nodo.

Una **lista enlazada** es un conjunto de elementos llamados nodos, que contienen un dato y a su vez un enlace a otro nodo. Puede ser simple o abierta, o circular.

Otro punto importante en esta unidad es el concepto de *pilas* y *colas,* que forman parte del grupo de estructuras de datos dinámicas junto con las listas enlazadas.

Una *pila* es un tipo especial de lista lineal en la cual un elemento sólo puede ser añadido o eliminado por un extremo, y contiene dos operaciones básicas: empujar y sacar.

Una **pila** es un tipo especial de lista lineal en la cual un elemento sólo puede ser añadido o eliminado por un extremo, y contiene dos operaciones básicas: empujar y sacar.

Una *cola* también es un tipo especial de lista lineal, pero su diferencia con las pilas radica en que la eliminación de elementos se realiza solamente al principio de la lista y la inserción, al final de la misma. Sus operaciones básicas son insertar y leer.

Una **cola** también es un tipo especial de lista lineal, pero su diferencia con las pilas radica en que la eliminación de elementos se realiza solamente al principio de la lista y la inserción, al final de la misma. Sus operaciones básicas son insertar y leer.

Por lo expuesto, es necesario inclinar nuestro interés primario a conocer qué tipo de estructura de datos requeriremos en una determinada situación.

Los arreglos ofrecen acceso rápido a un elemento mediante el uso de índices, y si están ordenados, también se pueden encontrar los valores buscados de manera más eficiente. Su desventaja reside en que son estáticos, es decir, que una vez creados mantienen su longitud, por lo que debemos prever el crecimiento de los datos. Además, la inserción y eliminación de elementos requiere el movimiento de todos los elementos posteriores a la posición a actualizar, lo que resulta en una pérdida de tiempo.

Por su parte, las listas enlazadas, las pilas y las colas son ineficientes en las búsquedas de elementos, ya que la única forma de hacerlo es la secuencial, pero son una buena opción si lo importante es la actualización permanente de los datos, pues –como veremos más adelante– se requiere sólo una reasignación de punteros.

2.1. Lista enlazada simple o abierta

Es un tipo de lista en el que se tiene un enlace por nodo. La figura que presentamos a continuación pertenece a una lista enlazada simple. Al primer nodo se lo denomina *cabeza de lista* y su enlace es guardado al crear la misma; al último nodo se lo llama *pie* y su enlace es NULL, es decir, es el único nodo de la lista que no apunta a otro.

La lista puede tener una longitud de cero, uno o más elementos. En el primer caso, la cabeza de lista es NULL; en el segundo caso, el primer nodo también coincide con el último, y en el tercer caso, el primer y el último nodo difieren, como sucede generalmente.

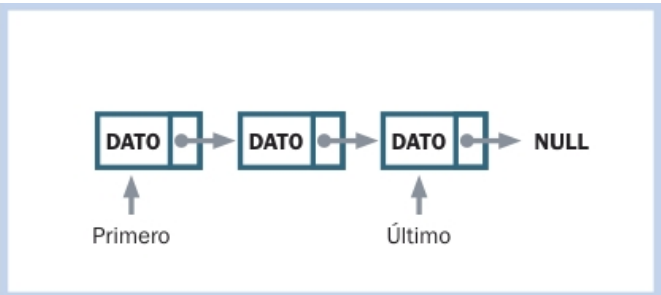


Fig. 1

A continuación, podemos apreciar un ejemplo en C++ de la estructura de un nodo y una lista básica.

```
class nodo {
   public:
    nodo(int v, nodo *sig = NULL)
```

```
    {
       valor = v;
       siguiente = sig;
    }

   private:
    int valor;
    nodo *siguiente;

   friend class lista;
};
typedef nodo *pnodo;

class lista {
   public:
    lista() { primero = actual = NULL; }
    ~lista();

// Por claridad, se omite declaración de métodos

   private:
    pnodo primero;
    pnodo actual;
};
```

2.1.1. Operaciones básicas

Como en cualquier colección de elementos, las operaciones básicas de una lista simplemente enlazada consisten en búsqueda, inserción, modificación y eliminación de un nodo.

Búsqueda

En una lista enlazada, las búsquedas son únicamente secuenciales y en un solo sentido. Para empezar a recorrer una lista debemos contar con el primer elemento y, a partir de este, ir tomando los enlaces hasta que llegar al nodo buscado o al nodo final. La única forma de retroceder en este tipo de listas es empezando desde la cabeza nuevamente.

El siguiente código muestra un ejemplo donde se definen los métodos de clase *`lista`*, utilizados para moverse dentro de la lista enlazada:

```
void lista::Siguiente() {
   if(actual) actual = actual->siguiente;
}

void lista::Primero() {
   actual = primero;
}

void lista::Ultimo() {
   actual = primero;
   if(!ListaVacia())
      while(actual->siguiente) Siguiente();
}
```

Inserción

Las listas presentan diferentes casos de inserción, de acuerdo a la posición donde se quiera ubicar el elemento y al estado de la lista. Para todos los casos, suponemos que contamos con el nodo creado.

Inserción al inicio de la lista

1. Asignamos el valor de la cabeza de la lista como el siguiente en el nuevo nodo (o NULL en caso de que la lista este vacía).

2. Asignamos a la cabeza de la lista la referencia al nuevo nodo.

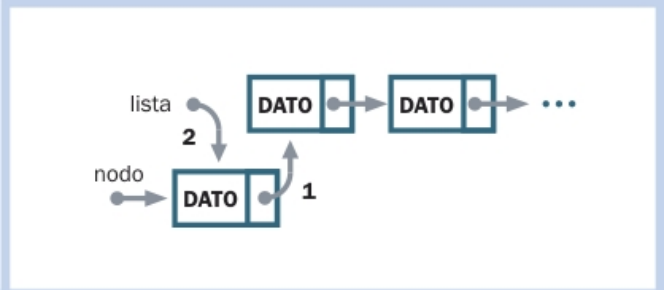


Fig. 2

Inserción entre dos nodos (o al final de la lista)

Contaremos aquí con un nodo nuevo, uno anterior y uno posterior. En este caso, el primero se colocará entre los dos últimos, respectivamente.

1. Asignamos la referencia al nodo posterior como el siguiente en el nuevo nodo (o NULL en caso de que sea el último nodo).

2. Asignamos la referencia al nuevo nodo como el siguiente en el nodo anterior.

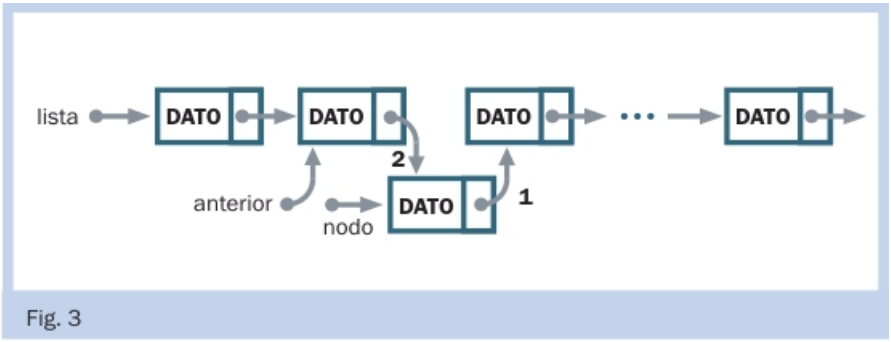


Fig. 3

El siguiente código muestra un ejemplo donde se define el método *Insertar*. Éste cubre ambos casos de inserción, descriptos anteriormente:

```
void lista::Insertar(int v){
   pnodo anterior;

   // Si la lista está vacía
   if(ListaVacia() || primero->valor > v) {
      // Asignamos a lista un nievo nodo de valor v y
      // cuyo siguiente elemento es la lista actual
      primero = new nodo(v, primero);
   }
   else {
      // Buscar el nodo de valor menor a v
      anterior = primero;

      //  Avanzamos  hasta  el  último  elemento  o  hasta  que  el
siguiente tenga
      // un valor mayor que v
      while(anterior->siguiente && anterior->siguiente->valor <= v)
         anterior = anterior->siguiente;

      // Creamos un nuevo nodo después del nodo anterior, y cuyo
siguiente
      // es el siguiente del anterior
      anterior->siguiente = new nodo(v, anterior->siguiente);
   }
}
```

Eliminación

Al igual que la inserción, la eliminación de nodos en una lista presenta casos especiales. Aquí, sin embargo, deben contemplarse también las situaciones en que se trate de una lista vacía o que quede vacía después de eliminar un nodo.

Eliminación del primer elemento de la lista

1. Conservamos el puntero al nodo que queremos borrar.
2. Asignamos la referencia al siguiente nodo a la cabeza (o NULL en caso de que la lista quede vacía).
3. Liberamos la memoria del nodo borrado.

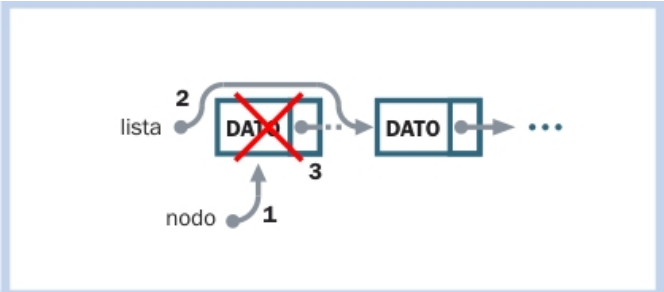


Fig. 4

Eliminación entre dos nodos (o al final de la lista)

Contaremos aquí con un nodo a borrar, uno anterior y uno posterior. En este caso, el primero está entre los dos últimos, respectivamente.

1. Conservamos el puntero al nodo que queremos borrar.
2. Asignamos la referencia del nodo posterior como el siguiente en el nodo anterior (o NULL en caso de que el nodo a borrar sea el último).
3. Liberamos la memoria del nodo borrado.

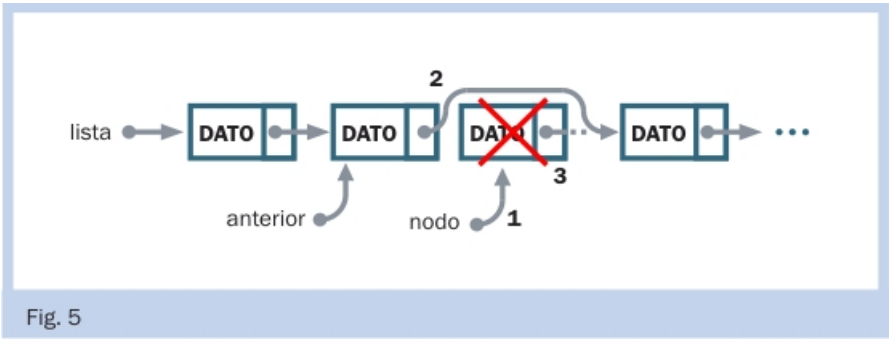


Fig. 5

A continuación, el código de ejemplo define el método *Borrar,* que contempla los dos casos de eliminación descriptos anteriormente.

```
void lista::Borrar(int v){
   pnodo anterior, nodo;

   nodo = primero;
   anterior = NULL;
   while(nodo && nodo->valor < v) {
      anterior = nodo;
      nodo = nodo->siguiente;
   }
   if(!nodo || nodo->valor != v) return;
```

```
        else { // Borrar el nodo
            if(!anterior) // Primer elemento
                primero = nodo->siguiente;
            else  // un elemento cualquiera
                anterior->siguiente = nodo->siguiente;
            delete nodo;
        }
    }
```

2.2. Listas enlazadas circulares

Son un tipo especial de listas donde la única diferencia con respecto a las abiertas o simples es que su último elemento apunta al primero en lugar de establecerlo a NULL.

Las listas circulares evitan excepciones en las operaciones que se realicen sobre ellas. No existen, por ejemplo, casos especiales de final o principio de lista al momento de buscar; cada nodo siempre tiene uno anterior y uno siguiente.

Sin embargo, además de simplificar las operaciones en sí mismas, las listas circulares acarrean algunas complicaciones. Por ejemplo, en un algoritmo de búsqueda no es tan sencillo dar por terminada la búsqueda cuando el elemento buscado no existe.

Por esta causa, una alternativa que simplifica de cierta manera el uso de listas circulares consiste en crear un nodo especial que cumplirá la función de nodo cabecera. De esta manera, la lista nunca estará vacía y se eliminarán casi todos los casos especiales.

La estructura de nodo y de lista es similar al de las listas abiertas:

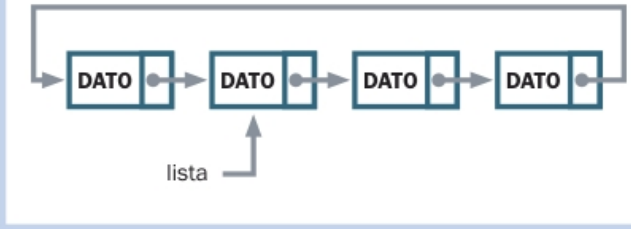


Fig. 6

2.2.1. Operaciones básicas

Las operaciones básicas de una lista enlazada circular consisten en búsqueda, inserción, modificación y eliminación de un nodo.

Búsqueda

En una lista enlazada circular las búsquedas son, al igual que en las abiertas, únicamente secuenciales y en un solo sentido. Lo único a tener en cuenta es que, al empezar a recorrer una lista, debemos marcar un nodo cualquiera como pivote y, a partir de éste, ir tomando los enlaces hasta que lleguemos al nodo buscado. Si al término de la búsqueda nos encontramos con el nodo pivote, ello significará que hemos recorrido todos los nodos sin encontrar el elemento.

Inserción

Las listas presentan diferentes casos de inserción, de acuerdo a la posición donde se quiera ubicar el elemento y al estado de la lista. Para todos los casos, suponemos que contamos con el nodo creado.

La única situación especial que debemos tener en cuenta al momento de insertar nodos en listas circulares es cuando ésta se encuentra vacía.

Inserción en una lista vacía

1. Asignamos el nodo actual de la lista, que es NULL, al nodo creado.

2. Hacemos que *nodo* apunte al siguiente, es decir, a sí mismo.

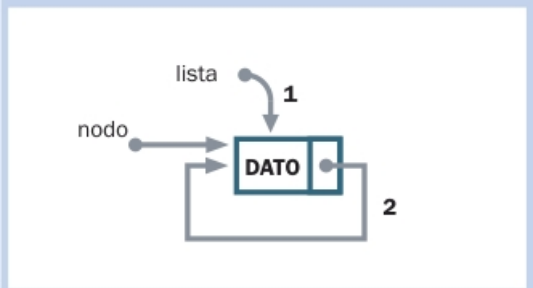


Fig. 7

Inserción en una lista con nodos

1. Hacemos que el nuevo nodo apunte al mismo lugar adonde apunta el nodo actual de la lista.

2. Después, que el nodo actual de la lista apunte a *nodo.*

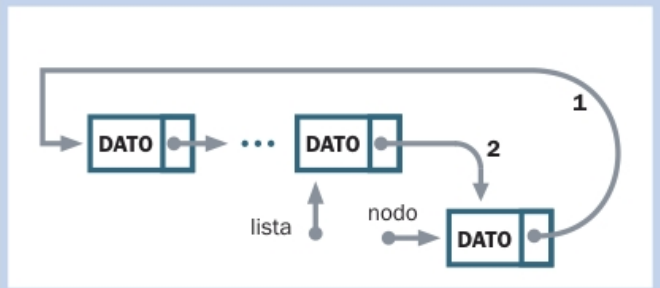


Fig. 8

```
void lista::Insertar(int v) {
   pnodo Nodo;

   // Creamos un nodo para el nuevo valor a insertar
   Nodo = new nodo(v);

   // Si la lista está vacía, la lista será el nuevo nodo
   // por lista
   if(actual == NULL) actual = Nodo;

   // Si no lo está, insertamos el nuevo nodo a continuación del
apuntado
   else Nodo->siguiente = actual->siguiente;

   // En cualquier caso, cerramos la lista circular
   actual->siguiente = Nodo;
}
```

Eliminación

Eliminación de un nodo en una lista circular con más de un elemento

Supongamos que se quiere eliminar un nodo señalado y que *lista* es un puntero utilizado como pivote.

1. El primer paso es conseguir que *lista* apunte al nodo anterior al que queremos eliminar. Esto se consigue iterando y haciendo que *lista* valga su siguiente, mientras que su siguiente sea distinto de *nodo.*

2. Hacemos que *lista* apunte adonde lo está haciendo *nodo.*

3. Eliminamos el nodo.

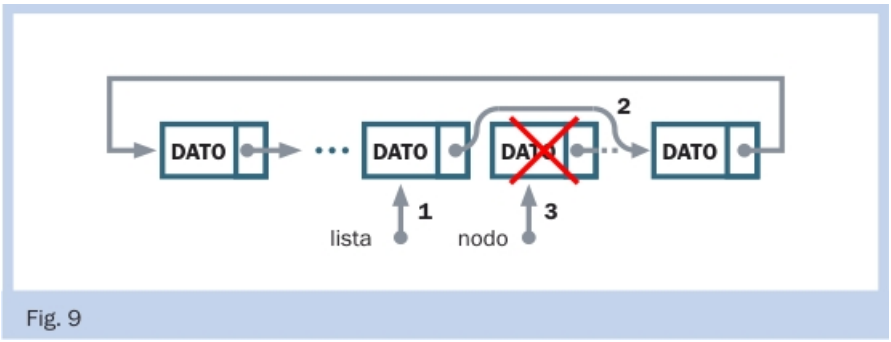


Fig. 9

Eliminación del único nodo en una lista circular

Este caso es mucho más sencillo. Si *lista* es el único nodo de una lista circular:

1. Borramos el nodo apuntado por *lista.*

2. Hacemos que *lista* valga NULL.

```
void lista::Borrar(int v) {
   pnodo nodo;

   nodo = actual;

   // Hacer que lista apunte al nodo anterior al de valor v
   do {
      if(actual->siguiente->valor != v) actual = actual->siguiente;
   } while(actual->siguiente->valor != v && actual != nodo);

   // Si existe un nodo con el valor v:
   if(actual->siguiente->valor == v) {

      // Y si la lista sólo tiene un nodo
      if(actual == actual->siguiente) {
       // Borrar toda la lista
         delete actual;
         actual = NULL;
      } else {
       // Si la lista tiene más de un nodo, borrar el nodo  de valor v
         nodo = actual->siguiente;
         actual->siguiente = nodo->siguiente;
         delete nodo;
      }
   }
}
```

2.3. Aplicación de listas enlazadas en videojuegos

Como ya hemos mencionado, existen ventajas de listas vinculadas por sobre los arreglos.

Los arreglos son una buena forma de almacenar información, pero únicamente cuando sabemos la cantidad con exactitud. Por ejemplo, si tuviésemos que desarrollar un juego de tipo plataforma con quince monstruos en cada nivel, los arreglos constituirían una opción suficiente. Pero si deseáramos agregar o eliminar monstruos, es decir, permitir la creación dinámica de objetos, los arreglos resultarían insuficientes y deberíamos evitar su uso.

Aquí es donde las listas entran en juego, pues una lista vinculada puede ser usada en casi todo el diseño de un juego: personajes, monstruos, partículas, proyectiles, ítems, etc. Agregar nuevos objetos es prácticamente fácil; sólo debemos crear un nuevo nodo que será nuestro objeto, en el cual C++ le adjudicará una posición de memoria para su alojamiento y lo conectará a la lista.

Veamos esto con un ejemplo práctico en C++. Supongamos que tenemos la siguiente clase `Proyectil` en un juego 2D:

```
class Proyectil {
public:
float _x; //La posición x del proyectil
float _y; //La posición y del proyectil
        bool isDead; //Cuando se setea en true, el bucle de
        renderizado
                //detecta que debe eliminarse este proyectil.
Proyectil* next; //El siguiente proyectil(nodo) en la lista
};

//Definimos dos punteros para la lista
Proyectil* primerProyectil = NULL; //La cabeza de la lista
Proyectil* ultimoProyectil = NULL; //La cola de la lista
```

Ahora, los siguientes métodos nos proporcionarán una estructura tradicional utilizada en videojuegos, la cual nos permitirá manipular eficientemente nuestra lista de objetos, en este caso, de proyectiles.

1)
```
Proyectil* NuevoProyectil(int x, int y);
```

Crea un nuevo proyectil en la posición *x, y,* es decir, un nuevo nodo a la lista. Esta función es llamada, por ejemplo, en el instante en que se presiona el gatillo de un lanzacohetes.

2)
```
void ActualizarProyectiles();
```

Esta función de actualización es llamada en cada bucle principal del juego, donde actualiza todos los proyectiles de la lista. En su cuerpo es donde se debe programar el comportamiento de los proyectiles.

3)
```
void RenderProyectiles();
```

Esta función es llamada al final del bucle principal del juego. Se encarga de dibujar todos nuestros proyectiles en pantalla. También borra cualquier proyectil que ha explotado, salido de la pantalla, etc.

A continuación presentamos los cuerpos de los métodos descriptos:

```
Proyectil* NuevoProyectil(int x,int y, int nro) {
existenProyectiles = true;

        //En este caso la lista se encuentra vacía
if(primerProyectil == NULL) {
        //Creamos un nuevo proyectil que será la cabecera de la
lista
        primerProyectil = new Proyectil;

                //No hay otros nodos en la lista, asi que la cola
                debe ser igual a //la cabecera
                ultimoProyectil = primerProyectil;

        //La cola debe apuntar a NULL
        ultimoProyectil->next = NULL;
} else {
                //Agregar un nodo al final de la lista
        ultimoProyectil->next = new Proyectil;

        //Desplazar la cola al final de la lista
        ultimoProyectil = ultimoProyectil->next;
```

```
                //El siguiente del ultimo es NULL
        ultimoProyectil->next = NULL;
}

        //En este ejemplo x, será igual al parametro pasado a la
        funcion y no //cambiará
ultimoProyectil->x = x;

//Será igual al parametro pasado a la funcion y será actualizado
        ultimoProyectil->y = y;

        //Será 'true' solamente cuando se desee eliminar el
proyectil
ultimoProyectil->eliminar = false;

        //Asignamos un numero de identificacion
ultimoProyectil->nro = nro;

        cout << "Se creo el proyectil numero " << ultimoProyectil-
>nro << endl;

return ultimoProyectil; //Devolvemos un puntero al nuevo proyectil
}
```

En este método se observan ambos casos de inserción. Esto es, cuando la lista se encuentra vacía y cuando se inserta un elemento al final:

```
void ActualizarProyectiles() {
        //proyectilActual es un puntero para acceder a la lista,
        desde la //cabecera, y luego moverse a través de la lista,
        actualizando cada nodo.
Proyectil* proyectilActual = primerProyectil;

        while(proyectilActual!=NULL){  //Mientras no se llegue al
final
        proyectilActual->y -= 2;  //Mover el proyectil actual hacia
arriba

                cout << "La Posicion y del proyectil nro: " <<
                proyectilActual ->nro << " es " << proyectilActual-
                >y << endl;

        if(proyectilActual->y < 0)
                        //Si  el  proyectil  sale  de  pantalla,
                        entonces debe ser //eliminado (borrado de
                        la lista)
                proyectilActual->eliminar = true;

                //Avanzamos hacia el siguiente nodo
        proyectilActual = proyectilActual->next;
        }
}
```

Básicamente, se decrementa el valor de la posición *y* de cada proyectil. Cuando el mismo supera el alto de la pantalla, se marca para ser eliminado:

```
void RenderProyectiles() {
        //Accedemos a la lista
Proyectil* proyectilActual = primerProyectil;

        //El puntero proyectilAEliminar se utiliza para eliminar un
        nodo de la //lista
Proyectil* proyectilAEliminar = NULL;

        //Se utiliza para fines de borrado
        Proyectil* nodoAnterior = NULL;

while(proyectilActual!=NULL){
                //Si el nodo esta marcado para eliminarse, entonces
                lo removemos de //la lista
                if(proyectilActual->eliminar) {
                proyectilAEliminar = proyectilActual;

                        //Avanzamos al siguiente nodo
```

```
                        proyectilActual = proyectilActual->next;


                        if(primerProyectil == proyectilAEliminar){
                                primerProyectil = proyectilActual;
                                nodoAnterior = primerProyectil;
                        } else {
                                nodoAnterior->next                =
                        proyectilActual;
                        }

                        cout << "Se elimino el proyectil nro : " <<
                        proyectilAEliminar->nro << endl;

                        proyectilAEliminar->next       =       NULL;
                //Eliminamos enlace

                        //Se elimina el puntero de nuestra lista y
                de la memoria
                delete proyectilAEliminar;
        } else {
                        //Aquí es donde se debe insertar el código
                        de renderizado, //dibujando el proyectil en
                        pantalla usando alguna librería //grafica
                        como SDL
                        nodoAnterior = proyectilActual;

                        //Avanzamos al siguiente nodo
                proyectilActual=proyectilActual->next;
                }
}

}
```

Si existen nodos marcados, listos para eliminar (en este caso, cuando se van de pantalla), los borra, liberando memoria. De lo contrario, los muestra en pantalla.

Finalmente, presentamos la función principal del juego. Se crean dos proyectiles y debajo comienza el bucle principal del juego, que actualiza las posiciones de los proyectiles y los muestra en pantalla. Este bucle finalizará cuando todos los proyectiles sean destruidos, o lo que es lo mismo, cuando no haya nodos en la lista.

```
int main(int argc, char *argv[]) {
        //Creamos los proyectiles
NuevoProyectil(0,100,1);
NuevoProyectil(0,200,2);

//Bucle principal
while(primerProyectil != NULL){ //Mientras existan proyectiles
        ActualizarProyectiles();
        RenderProyectiles();
}

return 0;
}
```

2.4. Pila

La pila es un tipo de estructura de datos muy utilizada, aunque para propósitos específicos como, por ejemplo, para ordenar las llamadas que se generan en funciones recursivas, pues solamente se puede acceder en todo momento al último elemento agregado. Por ello se la conoce también como estructura *LIFO (Last In First Out),* ya que en una pila el último elemento en entrar es siempre el primero en salir. Las implementaciones generalmente se realizan mediante arreglos o listas. Aquí utilizaremos las últimas para nuestro estudio.

El nodo de una pila es similar al de una lista simple, por lo que omitiremos la explicación de su código.

Estructura básica de una pila con elementos:

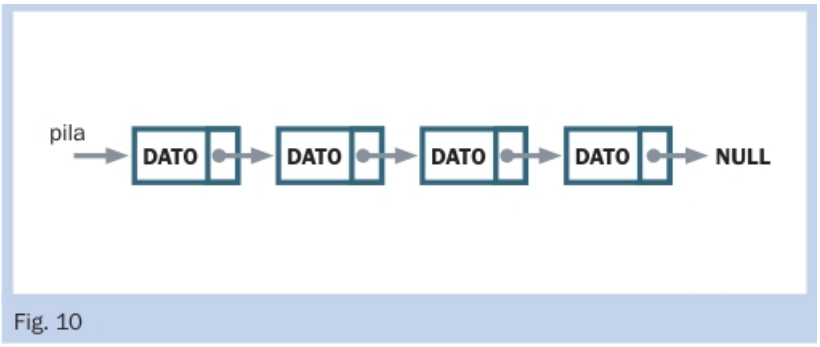


Fig. 10

2.4.1. Operaciones básicas

Las pilas poseen operaciones similares a las demás estructuras, aunque con prestaciones más restrictivas debido a su naturaleza.

Empujar (Push)

Permite añadir un elemento a la pila. Adquiere este nombre porque el último elemento siempre va primero, lo cual nos obliga a empujar a los demás hacia atrás. El proceso coincide con la inserción de un nodo al inicio de una lista.

1. Asignamos el valor de la cabeza de la lista como el siguiente en el nuevo nodo (o NULL en caso de que la lista este vacía).

2. Asignamos a la cabeza de la lista la referencia al nuevo nodo.

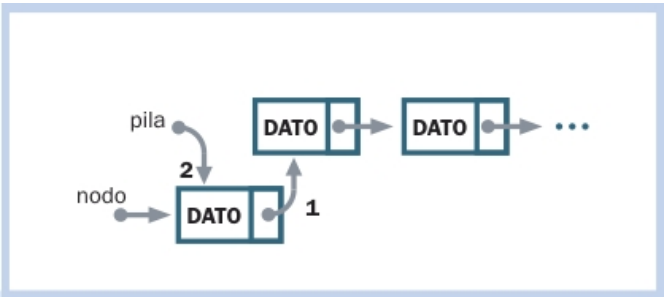


Fig. 11

Seguidamente, el código presenta el método *push*, encargado de realizar las inserciones en la pila.

```
void pila::Push(int v){
   pnodo nuevo;

   // Creamos un nuevo nodo
   nuevo = new nodo(v, cabeza);

   // Asignamos el nuevo nodo a la cabeza
   cabeza = nuevo;
}
```

Sacar (Pop)

Permite leer y eliminar un elemento de la lista. Por ello se clasifica como una operación de lectura destructiva. Coincide con la eliminación del elemento inicial de una lista y con la lectura previa del valor.

1. Conservamos el valor y el puntero al nodo que queremos borrar.

2. Asignamos la referencia al siguiente nodo a la cabeza (o NULL en caso de que la lista quede vacía).

3. Liberamos la memoria del nodo borrado.

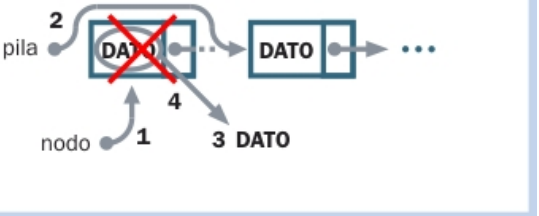


Fig. 12

Debajo, vemos cómo se define un método *pop,* utilizado para sacar y leer un elemento en una pila:

```
int pila::Pop(){
   pnodo nodo;
   int v;

   //Control de pila vacia
   if (!ultimo) return 0;

   // Resguardamos el primer elemento
   nodo = cabeza;

   // Ponemos a la cabeza el siguiente elemento
   cabeza = nodo->siguiente;

   // Retornamos el valor

   v = nodo->valor;

   // Borramos el nodo
   delete nodo;
   return v;
}
```

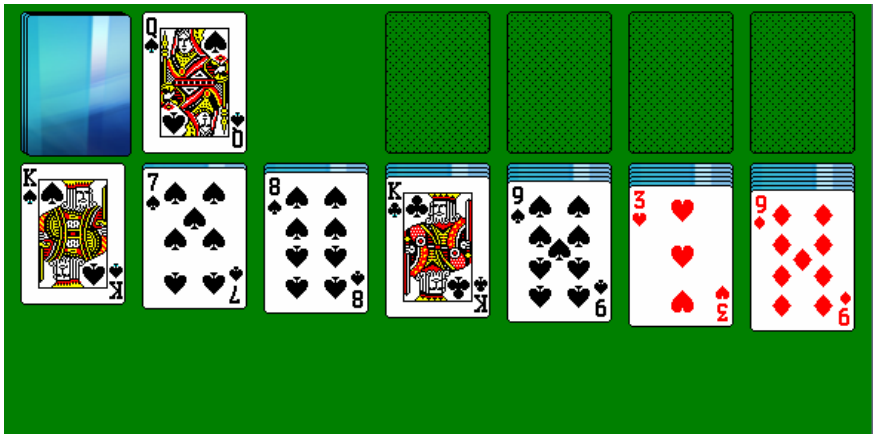
En ciertas implementaciones también existe una lectura no destructiva del primer elemento, ya que no siempre se quiere borrar el elemento al leerlo.

2.4.2. Aplicación de pilas en videojuegos

Tomemos como ejemplo a un simple juego de cartas como el clásico *Solitario,* en el cual las diferentes agrupaciones de cartas son representadas por estructuras de tipo pila.

Este juego está conformado por diferentes pilas, ubicadas en pantalla de la siguiente manera: arriba y a la izquierda, el mazo principal boca abajo, de donde se van descubriendo a su costado las nuevas cartas, de una en una; arriba y a la derecha, cuatro columnas que van a alojar los distintos palos ordenados desde el As hasta K, y finalmente, más abajo, siete columnas con diferentes cantidades de cartas donde se realizan los movimientos, según las reglas.

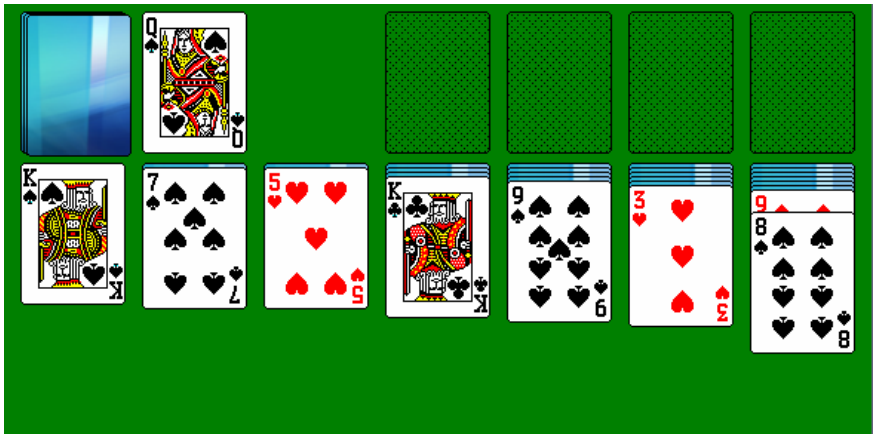
La siguiente figura es una captura del estado inicial de juego:



Así como de una pila de cartas se puede tomar o sacar una de ellas de la parte superior, en una estructura de pila puede realizarse una operación *pop,* que saca un elemento de la misma para su lectura, de uno en uno, desde un mismo extremo. En los casos en que sea necesario sacar un grupo de cartas, se crea una iteración y se realizan los pop necesarios.

En nuestro ejemplo representaremos cada montículo de la segunda fila con una estructura de pila. Cuando ocurre un movimiento permitido por el juego, las pilas correspondientes se actualizan.

La siguiente figura es una captura después del primer movimiento del jugador:



La carta 8 de Picas, que estaba en la columna 3, se coloca en la última columna donde existe una carta consecutiva mayor en número y de palo opuesto en color.

Veamos la estructura básica de una clase carta, que será nuestro nodo de pila:

```
class carta {
public:
        bool estaDescubierta; // Si es visible o esta boca abajo
        int numero; // 1 a 13 del As al Rey(K)
        char palo; //C = Corazones, T = Treboles, P = Picas, D =
Diamantes

        //Constructor
        carta(int pnumero, char ppalo, bool desc){
                numero = pnumero;
                palo = ppalo;
                estaDescubierta  = desc;
        }
};
```

Cada vez que el jugador apila un grupo de cartas de una columna a otra, se deben realizar los siguientes pasos:

- Sacar el grupo de cartas, es decir realizar *pop,* de la pila origen.

- Insertar el nuevo grupo de cartas a la pila destino.

El siguiente código se utiliza para llevar a cabo estos pasos.

Supongamos que tenemos la siguiente partida, donde sólo mostraremos las primeras cuatro columnas de la segunda fila del juego.

El estado inicial nos daría como resultado:

```
---------------------------------
| Col 1 | Col 2 | Col 3 | Col 4 |
---------------------------------
|  12T  |  ---  |  ---  |  ---  |
|       |  10T  |  ---  |  ---  |
|       |       |  11D  |  ---  |
|       |       |       |  13C  |
```

El único movimiento posible entre las columnas es mover el diez de Tréboles sobre el once de Diamantes.

```
//Obtiene y elimina de la pila origen un carta con el método pop y
almacena en una estructura array.
cartas = pilaOrigen->pop(1);

//Damos vuelta la carta que queda en la pilaOrigen.
pilaOrigen->top()->estaDescubierta = true;

//Pasa como parametro el grupo a la pilaDestino.
pilaDestino->push(cartas);
```

El resultado obtenido es el siguiente:

```
---------------------------------
| Col 1 | Col 2 | Col 3 | Col 4 |
---------------------------------
|  12T  |  2D   |  ---  |  ---  |
|       |       |  ---  |  ---  |
|       |       |  11D  |  ---  |
|       |       |  10T  |  13C  |
```

Como vemos, es posible otro intercambio. Entonces, esta vez sacamos dos elementos de la columna 3 y los insertamos en la única opción posible, la columna 1.

```
---------------------------------
| Col 1 | Col 2 | Col 3 | Col 4 |
---------------------------------
|  12T  |  2D   |  ---  |  ---  |
|  11D  |       |  8P   |  ---  |
|  10T  |       |       |  ---  |
|       |       |       |  13C  |


//Obtiene y elimina de la pila origen dos cartas con el método pop
y almacena en una estructura array.
cartas = pilaOrigen->pop(2);

//Damos vuelta la carta que queda en la pilaOrigen.
pilaOrigen->top()->estaDescubierta = true;

//Pasa como parametro el grupo a la pilaDestino.
pilaDestino->push(cartas);
```

Las pilas pueden utilizarse también para el mazo principal y repositorio de cartas ordenadas. Asimismo, como alternativa en este juego, pueden usarse estructuras de cola como ayuda, por ejemplo, para imprimir en pantalla las columnas de cartas, ya que es necesario mostrar desde el fondo de las pilas hacia la parte superior. Si esas

pilas se imprimiesen directamente, se obtendría un dibujo inverso, ya que sólo se pueden tomar objetos desde un extremo superior y no es posible acceder a los elementos de otra posición.

2.5. Cola

Es otra estructura de datos de lista abierta fundamental. También se la conoce con el nombre de *FIFO (First In First-Out).* Los elementos entran por un extremo y salen por el otro y se rigen por la regla opuesta a la estructura de pila, es decir, el primer elemento que entra a la lista es el primero en salir. Dicho de otra manera, el elemento que más tiempo ha estado en la cola es el primero en salir. Esto sucede, por ejemplo en un surtidor de combustible: el auto que llega primero a la cola es atendido y sale; luego continúa el vehículo que está detrás, y así sucesivamente hasta terminar la cola.

La estructura del nodo de una cola es similar a la de una lista, así que omitiremos la explicación de su código.

Estructura básica de una cola con elementos:

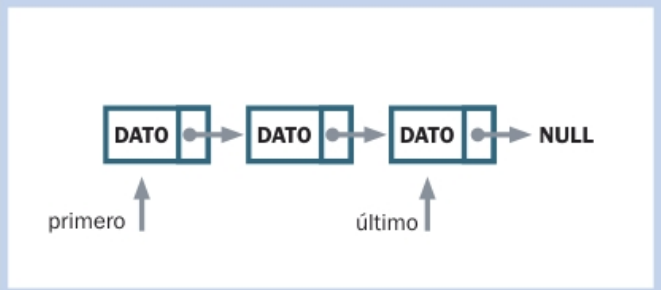


Fig. 13

2.5.1. Operaciones básicas

Las colas poseen operaciones similares a las demás estructuras, aunque con prestaciones más restrictivas debido a su naturaleza.

Inserción

Al insertar un elemento, lo hace siempre al final de la cola.

1. Se crea un nodo y hacemos que apunte a NULL.
2. Si el último no es NULL, entonces el último apunta a *nodo.*
3. Luego, se actualiza el último, haciendo que apunte a *nodo.*
4. Si primero es NULL, significa que la cola estaba vacía, así que primero apuntará también a *nodo.*

El siguiente código describe el método ***Anadir,*** utilizado para la inserción de nuevos elementos a la cola:

```
void cola::Anadir(int v) {
   pnodo nuevo;

   // Crear un nodo nuevo
   nuevo = new nodo(v);

   // Si la cola no estaba vacía, se añade el nuevo a continuación
de ultimo
   if (ultimo) ultimo->siguiente = nuevo;

   // Ahora, el último elemento de la cola es el nuevo nodo
   ultimo = nuevo;
```

```
        // Si primero es NULL, la cola estaba vacía, ahora primero
        apuntará también al nuevo nodo
        if (!primero) primero = nuevo;
    }
```

Inserción, cuando la cola está vacía:

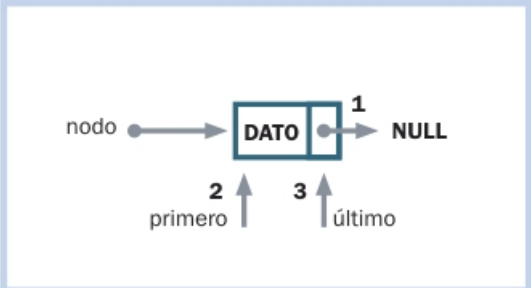


Fig. 14

Inserción, cuando la cola no está vacía:

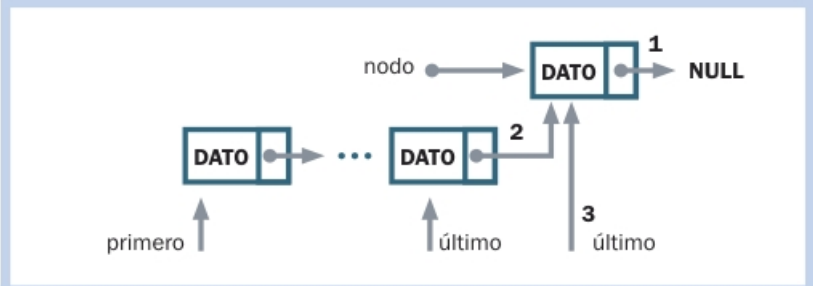


Fig. 15

Leer (eliminar)

Leer un elemento significa eliminarlo. Los elementos se leen del principio de la cola.

Usaremos un puntero a un nodo auxiliar para este algoritmo:

1. Hacer que el nodo apunte al primer elemento de la pila, es decir, al primero.
2. Apuntar el primero al segundo nodo de la lista.
3. Se guarda el contenido del nodo para devolverlo como retorno, (recordar que la operación de lectura en colas implica también borrar).
4. Liberar la memoria asignada al primer nodo, el que se quiere eliminar.
5. Si el primero es NULL, hacer que el último también apunte a NULL, ya que la lectura ha dejado la cola vacía.

El siguiente código describe el método ***Leer,*** utilizado para leer y eliminar un elemento de la cola:

```
int cola::Leer() {
   pnodo nodo; // variable auxiliar para manipular nodo
   int v;      // variable auxiliar para retorno

   // Nodo apunta al primer elemento de la cola
   nodo = primero;
   if (!nodo) return 0; // Si no hay nodods en la cola retornamos 0

   // Asiganamos a primero la dirección del segundo nodo
   primero = nodo->siguiente;
```

```
        // Guardamos el valor de retorno
        v = nodo->valor;

        // Borrar el nodo
        delete nodo;

        // Si la cola quedó vacía, ultimo debe ser NULL también
        if (!primero) ultimo = NULL;

        return v;
    }
```

Eliminación, cuando la cola tiene un solo elemento:

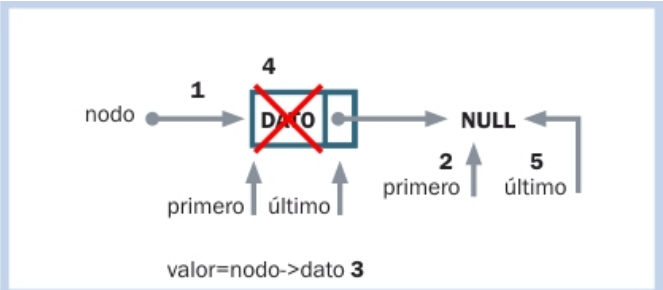


Fig. 16

Eliminación, cuando la cola no está vacía:

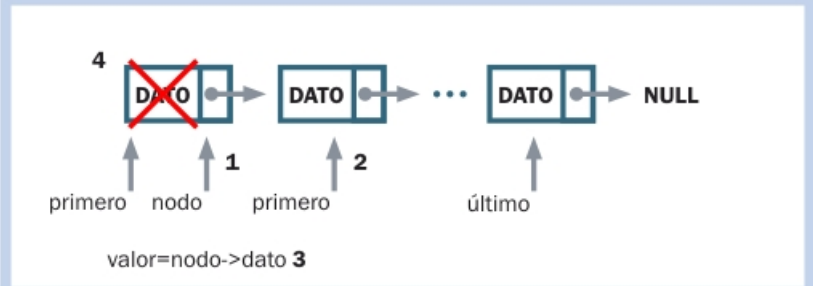


Fig. 17

2.5.2. Aplicación de colas en videojuegos

Como ya hemos explicado anteriormente, las colas son un tipo especial de lista donde los nodos deben insertarse y leerse desde extremos concretos.

Para ilustrar este enunciado, podemos considerar un juego en el cual el objetivo es servir a las personas que llegan a un restaurante, esperar a que coman, pidan la cuenta para recaudar el dinero y dejen libre la mesa para nuevos clientes. Si el servicio no es lo suficientemente rápido, perderemos clientes. Además, contamos con un único mesero.

Evidentemente es claro qué estructura debe utilizarse para el arribo de personas: una cola, tal como sucede en la vida real. Es decir, las personas que llegan a la puerta del local, comienza a colocarse una detrás de otra y las que estén más próximas al ingreso serán atendidas primero.

Ahora, nos concentraremos únicamente en esta acción del juego, es decir, en el arribo y la inserción de personas en la cola.

Como simplificación, nos interesa saber cuántas personas se encuentran comiendo, esto es, cuántas personas fueron atendidas en la mesa. En un juego completo, al atender a un grupo de personas, el nodo correspondiente pasaría a formar parte de otra estructura, pero aquí lo eliminaremos sólo con un fin educativo.

Supongamos que tenemos una clase *GrupoPersona,* que será nuestro nodo en la cola:

```
class GrupoPersona {
public:
        int cantidad; //La cantidad de personas en el grupo
        GrupoPersona * siguiente; //El siguiente grupo(nodo) en la
cola
};

//Definimos dos punteros para la cola
GrupoPersona * primerGrupo = NULL; //La cabeza de la cola
GrupoPersona * ultimoGrupo = NULL; //El ultimo nodo de la cola
```

Ahora, definimos las funciones que nos permiten manipular la cola:

```
void ArribaGrupo(int cantidad);
```

El método se encarga de la inserción de un nuevo nodo a la cola, es decir, un grupo de personas que arriba al local en un momento determinado. También, se indica con un parámetro la cantidad de personas de ese grupo:

```
void AtenderGrupo();
```

Este método se encarga de sacar un elemento de la cola y recuperar su información. En nuestro juego equivale a que el grupo de personas sea atendido por el personal, otorgándole una mesa.

A continuación, se presenta el cuerpo de los métodos, similares a la inserción y lectura de una cola básica, respectivamente, tal como se describen:

```
void ArribaGrupo(int cantidad) {
// Crear un nodo nuevo
GrupoPersona* nuevoGrupo = new GrupoPersona();
nuevoGrupo->cantidad = cantidad;

// Si la cola no estaba vacía, se añade el nuevo a continuación de
ultimo
if (ultimoGrupo)
                ultimoGrupo->siguiente = nuevoGrupo;

        cout << "Arribo un grupo de  " << nuevoGrupo->cantidad << "
        personas " << endl;

// Ahora, el último grupo de la cola es el nuevo grupo que arriba
ultimoGrupo = nuevoGrupo;

        // Si primero es NULL, la cola estaba vacía, ahora primero
        apuntará también al nuevo nodo
if (!primerGrupo)
        primerGrupo = nuevoGrupo;
}
```

Cuando un grupo arriba, se inserta al final y espera allí su turno:

```
void AtenderGrupo() {
GrupoPersona*  grupoAtendido;  // Variable auxiliar para manipular
nodo


if (primerGrupo != NULL){ // Si existen grupos aún

        // grupoAtendido apunta al primer elemento de la cola
        grupoAtendido = primerGrupo;

        // Asignamos a primero la dirección del siguiente nodo de
la cola
        primerGrupo = grupoAtendido->siguiente;

        // Guardamos cuantas personas fueron atendidas
        totalPersonasAtendidas += grupoAtendido->cantidad;

                cout << "Se atendio el grupo de " << grupoAtendido-
                >cantidad << " personas " << endl;
```

```
                // Borrar el nodo (solo como fin práctico). Este
                nodo   podría   seguir   //utilizándose   en   otra
                estructura dentro del juego
        delete grupoAtendido;

        // Si la cola quedó vacía, ultimo debe ser NULL también
        if (primerGrupo == NULL)
                ultimoGrupo = NULL;
}

}
```

Simplemente se atiende a la gente que está al principio de la cola. Cuando el mesero se desocupe, atenderá al siguiente grupo.

Abajo se muestra, en la función de entrada del programa, un posible ejemplo de arribo y atención de los grupos:

```
int main(int argc, char *argv[]) {
        //Se define la semilla de la generación de números al azar
        srand(time(0));

        //Arribo de personas en un intervalo y cantidad al azar
        sleep((rand() % 5) + 1 );
ArribaGrupo((rand() % 6) + 1);

        //Arribo de personas en un intervalo y cantidad al azar
        sleep((rand() % 5) + 1 );
ArribaGrupo((rand() % 6) + 1);

//El mesero comienza la atención
while(primerGrupo != NULL){ //Si la cola aun no esta vacía
        AtenderGrupo();
                sleep((rand() % 10) + 1 ); //Tiempo que tarda el
        mesero en atender
}

        cout << "Estan atendidas "<< totalPersonasAtendidas << "
        personas " << endl;

return 0;
}
```

La cantidad de personas por grupo viene definida al azar con un máximo de seis por grupo. Estos números al azar se toman de la función *rand()*.

Los grupos arriban con diferencias de tiempos entre ellos. En el ejemplo se simulará con un retardo *sleep()*, pasando como parámetro un tiempo en segundos definido al azar, con un máximo de cinco segundos.

Finalmente, el mesero también tiene su retraso al azar para atender a los grupos y es de 10 segundos como máximo.

BIBLIOGRAFÍA

Sedgewick,R. *Algorithms in C++. Parts 1–4: Fundamentals, Data Structure, Sorting, Searching*. Third Edition , , Addison Wesley Professional, ISBN-13: 978-0-201-35088-3

C++ con Clase, http://c.conclase.net/

gamedev.net,
http://www.gamedev.net/page/resources/_/reference/programming/sweet-snippets/using-linked-lists-to-represent-game-objects-r2041