Práctica 3

Integrantes:

Buendía Velazco Abel

Becerra Carpio Gustavo

Hernandez Molina Leonardo Gaell

Velázquez Diaz Luis Francisco

Equipo:

Los tíos pelones

Grupo:

2CM1

Practica 3

INDICE

MARCO TEÓRICO

Las estructuras de datos en el ámbito de la programación son una manera de representar información en una computadora. Cuentan con un comportamiento interno, es decir, estas se rigen por reglas o restricciones específicas que han sido diseñadas con base en el funcionamiento interno de la estructura en cuestión.

Una estructura de datos permite al desarrollador de código organizar la información de manera eficiente. Permite trabajar en un nivel de abstracción alto para almacenar la información y posteriormente acceder a ella, modificarla y, en general, manipularla. Por lo tanto, podemos usarlas para diseñar una solución correcta y eficiente para un determinado problema.

COLA: Una cola es una estructura de datos que almacena elementos en una lista y permite acceder a los datos por uno de los dos extremos de la lista. Un elemento se inserta en la cola (parte final) de la lista y se suprime o elimina por la frente (parte inicial, cabeza) de la lista.

o Los elementos se eliminan (se quitan) de la cola en el mismo orden en que se almacenan y, por consiguiente, una cola es una estructura de tipo FIFO (first-in-first-out, primero en entrar, Primero en salir o bien primero en llegar/primero en ser servido). o El servicio de atención a clientes es un ejemplo típico de cola o el cajero de un banco.

Practica 3

COLA CIRCULAR: La cola circular es una mejora de la cola simple, debido a que es una estructura de datos lineal en la cual el siguiente elemento del último es, en realidad, el primero. La cola circular utiliza de manera más eficiente la memoria que una cola simple.

INSTITUTO POLITÉCNICO NACIONAL

ESCOM

Practica 3

Definición del problema

Con la implementación de TAD COLA en C, resolver los siguientes programas que realizan las siguientes 3 simulaciones.

1) Simulación de atención a clientes en un supermercado.
2) Simulación de la ejecución de procesos en el sistema operativo.
3) Simulación de la atención a clientes en un banco de prioridades.

SIMULACIÓN 01: Supermercado.

- Simular la atención a clientes en un supermercado, el cual deberá de atender al menos 100 clientes por día para no tener pérdidas, por lo que una vez que ya se atendieron 100 personas y no hay gente formada en las cajas puede cerrar la tienda. Mientras no se cierre la tienda, las personas podrán seguir llegando con productos a las cajas.

SIMULACIÓN 02: Ejecución de procesos en el sistema operativo.

- Simular la ejecución de los procesos gestionados por el sistema operativo en un equipo monoprocesador sin manejo de prioridades.
- Manejando únicamente el cambio de la cola de listas a ejecución y una vez terminado el proceso este se envía a la cola de terminados.

SIMULACIÓN 03: BANCO.

- Simular la atención de personas en un banco, cuidando que sean respetadas las políticas de atención de este y evitando que las personas no dejen de ser atendidas.

Para la realización de estos programas fue indispensable hacer uso de varios struct para poder “Encapsular” o abstraer varias entidades que por sí solas, el implementarlas en el código hubiera sido nada práctico y sin dudas complicaba el objetivo de la práctica.

Programa 1:

Para este y todos los programas, propusimos modularizarlo lo más que fuera posible para poder reutilizar funciones y hacer las cosas más sencillas y más concretas, además, propusimos la utilización de structs los cuales abstraían mucho más el problema:

```
typedef struct
{
    char nombre[100];
    int n;
    int tiempos_cajeras[9]; //100 10000
    cola cajeras[9];
    int tiempo_cliente;
} simulacion;
```

Este struct que utilizamos aquí y el cual lleva por nombre “Simulación” es el encargado directo de simular una cajera en todo su contexto, desde su nombre, su tiempo de atención, el número de clientes que va atendiendo y el tiempo en el que lo hace, por lo que hacerla una estructura de este tipo era realmente muy conveniente debido a que si queremos agregar n-cajeras a nuestro programa bastaría solo con darle los datos a una nueva cajera utilizando un ciclo for.

```
printf("Introduce el nombre del supermercado: ");
fgets(S.nombre, 100, stdin);
printf("Introduce el numero de cajeras: ");
scanf("%d", &S.n);
for(i = 0; i < S.n; i++){
    printf("Introduce el tiempo de atencion de la cajera %d: ", i + 1);
    scanf("%d", &S.tiempos_cajeras[i]);
    Initialize(&S.cajeras[i]);
```

Por otro lado, y para apoyarnos más acerca de la utilización de nuestras cajeras, modularizamos una serie de funciones las cuales nos fueron de mucha ayuda para poder llevar a cabo nuestro objetivo, estas funciones de las que hablo son:

```
int aleatorio(int minimo, int maximo);
void Formar(int n_cliente, int cajera, simulacion * S);
int Ejecutarcaj(int cajera, simulacion * S);
int EsVacia(simulacion * S);
```

Formar: La cual recibe el número del cliente a formar, la cajera en la que se formará y a la simulación a la que queramos encolar, es decir, si queremos ingresar al cliente 1, 2 o n en la cajera 1, 2 o n de igual forma.

Ejecutarcaj: La cual recibe un numero que identificará a la cajera que necesitemos ejecutar y la dirección de la cajera en cuestión que estemos manejando

EsVacia: Simplemente recibimos la dirección de la cajera de la cual queramos saber si es vacía o no, es decir, si esta tiene clientes formados o no.

Por ultimo y para darle paso al siguiente punto tenemos

Aleatorio: La cual recibe dos enteros los cuales nombramos como mínimo y máximo, estos números y con la ayuda de la operación módulo, obtendremos un valor entre este rango que nosotros hayamos asignado cómo parámetro

```
int aleatorio(int minimo, int maximo){
    return rand() % (maximo - minimo + 1) + minimo;
```

Como podemos observar, utilizamos rand() la cual nos devuelve un numero pseudoaleatorio y el cual al modularizarlo con la diferencia de máximo contra mínimo obtendremos un numero naturalmente más “aleatorio”. Como podemos darnos cuenta, después del “%” tenemos que estamos sumando un 1 a la diferencia de estos 2 valores propuestos y además le estamos añadiendo nuevamente el valor de mínimo al resultado de esta operación, esto debido a que cómo lo mencionamos antes, rand() nos genera un numero pseudoaleatorio el cual a partir de una serie de aleatorios generados uno podría darse cuenta del patrón que este sigue y además podría brindarnos un numero repetido, por ello utilizamos la operación módulo para evitar este tipo de errores.

Por otro lado, la práctica nos acotaba el tiempo base de cada cajera a un multiplo de 10 ms, esto debido a que la librería brindada por el profesor estaba definida de tal forma, por ello y haciendo uso de un par de enteros nuevos

```
int Tiempo_Transucrrido = 0;
int Tiempo_Base = 10;
```

Como podemos visualizar, además del tiempo base el cual ya definimos arriba, tenemos el tiempo transcurrido el cual es el tiempo que ha pasado cada cajera atendiendo sus clientes, este nos será útil más adelante ya que de igual manera, la práctica tiene

propuesto como objetivo que: “Deberá atender al menos 100 clientes por día para no tener perdidas, por lo que **una vez**

que ya se atendieron a más de 100 personas y no hay gente formada en las cajas puede cerrar la tienda”. De aquí puedo destacar la parte de atender a 100 personas ya que este es una de las condiciones que tiene nuestro programa para terminar la ejecución, la otra condición es que **no haya clientes formados** la cual si recordamos ya describimos un poco más arriba y está descrita por una función la cual indica si hay o no personas formadas en la cola. Por ello es que si no se cumple una condición u otra, nuestro programa no terminará.

```
    if(n_cliente >= 100 && CajerasVacias(&S)){
        printf("Hemos cerrado.\n");
        break;
    }
}
```

Programa 2:

Para este programa se nos dio a la tarea de simular la ejecución de procesos en un sistema operativo, recordando que originalmente el sistema operativo funciona de esta manera por ello es que este es un buen ejemplo para la estructura de datos “Cola” el cual podemos ver de forma más detallada con la siguiente imagen 2.1:

Figura 2.1

Como podemos observar en la figura 2.1 tenemos unas flechas las cuales indican el estado del programa que se esté ejecutando. Por ejemplo:

Al encolar un programa que llamaremos “Programa 1” este permanecerá entre la fase de ejecución y listo, cuando se esté en ejecución, el Programa 1 estará realizando sus tareas de forma normal para posteriormente dar paso a otro programa provocando que este se regresa a la cola de “Listo para despachar”. Una vez su tiempo de ejecución haya terminado pasará a la cola de finalizados para darle fin

Para poder realizar este programa, de igual manera fue de suma importancia utilizar structs y modularizar lo más que fuera posible para hacer que nuestro programa fuera lo más claro y sencillo posible de entender y manipular si es que lo requiriera.

De tal forma que cómo hemos venido abordando, solamente reutilizamos el tipo “Elemento” previamente definido en el punto h de ColaDin para renombrarlo cómo “programa”

```
typedef elemento programa;
```

```
typedef struct elemento
{
    char nombre[50], actividad[100], id[50];
    int tiempo, contador;
    int ID, tipo;
} elemento;
```

Como podemos observar, tenemos una serie de campos los cuales fueron propuestos en general para la utilización de nuestros programas, es decir, en el caso de la simulación 1 el campo int ID es únicamente usado por este programa, el campo int tipo es únicamente usado por el programa 3 y el cual será abordado más adelante.

En este programa optamos por hablar de forma somera las funciones utilizadas ya que estas se limitan en su mayoría a tareas muy simples como mostrar texto en pantalla

```
void prueba (cola *c);
void MuestraActual (programa p, int espera);
void MuestraUltimo (programa p);
void MuestraSiguiente (programa p);
void mostrarTerminado (programa p);
void mostrarFinalizados (cola *c);
void ProcesaCola (cola *c);
void ProtFunc (char *s, int lim);
void PideDatos (cola *c);
```

Para poder determinar el tiempo de ejecución de cada programa deberemos de adentrarnos un poco más en ciertas funciones para poder comprender cuando es que terminará de ejecutarse.

```
void MuestraActual (programa p, int espera){
    puts ("\nProceso en Ejecucion");
    printf ("Nombre del programa: %s \n", p.Nombre);
    printf ("ID: %s \n", p.id);
    printf ("Actividad: %s \n", p.Act);
    printf ("Tiempo total que lleva ejecutandose: %d segundos \n", p.contador + espera);
    return;
}
```

Por ejemplo, en esta función, en el ultimo printf tenemos que hay un campo dentro del programa “p” el cual hace alusión a un contador, el cual se refiere al tiempo que el programa debe estar en ejecución, este se está sumando con un entero llamado ”espera”, el cual es el tiempo que ha estado en espera, esto es destacable ya que se

repite algo similar en otras 3 funciones, sólo que restando o sumando otros campos ya que podemos obtener distintos resultados a partir de estos.

```
void ProcesaCola (cola *c){
    programa p;
    cola Finalizados;
    int Segundos = 1000, TiempoEspera = 0;

    Initialize (&Finalizados);

    while (!Empty (c)){
        p = Dequeue (c);
        MuestraActual (p, TiempoEspera);

        if (Size (c) >= 1){
            MuestraUltimo (Final (c));
            MuestraSiguiente (Front (c));
        }
        p.contador++;
        TiempoEspera++;

        if (p.contador < p.tiempo)
            Queue (c, p);
        else {
            mostrarTerminado (p);
            p.tiempo += TiempoEspera;
            Queue (&Finalizados, p);
        }
        EsperarMiliSeg (1 * Segundos);
        BorrarPantalla ();
    }
    mostrarFinalizados (&Finalizados);
    return;
}
```

Esta parte del programa es realmente importante ya que es la encargada de procesar toda la cola, es decir, prácticamente es la encargada de que el programa realice lo propuesto.

Tenemos una variable entera nombrada “Segundos”, la cual nos sirve para obtener los tiempos de ejecución en el mencionado formato, por otro lado, el tiempo de espera que nos servirá para determinar el tiempo que se ha estado en dicha situación.

El algoritmo funciona de esta forma:

1. Mientras no sea vacía la cola entonces desencolamos y guardamos en “p” el ultimo elemento de la cola y lo mostramos en pantalla
2. Si el tamaño es mayor igual a 1, entonces hay al menos otro elemento en la cola y por ello debemos mostrarlo y aumentamos su contador y tiempo de espera
3. Si el tiempo de ejecución es menor al tiempo propuesto entonces volvemos a encolarlo, en caso contrario, lo vamos a encolar a la de finalizados.
4. Mostramos la cola de finalizados

Programa 3:

Para la realización de este programa, de igual forma nos apoyamos de modularizar el programa para hacerlo más sencillo.

El objetivo de este programa es: “Simular la atención de personas en un banco, cuidando sean respetadas las políticas de atención del mismo y evitando que las personas no dejen de ser atendidas”. Contaremos con 1 o 10 cajas de operación las cuales pueden atender a tres filas (Clientes, usuarios y preferentes). Tenemos que los clientes son atendidos por cualquier cajero y nunca dejan de ser atendidos por alguna caja, los usuarios son atendidos según la disponibilidad de una caja cuidado que no

pasen más de 5 personas de las otras dos filas sin que una de esta sea atendida y la de preferentes las cuales son atendidos por cualquier cajero disponible con mayor prioridad que a los usuarios.

```
int LiberaCajeros(simulacion * S);
int CajeroEstaLibre(simulacion * S);
void FormarPersona(int n_persona, int tipo, simulacion * S);
void PasarPersona(int n_persona, int tipo, int donde, simulacion * S);
int PuedePasar(int tipo, simulacion * S);
void EjecutaPersonaXD(int n_persona, int tipo, simulacion * S);
int EjecutaColas(simulacion * S);S
```

Para comenzar a entender el programa, debemos de entender el funcionamiento básico de las operaciones implementadas.

LiberaCajeros: Es básicamente inicializarlos, simplemente deja a un cajero desocupado

CajeroEstaLibre: Devuelve un entero para verificar si el cajero esta o no libre, es decir, si tiene o no clientes formados y listos para pasar

FormarPersona: Recibe un int el cual identifica a la persona que llega, su tipo (cliente, usuario o preferente), un int donde en el cual indica a que cajera irá y una simulación s

PuedePasar: A partir de su tipo, verifica si la persona puede o no pasar al cajero

EjecutaPersona: Recibe un int que identifica a la persona, su tipo y la simulación en la que se ejecutará

EjecutaColas: Realiza la ejecución de la cola

Para el algoritmo de las políticas de atención realizamos lo siguiente:

```
void FormarPersona(int n_persona, int tipo, simulacion * S){
    elemento persona;
    persona.ID = n_persona;
    persona.tipo = tipo;
    Queue(&S->colas[tipo], persona);
    return;
}
```

Primeramente, es importante reconocer a donde se formará la persona, esto lo sabemos a partir de su tipo el cual se indica en el parámetro de la función.

```
int PuedePasar(int tipo, simulacion * S){
    int disponible = CajeroEstaLibre(S);
    if(disponible == -1) return -1;
    if(S->clientes_y_preferentes == 5){
        if((tipo == 0 || tipo == 1) && !Empty(&S->colas[2]))
            return -1;
        return disponible;
    }
    return disponible;
}
```

En esta función podemos ver la parte de “Nunca permitiendo que pasen más de 5 personas de las otras filas sin que una persona de la fila de usuarios sea atendida”. Primeramente revisamos si el cajero que estamos manipulando está disponible, si no lo está entonces terminamos y seguimos el algoritmo, si no entonces si el campo de clientes y preferentes tiene que han pasado 5 de estos desde el ultimo usuario, entonces no puede pasar un cliente o preferente pero SI un usuario. Si no sucede ninguno de esos casos entonces está disponible para pasar

```
void EjecutaPersonaXD(int n_persona, int tipo, simulacion * S){
    int pos = PuedePasar(tipo, S);
    if(Empty(&S->colas[tipo]) && pos != -1){
        PasarPersona(n_persona, tipo, pos, S);
    }else{
        FormarPersona(n_persona, tipo, S);
    }
    return;
}
```

Dada una persona, su tipo y la simulación, determina si la persona podría pasar directamente a algún cajero disponible o se tendría que formar a su cola determinada

```
void PasarPersona(int n_persona, int tipo, int donde, simulacion * S)
    elemento persona;
    persona.ID = n_persona;
    persona.tipo = tipo;
    S->cajeros[donde].persona = persona;
    S->cajeros[donde].ocupado = TRUE;
    if(tipo == 2)
        S->clientes_y_preferentes = 0;
    else
        S->clientes_y_preferentes++;
    return;
}
```

En esta función de aquí pasamos a una persona a formarse dependiendo de su tipo y una posición de cajero que sea válida, lo pone como la persona que está siendo atendida en ese cajero y además cambia el estado del cajero de libre a desocupado.

```
int EjecutaColas(simulacion * S){
    int i = 0, pos, cuantas = 0;
    elemento persona;
    for(i = 0; i < 3; i++){
        if(!Empty(&S->colas[i])){
            persona = Front(&S->colas[i]);
            pos = PuedePasar(persona.tipo, S);
            if(pos != -1){
                PasarPersona(persona.ID, persona.tipo, pos, S);
                Dequeue(&S->colas[i]);
                cuantas++;
            }
        }
    }
    return cuantas;

}
```

Por último tenemos esta función la cual se ejecuta cada que es un tiempo de atención y determina si las personas que están esperando al frente de cada cola pueden o no pasar a algún cajero. Si sí pueden pasar entonces las atiende y las desencola

IMPLEMENTACIÓN DE LA SOLUCIÓN

Para poder implementar la solución de cada uno de los programas, como ya revisamos un poco más arriba tuvimos que hacer uso del principio de “divide y vencerás” el cual consiste en fragmentar nuestro problema en partes pequeñas y resolver cada parte de forma individual para después unirlas todas y hacer que el programa funcione.

Programa 1:

```
int aleatorio(int minimo, int maximo);
void Formar(int n_cliente, int cajera, simulacion * S);
int Ejecutarcaj(int cajera, simulacion * S);
int EsVacia(simulacion * S);
```

Programa 2:

```
void prueba (cola *c);
void MuestraActual (programa p, int espera);
void MuestraUltimo (programa p);
void MuestraSiguiente (programa p);
void mostrarTerminado (programa p);
void mostrarFinalizados (cola *c);
void ProcesaCola (cola *c);
void ProtFunc (char *s, int lim);
void PideDatos (cola *c);
```

Programa 3:

```
int LiberaCajeros(simulacion * S);
int CajeroEstaLibre(simulacion * S);
void FormarPersona(int n_persona, int tipo, simulacion * S);
void PasarPersona(int n_persona, int tipo, int donde, simulacion * S);
int PuedePasar(int tipo, simulacion * S);
void EjecutaPersonaXD(int n_persona, int tipo, simulacion * S);
int EjecutaColas(simulacion * S);S
```

Estas fueron todas las funciones que utilizamos en nuestros programas, podemos ver que fueron demasiadas debido a que de esta manera hacíamos las cosas más fáciles para nosotros y a la manipulación si es que se requiere.

Lo complicado de esto radica en que tuvimos que revisar que todas nuestras ideas coincidieran con las sentencias permitidas en el lenguaje de C++ y si no sucedía teníamos que idearnos una manera para que lo hiciera, tal es el caso de una función implementada en el programa 2 la cual lleva por nombre:

```
void ProtFunc (char *s, int lim);
```

Esto surge debido a que queríamos utilizar la función fgets pero esta recibía datos de un archivo y no daba fin de carácter a cada string, así que decidimos implementar esta solución para hacer más fácil el trabajo.

FUNCIONAMIENTO:

Simulación 1: Supermercado.

Primeramente, la compilación del programa y la ejecución de ".exe"

```
C:\Users\USER\Desktop\CODIGOS\hoa7\practica 2> gcc programa1.c ColaDin.c presentacionWin.c -o Simulacion1

C:\Users\USER\Desktop\CODIGOS\hoa7\practica 2>Simulacion1.exe
Introduce el nombre del supermercado:
```

Al ejecutar el programa nos pide el nombre del supermercado, enseguida el número de cajeras que queremos simular, después nos pide el tiempo de atención de las cajeras y al final el tiempo en el que van a llegar los clientes (todos los tiempos en milisegundos):

```
C:\Users\USER\Desktop\CODIGOS\hoa7\practica 2>Simulacion1.exe
Introduce el nombre del supermercado: ESCOMarket
Introduce el numero de cajeras: 3
Introduce el tiempo de atencion de la cajera 1: 2000
Introduce el tiempo de atencion de la cajera 2: 2100
Introduce el tiempo de atencion de la cajera 3: 2300
Introduce el tiempo de llegada de los compradores: 1500
```

Después comienza la simulación del proceso, que cajeras están atendiendo y cuando se va formando un cliente nuevo:

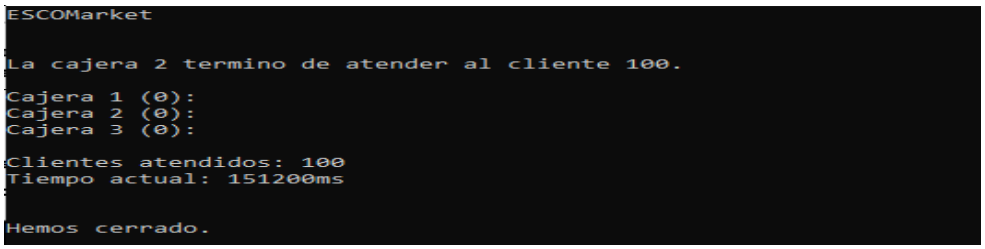
```
ESCOMarket


Llego el cliente 9 a la caja 2.

Cajera 1 (0):
Cajera 2 (2): 7 <-- 9
Cajera 3 (0):

Clientes atendidos: 9
Tiempo actual: 13500ms
```

Por último, la simulación finaliza cuando se atendieron 100 clientes y ya no hay ninguno formado, avisa todo el tiempo que se ejecutó la simulación y avisa que se cierra el supermercado:



Errores detectados:

En la implementación de gráficos, a todo el equipo se nos hizo un poco laboriosa esa parte y fue una de las que nos llevo más tiempo, por lo mismo fue que al programa se agregaron gráficos algo comunes, es decir que no eran muy demostrativos, en pocas palabras muy sencillos, al igual que al hacer pruebas con las cajeras, nos dimos cuenta que el programa no acepta 10 cajeras (o más), solo llega hasta el punto en el que se piden los tiempos de cada cajera y el de llegada de los clientes, desde ahí el programa truena y se cierra automáticamente.

Posibles Mejoras:

Trataremos de implementar unos gráficos diferentes, que hagan que este programa se distinga de los demás y con esto tengan algo único y que lo diferencie, además que haremos la corrección para que el programa acepte 10 cajeras o más y que funcione correctamente como lo hace con menos cajeras, con esto creo que esta simulación quedaría muy bien y estaría lista para seguir haciendo pruebas con ella.

Funcionamiento:

Simulación 2: Ejecución de procesos en el sistema operativo.

Primeramente, la compilación del programa y la ejecución de ".exe"

```
C:\Users\USER\Desktop\CODIGOS\hoa7\practica 2> gcc programa2.c ColaDin.c presentacionWin.c -o Simulacion2

C:\Users\USER\Desktop\CODIGOS\hoa7\practica 2>Simulacion2.exe

Desea encolar un programa? (s/n):
```

Al ejecutar el programa nos pide si queremos encolar un programa, al decir que si, nos pedirá el nombre del programa, su actividad, el ID del programa y el tiempo (en segundos):

```
C:\Users\USER\Desktop\CODIGOS\hoa7\practica 2>Simulacion2.exe

Desea encolar un programa? (s/n): s

Nombre del programa: Word
Actividad del programa: Editor de texto
ID: 001
Tiempo: 5
```

Después de encolar todos los programas que quiera el usuario, comienza la simulación:

```
Proceso en Ejecucion
Nombre del programa: Word
ID: 001
Actividad: Editor de texto
Tiempo total que lleva ejecutandose: 8 segundos

Ultimo proceso
ID: 003          Nombre: Power Point
Tiempo restante: 1 segundos

Proceso siguiente
ID: 002          Nombre: Excel
Tiempo restante: 5 segundos
```

Al final nos aparece que los programas se ejecutaron de manera correcta, junto con el tiempo que tardo para finalizar el programa:

```
Programas ejecutados exitosamente :D

Nombre: Power Point      ID: 003
Tiempo usado para finalizar el programa: 12 segundos

Nombre: Word     ID: 001
Tiempo usado para finalizar el programa: 17 segundos

Nombre: Excel    ID: 002
Tiempo usado para finalizar el programa: 22 segundos
```

Errores detectados:

Al hacer pruebas antes de la entrega de la práctica, nos dimos cuenta que cuando no se le ingresa tiempo al programa en su ejecución (cosa que se hizo por accidente y ahí se encontró el error), automáticamente se le asigna un tiempo (sin tomar en cuenta los tiempos de los demás programas, si es que se les asigno), pero es muy elevado y por lo tanto la simulación tardara mucho en terminar, es algo que no debería de suceder.

Posibles Mejoras:

La solución a este error es hacer que el programa no se comience a ejecutar si es que se salta la parte del tiempo, ya que el usuario por error podría dar un enter y el programa se saltaría a la parte donde te pregunta si se desea encolar un nuevo programa y no habría vuelta atrás, por eso es que debemos de agregar esa parte, de lo contrario la simulación se haría demasiado larga y no le serviría al usuario.

Funcionamiento:

Simulación 3: Banco.

Primeramente, la compilación del programa y la ejecución de ".exe"

```
C:\Users\USER\Desktop\CODIGOS\hoa7\practica 2> gcc programa3.c ColaDin.c presentacionWin.c -o Simulacion3

C:\Users\USER\Desktop\CODIGOS\hoa7\practica 2>Simulacion3.exe
Introduce el numero de cajeros disponibles:
```

Al ejecutar el programa nos pide el numero de cajeros disponibles, el tiempo de atención, el tiempo de llegada de los clientes, usuarios y preferentes (todos los tiempos son en milisegundos):

```
C:\Users\USER\Desktop\CODIGOS\hoa7\practica 2>Simulacion3.exe
Introduce el numero de cajeros disponibles:  3
Introduce el tiempo de atencion: 2000
Introduce el tiempo de llegada de los clientes: 3000
Introduce el tiempo de llegada de los usuarios: 2500
Introduce el tiempo de llegada de los preferentes: 2000
```

Después comienza la simulación del programa, nos muestra los clientes, usuarios y preferentes formados, al igual que los cajeros que están ocupados y con qué tipo de cliente:

```
Informacion del banco

Llego un usuario con ID 15.

Cola de preferentes (0):

Cola de clientes    (0):

Cola de usuarios    (0):

Cajero 1: ocupado: (ID: 13, tipo: preferente)
Cajero 2: ocupado: (ID: 14, tipo: cliente)
Cajero 3: ocupado: (ID: 15, tipo: usuario)

Tiempo actual: 12500ms
```

Al finalizar nos dice cuanto tiempo tardo en ejecutarse todo el proceso

Errores detectados:

En este encontramos 2 errores, el primero es que si se ponen una gran cantidad de cajeros es posible que no se ocupen todos, pero al igual depende del tiempo de llegada de los clientes, usuarios y preferentes, así que esta parte no seria tanto un error, ahora el error principal que detectamos al hacer las pruebas finales, es que si se le pone un gran cantidad de tiempo de llegada, el programa se tarda muchísimo y literalmente nunca acaba su ejecución (se podría decir que es infinita ya que atiende a miles de clientes).

Posibles Mejoras:

Tenemos que analizar los códigos y ver en que parte es donde esta la falla, ya que de lo contrario cuando el usuario quiera hacer simulaciones con cantidades de tiempo más grandes el programa se le haría inservible ya que se atenderían a miles de clientes y nunca terminaría, por lo que con esta corrección el programa puede hacer cualquier simulación que desee el usuario.

CONCLUSIONES

Conclusión Buendía Velazco Abel:

Con el desarrollo de esta práctica quedo más más en claro la definición y el funcionamiento de la estructura de datos “cola” y la importancia que tiene en la programación además que se entiende más el concepto “FIFO” y vemos su importancia, tanto en estas simulaciones, como en más aplicaciones de la programación y nuestra vida diaria.

La realización de esta práctica fue más compleja que la practica anterior, pero al igual que la pasada fue muy interesante hacerla, aunque lo que como equipo nos costó mucho más trabajo fue la implementación de los gráficos al primer programa, ya que como se mencionó en el apartado de errores no teníamos mucha idea de cómo hacerlo y por lo mismo lo implementamos de una manera sencilla, al igual que nos dimos cuenta que las simulaciones que se dejaron prácticamente son situaciones que se viven día a día de manera cotidiana y nos sirve para entender de mejor manera la estructura de datos cola y de la misma manera vemos que es muy importante en el mundo de la programación, ya que lo podemos implementar en una gran cantidad de programas para simular un sinfín de aplicaciones, me gustó mucho esta práctica y espero que sigamos haciendo más simulaciones de este tipo de aplicaciones.

Conclusión Carpio Becerra Erick Gustavo

En esta práctica pude darme cuenta de la importancia de las colas tanto en el mundo de la programación, como en la vida cotidiana, pues en realidad es un concepto con el que tratamos a diario cuando vamos prácticamente a cualquier lugar donde se pueden comprar cosas (el primero que llega es el primero que se atiende, y se van formando conforme van llegando). Resulta interesante como un una estructura tan común y natural para el humano puede trasladarse a un plano informático, en el que toma partido en situaciones como el orden de ejecución de procesos con ayuda de la implementación de prioridades, las cuales son determinadas por el sistema operativo. De igual forma, creo que es de relevancia la inclusión de prioridades en las colas, pues es esta característica la que las hace imprescindibles en el ámbito de la computación, y es por esto que

resulta interesante las distintas opciones de implementación que se muestran en la respectiva presentación de esta estructura de datos en la plataforma.

Si bien las tres implementaciones de cola solicitadas en esta práctica en realidad no fueron tan complicadas, siento que el verdadero reto constó de incorporar los gráficos en la consola (de Windows en nuestro caso), y a pesar de que nuestro resultado final fue una animación muy simple, creo que fuimos capaces de darle solución al problema planteado

Conclusión Hernández Molina Leonardo Gaell:

Esta práctica ha sido de las más complicadas que hemos realizado debido a que la estructura de datos cola a mi parecer es una de las más complejas y laboriosas de implementar a la hora de solucionar un problema, el cual si lo pensamos más a detalle la cola solo es una como una lista, pero por ambos lados, casi como una lista doblemente ligada.

Por otro lado, la implementación de los programas fue mucho de pensarse porque había muchos casos los cuales atender, que si clientes con prioridad, que si se atienden tantos o cuantos clientes, entre muchos otros problemas que fuimos solucionando en el camino han hecho que esta sea una de las prácticas más complicadas.

Por otro lado, el desarrollarlo ha sido de mucha ayuda para poder entender muchas otras cosas que no entendía o que ignoraba por completo, utilizamos funciones más útiles, librerías más directas para trabajar con cadenas entre muchas otras cosas las cuales nos facilitaran la abstracción.

Conclusión Velázquez Díaz Luis Francisco.

A mi parecer, es la práctica más difícil que hemos realizado hasta la fecha, fue un desafío enorme lograr la implementación de la cola en las 3 simulaciones solicitas, considero que la “COLA” es la estructura de datos más útil en la vida cotidiana, ya que, su implementación puede darse en muchos ámbitos.

Practica 3

Esta práctica me ayudó a darme cuenta de que, aunque un problema se vea sencillo de resolver, buscar una solución apta es bastante difícil, pero también me agrada que cada vez vamos viendo más escenarios en los que la programación de todo tipo es muy necesaria en la vida cotidiana.

Anexos

Códigos fuente

Simulación 1

```
/*
programa1.c  (SuperMercado)
V 1.0   Diciembre 2022
-------------------------------------------------------
Autores: Los t▯os pelones
        Grupo: 2CM1
        Buendia Velazco Abel
        Carpio Becerra Erik Gustavo
        Hernandez Molina Leonardo Gaell
        Velazquez Diaz Luis Francisco
-------------------------------------------------------
DESCRIPCI▯N:
        Programa que simula la atenci▯n de clientes en un supermercado.
        -Contamos con n cajeras disponibles para atenderlos, cada una cuenta con
        su propio tiempo de ateci▯n.
        -Los clientes llegan cada cierto tiempo, se forman al azar en alguna caja
        y se quedan ah▯ hasta que son atendidos.
        -Si ya se atendieron al menos 100 clientes y no hay clientes en las cajas,
        se cerrar▯ el supermercado.
-------------------------------------------------------

Compilacion:
        Windows: gcc programa1.c ColaDin.c presentacionWin.c -o programa1.exe  (SuperMercado)
        Windows: gcc programa1.c ColaDin.c presentacionWin.c -o programa1      (SuperMercado)

*/

//Librerias a utilizar para la ejecucuion del programa
#include "ColaDin.h"
#include <stdio.h>
#include "presentacion.h"
#include <time.h>
#include <stdlib.h>

/*
Definimos una estructura simulaci▯n:
```

```
-char nombre[100]: string que contiene el nombre del supermercado
-int n: entero que contiene el n?mero de cajeras disponibles
-int tiempos_cajeras[9]: Timepo de atencion de cada cajera
-cola cajeras[9]: las n filas de la simulaci?n
-int tiempo_cliente: cada cu?ndo llega un nuevo cliente
*/
typedef struct
{
    char nombre[100];
    int n;
    int tiempos_cajeras[9];
    cola cajeras[9];
    int tiempo_cliente;
} simulacion;

// Protipos de funciones

int Random(int minimo, int maximo);
void FormaCliente(int n_cliente, int cajera, simulacion * S);
int EjecutaCajera(int cajera, simulacion * S);
int CajerasVacias(simulacion * S);

//Programa principal
int main(){
    //Declaramos una simulaci?n
    simulacion S;

    //Contadores auxiliares
    int i, j;

    //Variables para controlar el tiempo

    int tiempo_actual = 0;
    int tiempo_base = 10;

    //Datos de cada cliente
```

```
73      int n_cliente = 0;
74      int ID;
75      elemento cliente;
76      int cajera_escogida;
77
78      //Estas variables sirven para llevar el control de los mensajes de la simulacion
79      char mensajes[10][100];
80      int m;
81      int refrescar = 1;
82
83      //Semilla que genera numeros aleatorios en base la hora de nuestro equipo
84      srand(time(0));
85
86      //Leemos los datos de la simulaci?n
87      printf("Introduce el nombre del supermercado: ");
88      fgets(S.nombre, 100, stdin);
89      printf("Introduce el numero de cajeras: ");
90      scanf("%d", &S.n);
91      for(i = 0; i < S.n; i++){
92          printf("Introduce el tiempo de atencion de la cajera %d: ", i + 1);
93          scanf("%d", &S.tiempos_cajeras[i]);
94          Initialize(&S.cajeras[i]);
95      }
96      printf("Introduce el tiempo de llegada de los compradores: ");
97      scanf("%d", &S.tiempo_cliente);
98
99      //Ejecutamos la simulaci?n de manera indefinida
100     while(TRUE){
101         //Esperamos el tiempo base e incrementamos el tiempo actual
102         EsperarMiliSeg(tiempo_base);
103         tiempo_actual += tiempo_base;
104         m = 0;
105
106         //Lleg? un cliente
107         if(tiempo_actual % S.tiempo_cliente == 0){
108             //El cliente escoger? una caja al azar
```

```
109             cajera_escogida = Random(0, S.n - 1);
110             n_cliente++;
111             FormaCliente(n_cliente, cajera_escogida, &S);
112             sprintf(mensajes[m++], "Llego el cliente %d a la caja %d.\n", n_cliente, cajera_escogida + 1);
113             refrescar = 1;
114         }
115
116         //Revisamos las cajas a ver si a alguna le toca finalizar de atender
117         for(i = 0; i < S.n; i++){
118             //La (i+1)-ésima cajera ha finalizado de atender al cliente actual
119             if(tiempo_actual % S.tiempos_cajeras[i] == 0){
120                 ID = EjecutaCajera(i, &S);
121                 if(ID != -1){
122                     //Se atendió correctamente a un cliente
123                     sprintf(mensajes[m++], "La cajera %d termino de atender al cliente %d.\n", i + 1, ID);
124                     refrescar = 1;
125                 }
126             }
127         }
128
129         //Imprimimos la información de la simulación, solo si hubo algun cambio
130         if(refrescar){
131             BorrarPantalla();
132             printf("%s\n\n", S.nombre);
133
134             for(j = 0; j < m; j++)
135                 printf("%s", mensajes[j]);
136             printf("\n");
137
138             for(i = 0; i < S.n; i++){
139                 printf("Cajera %d (%d): ", i + 1, Size(&S.cajeras[i]));
140                 for(j = 1; j <= Size(&S.cajeras[i]); j++){
141                     if(Size(&S.cajeras[i]) > 10 && j >= 6 && j <= (Size(&S.cajeras[i]) - 5)){
142                         printf(" <-- ...");
143                         j = Size(&S.cajeras[i]) - 5;
144                     }else{
```

```
145                         if(j > 1)
146                             printf(" <-- ");
147                         cliente = Element(&S.cajeras[i], j);
148                         printf("%d", cliente.ID);
149                     }
150                 }
151                 printf("\n");
152             }
153             printf("\nClientes atendidos: %d\nTiempo actual: %dms\n\n\n", n_cliente, tiempo_actual);
154             refrescar = 0;
155         }
156
157         //Si ya atendimos al menos a 100 clientes y todas las cajeras están vacías, cerramos
158
159         if(n_cliente >= 100 && CajerasVacias(&S)){
160             printf("Hemos cerrado.\n");
161             break;
162         }
163     }
164
165     return 0; //FIN
166 }
167
168 //Funciones
169
170 /*
171     Aleatorio (aleatorio): recibe <- int minimo; recibe <- int maximo; retorna -> int; retorna -> rand;
172     aleatorio(minimo, maximo)
173     Efecto: dados dos enteros, mínimo y máximo, devuelve un número
174     aleatorio en el intervalo [minimo, maximo]
175
176 */
177 int Random(int Min, int Max){
178     return rand() % (Max - Min + 1) + Min;
179 }
180
```

```
/*
    Formar Cliente (FormarCliente): recibe <- int n_cliente; recibe <- int cajera; recibe <- simulacion * S;
    FormarCliente(n_cliente, cajera,* S)
    Efecto: dado un ID de un cliente, un ndice de una cajera y
    una simulacin S, se encola el cliente a la cajera correspondiente

*/
void FormaCliente(int n_cliente, int cajera, simulacion * S){
    elemento cliente;
    cliente.ID = n_cliente;
    Queue(&S->cajeras[cajera], cliente);
    return;
}

/*
    Procesar Cajera (ProcesarCajera): recibe <- int cajera; recibe <- simulacion *S; retorna -> int -1;
    ProcesarCajera(cajera, *S)
    Efecto: dado un ndice de una cajera y una simulacin, se desencola
    al cliente que est hasta enfrente si la cola especificada no est vaca.

*/
int EjecutaCajera(int cajera, simulacion * S){
    if(!Empty(&S->cajeras[cajera])){
        return Dequeue(&S->cajeras[cajera]).ID;
    }
    return -1;
}

/*
    Cajeras Vacias (CajerasVacias): recibe <- simulacion *S; retorna -> int 1;
    CajerasVacias(* S)
    Efecto: dada una simulacin, determina si todas las cajeras estn vacas.

*/
int CajerasVacias(simulacion * S){
    int i;
```

```
    for(i = 0; i < S->n; i++){
        if(!Empty(&S->cajeras[i]))
            return 0;
    }
    return 1;
}

```

INSTITUTO POLITÉCNICO NACIONAL

ESCOM

Practica 3

SIMULACIÓN 2

```c
/*
programa2.c    (Procesos)
V 1.0   Diciembre 2022
-----------------------------------------------------
Autores: Los t□os pelones
        Grupo: 2CM1
        Buendia Velazco Abel
        Carpio Becerra Erik Gustavo
        Hernandez Molina Leonardo Gaell
        Velazquez Diaz Luis Francisco
-----------------------------------------------------
DESCRIPCI□N:
        Programa que emula la gestion de programas de un sistema operativo.
-----------------------------------------------------

Compilacion:
        Windows: gcc programa2.c ColaDin.c presentacionWin.c -o programa2.exe   (Procesos)
        Windows: gcc programa2.c ColaDin.c presentacionWin.c -o programa2       (Procesos)

*/

//Librerias a utilizar para la ejecucuion del programa
#include <string.h>
#include <stdio.h>
#include "ColaDin.h"
#include "presentacion.h"

/*  el elemento e esta conformado por su nombre, actividad (lo que hace), su ID,
    tiempo propuesto para la ejecucion del programa y un contador que se usara
    mas adelante (En la funcion procesar, para ser exactos).    */

//  Redefinimos el tipo elemento a programa para entrar mas en contexto.
typedef elemento programa;

// Protipos de funciones

void prueba (cola *c);
void MuestraActual (programa p, int espera);
void MuestraUltimo (programa p);
void MuestraSiguiente (programa p);
void mostrarTerminado (programa p);
void mostrarFinalizados (cola *c);
void ProcesaCola (cola *c);
void ProtFunc (char *s, int lim);
void PideDatos (cola *c);

/*  Manda a llamar a la funcion pedirDatos (), forma los programas en la cola c.
    Si la cola esta vacia, muetra un mensaje "No hay nada que mostrar", si no,
    procede a ejecutar el algoritmo propuesto para la practica 2, programa 2.   */
int main (void){
    cola c;
    Initialize (&c);
    PideDatos (&c);

    if (Empty (&c)){
        puts ("No hay nada que mostrar.");
        Destroy (&c);
        return 0;
    }
    ProcesaCola (&c);
    Destroy (&c);
    return 0;
}

/*
    Mostrar Actual (mostrarActual): recibe <- programa p; recibe <- int espera
    MuestraActual(p, espera)
    Efecto: Dado un programa p y un tiempo de espera, esta funcion imprime los campos del programa
    y su tiempo de espera.
*/
void MuestraActual (programa p, int espera){
    puts ("\nProceso en Ejecucion");
```

```
73      printf ("Nombre del programa: %s \n", p.Nombre);
74      printf ("ID: %s \n", p.id);
75      printf ("Actividad: %s \n", p.Act);
76      printf ("Tiempo total que lleva ejecutandose: %d segundos \n", p.contador + espera);
77      return;
78  }
79
80  /*
81      Mostrar Ultimo(mostrarUltimo): recibe <- programa p;
82      MuestraUltimo(p)
83      Efecto: Muestra el ID y nombre del ultimo programa, junto con su tiempo restante para culminar
84      su ejecucion. Recibe el ultimo programa formado en la cola.
85  */
86  void MuestraUltimo (programa p){
87      puts ("\nUltimo proceso");
88      printf ("ID: %s \t Nombre: %s \n", p.id, p.Nombre);
89      printf ("Tiempo restante: %d segundos \n", p.tiempo - p.contador);
90      return;
91  }
92
93  /*
94      Mostrar Siguiente (mostrarSiguiente): recibe <- programa p;
95      MuestraSiguiente p)
96      Efecto: Muestra el programa que esta en el frente de la cola, con su tiempo que hace falta para
97      que se culmine su ejecucion.
98      */
99  void MuestraSiguiente (programa p){
100     puts ("\nProceso siguiente");
101     printf ("ID: %s \t Nombre: %s \n", p.id, p.Nombre);
102     printf ("Tiempo restante: %d segundos \n", p.tiempo - p.contador);
103     return;
104 }
105
106 /*
107     Mostrar Terminado (mostrarTerminado): recibe <- programa p;
108     mostrarTerminado(p)
```

```
109     Efecto: Imprime el nombre de el programa que ya finalizo.
110 */
111 void mostrarTerminado (programa p){
112     printf ("\nEl programa %s se ha acabado de ejecutar. \n", p.Nombre);
113     return;
114 }
115
116 /*
117     Mostrar Finalizados (mostrarFinalizados): recibe <- cola *c
118     mostrarFinalizados(*c)
119     Efecto: Dada una cola, en la cual se fueron formando los programas conforme se
120     acabaron de ejecutar, esta funcion imprime sus campos y su tiempo total
121     que se empleo para su ejecucion.
122 */
123 void mostrarFinalizados (cola *c){
124     programa p;
125
126     puts ("\nProgramas ejecutados exitosamente :D");
127     while (!Empty (c)){
128         p = Dequeue (c);
129         printf ("\nNombre: %s \t ID: %s \n", p.Nombre, p.id);
130         printf ("Tiempo usado para finalizar el programa: %d segundos \n", p.tiempo);
131     }
132     Destroy (c);
133     return;
134 }
135
136 /*
137     Procesar (procesar): recibe <- cola *c;
138     ProcesaCola(*c)
139     Efecto: Funcion que procesa una cola de programas conforme el siguiente criterio:
140     Se desencolara el programa p y se incrementara su contador ya definido en
141     su campo, si el contador es menor al tiempo propuesto por el usuario, se
142     volvera a encolar el programa p, si no, se formara en la cola "Finalizados",
143     la cual se enviara a la funcion de aqui arriba.
144 */
```

Practica 3

```
145  void ProcesaCola (cola *c){
146      programa p;
147      cola Finalizados;
148      int Segundos = 1000, TiempoEspera = 0;
149
150      Initialize (&Finalizados);
151
152      while (!Empty (c)){
153          p = Dequeue (c);
154          MuestraActual (p, TiempoEspera);
155
156          if (Size (c) >= 1){
157              MuestraUltimo (Final (c));
158              MuestraSiguiente (Front (c));
159          }
160          p.contador++;
161          TiempoEspera++;
162
163          if (p.contador < p.tiempo)
164              Queue (c, p);
165          else {
166              mostrarTerminado (p);
167              p.tiempo += TiempoEspera;
168              Queue (&Finalizados, p);
169          }
170          EsperarMiliSeg (1 * Segundos);
171          BorrarPantalla ();
172      }
173      mostrarFinalizados (&Finalizados);
174      return;
175  }
176
177  /*
178      ProtFunc(ProtFunc): recibe <- char *s; recibe <- int lim;
179      ProtFunc (*s, lim)
180      Efecto: Funcion para recibir datos por el teclado.
```

```
181     En comparacion con fgets(), esta pone un
182     fin de cadena al final de cada string (\0)
183 */
184
185 void ProtFunc (char *s, int lim){
186     char c;
187     int i = 0;
188
189     while ((c = getchar ()) != '\n' && i < lim){
190         s[i++] = c;
191     }
192     s[i] = '\0';
193     return;
194 }
195
196 /*
197     Pide Datos (Pide Datos): recibe <- cola *c;
198     PideDatos(*c)
199     Efecto: Funcion destinada a preguntar los campos del programa al usuario y formarlos
200     en la cola c.
201 */
202 void PideDatos (cola *c){
203     programa p;
204     char aux[9];
205
206     printf ("\nDesea encolar un programa? (s/n): ");
207     ProtFunc (aux, 9);
208     if (aux[0] == 'n' || aux[0] == 'N')
209         return;
210     while (TRUE){
211         printf ("\nNombre del programa: ");
212         ProtFunc (p.Nombre, 45);
213         printf ("Actividad del programa: ");
214         ProtFunc (p.Act, 200);
215         printf ("ID: ");
216         ProtFunc (p.id, 45);
```

```
217             printf ("Tiempo: ");
218             ProtFunc (aux, 9);
219             sscanf (aux, "%d", &p.tiempo);
220             p.contador = 0;
221             Queue (c, p);
222             printf ("Desea encolar otro programa? (s/n): ");
223             ProtFunc (aux, 9);
224             if (aux[0] == 'n' || aux[0] == 'N'){
225                 BorrarPantalla ();
226                 return;
227             }
228         }
229         return;
230     }
```

Simulación 3

```
1   /*
2   programa3.c    (Banco)
3   V 1.0   Diciembre 2022
4   ------------------------------------------------------
5   Autores: Los tios pelones
6           Grupo: 2CM1
7           Buendia Velazco Abel
8           Carpio Becerra Erik Gustavo
9           Hernandez Molina Leonardo Gaell
10          Velazquez Diaz Luis Francisco
11  ------------------------------------------------------
12  DESCRIPCION:
13          Programa que simula la llegada de personas a un banco.
14          El banco consta de tres colas para distintos tipos de personas:
15          -cola 0: preferentes
16          -cola 1: clientes
17          -cola 2: usuarios
18          Contamos con n cajeros disponibles, y los tiempos de llegada y de atencion son fijos
19          durante la simulacion.
20          El orden de preferencia es: preferentes > clientes > usuarios.
21          Sin embargo, no se debe permitir que pasen mas de 5 preferentes o clientes sin que un
22          usuario en espera sea atendido.
23  ------------------------------------------------------
24
25  Compilacion:
26          Windows: gcc Banco.c ColaDin.c presentacionWin.c -o programa3.exe    (Banco)
27          Windows: gcc Banco.c ColaDin.c presentacionWin.c -o programa3        (Banco)
28
29  */
30
31
32  //Librerias a utilizar para la ejecucion del programa
33  #include "ColaDin.h"
34  #include <stdio.h>
35  #include "presentacion.h"
36
```

INSTITUTO POLITÉCNICO NACIONAL

ESCOM

Practica 3

```
/*
    Definimos a un cajero, podremos saber si está ocupado o libre;
    y la persona atendida actualmente en caso de que esté ocupado
*/
typedef struct{
    boolean ocupado;
    elemento persona;
} cajero;

/*
    Definimos a la simulación:
    -n_cajeros: cuántos cajeros hay disponibles para atender a las personas
    -tiempo_atencion: cada cuándo se desocupan todos los cajeros
    -tiempo_clientes: cada cuándo llegan los clientes
    -tiempo_usuarios: cada cuándo llegan los usuarios
    -tiempo_preferentes: cada cuánto llegan los preferentes
    -clientes_y_preferentes: cuántos clientes y preferentes han pasado desde el último
     usuario
    -colas[3]: las tres colas de nuestra simulación, 0:preferentes, 1:clientes, 2:usuarios
    -cajeros[9]: los cajeros de nuestra simulacion
*/
typedef struct
{
    int n_cajeros;
    int tiempo_atencion;
    int tiempo_clientes;
    int tiempo_usuarios;
    int tiempo_preferentes;
    int clientes_y_preferentes;
    cola colas[3];
    cajero cajeros[9];
} simulacion;

```

INSTITUTO POLITÉCNICO NACIONAL

ESCOM

Practica 3

```
70
71  // Protipos de funciones
72
73  int LiberaCajeros(simulacion * S);
74  int CajeroEstaLibre(simulacion * S);
75  void FormarPersona(int n_persona, int tipo, simulacion * S);
76  void PasarPersona(int n_persona, int tipo, int donde, simulacion * S);
77  int PuedePasar(int tipo, simulacion * S);
78  void EjecutaPersonaXD(int n_persona, int tipo, simulacion * S);
79  int EjecutaColas(simulacion * S);
80
81  //Programa principal
82  int main(){
83      //Creamos una simulacion e inicializamos en 0 el campo de clientes y preferentes
84      simulacion S;
85      S.clientes_y_preferentes = 0;
86
87      //Variables para controlar el tiempo multiplos de 10
88      int tiempo_actual = 0;
89      int tiempo_base = 10;
90
91      //Contadores auxiliar
92      int i, j;
93
94      //Dato del tipo elemento que nos sera de auxiliar
95      elemento e;
96
97      //Contador para identificar el numero de personas que vayan llegando
98      int n_persona = 0;
99
100     //Estas variables sirven para llevar el control de los mensajes de la simulacion
101     int refrescar = 1;  //Variable auxiliar que nos ayudara a determinar si es que hubo o no un cambio en la expresion para poder actualizar pantalla
102     char mensajes[5][100];  // Nos servira para poder mostrar los mensajes que se vayan generando en pantalla
103
104     //Leemos la entrada de la simulacion
```

INSTITUTO POLITÉCNICO NACIONAL

ESCOM

Practica 3

```
105     printf("Introduce el numero de cajeros disponibles: ");
106     scanf("%d", &(&S)->n_cajeros);  //&S.n_cajeros; A final de cuentas es lo mismo, solo que utilizamos la nomenclatura documentada para "ahorrar" caracteres
107     printf("Introduce el tiempo de atencion: ");
108     scanf("%d", &S.tiempo_atencion);
109     printf("Introduce el tiempo de llegada de los clientes: ");
110     scanf("%d", &S.tiempo_clientes);
111     printf("Introduce el tiempo de llegada de los usuarios: ");
112     scanf("%d", &S.tiempo_usuarios);
113     printf("Introduce el tiempo de llegada de los preferentes: ");
114     scanf("%d", &S.tiempo_preferentes);
115
116     //Inicializamos las tres colas con un forsito
117     for(i = 0; i < 3; i++)
118         Initialize(&S.colas[i]);
119
120     //Para iniciar, debemos asegurarnos que todos los cajeros esten desocupados
121     LiberaCajeros(&S);
122
123     //Inicamos de manera indefinida
124     while(TRUE){
125         EsperarMiliSeg(tiempo_base);
126         tiempo_actual += tiempo_base;   //Avanzara en razon de 10 ms cada vez que el ciclo vuelva a iniciar
127         i = 0;
128
129         //Los cajeros han finalizado de atender a las personas que estaban atendiendo
130         if(tiempo_actual % S.tiempo_atencion == 0){ //Utilizando el modulo determinaremos si ya llego al tiempo maximo de atencion
131             if(LiberaCajeros(&S) > 0)
132                 sprintf(mensajes[i++], "Los cajeros terminaron de atender.\n");
133             if(EjecutaColas(&S) > 0)
134                 sprintf(mensajes[i++], "Pasaron nuevas personas a los cajeros.\n");
135             refrescar = 1;
136         }
137
138         //Llego un preferente
139         if(tiempo_actual % S.tiempo_preferentes == 0){
```

```
140             n_persona++;
141             EjecutaPersonaXD(n_persona, 0, &S); //El 0 indica que es un preferente
142             sprintf(mensajes[i++], "Llego un preferente con ID %d.\n", n_persona);
143             refrescar = 1; //Indica con un 1 si hubo cambio, esto para poder saber mas adelante si debemos actualizar la pantalla
144         }
145
146         //Llego un cliente
147         if(tiempo_actual % S.tiempo_clientes == 0){
148             n_persona++;
149             EjecutaPersonaXD(n_persona, 1, &S);   //El 1 indica que llego un usuario del tipo cliente, por ello le asignamos una prioridad de 1
150             sprintf(mensajes[i++], "Llego un cliente con ID %d.\n", n_persona);
151             refrescar = 1;
152         }
153
154         //Llego un usuario
155         if(tiempo_actual % S.tiempo_usuarios == 0){
156             n_persona++;
157             EjecutaPersonaXD(n_persona, 2, &S);   //El 2 indica que llego un usuario promedio, por lo que le asignamos una prioridad de 2
158             sprintf(mensajes[i++], "Llego un usuario con ID %d.\n", n_persona);
159             refrescar = 1;
160         }
161
162         //Mostramos la informacion de la simulacion, solo si hubo algun cambio
163         if(refrescar){
164             BorrarPantalla();
165             printf("Informacion del banco\n\n");
166             for(j = 0; j < i; j++)
167                 printf("%s", mensajes[j]);
168             printf("\n");
169             for(i = 0; i < 3; i++){
170                 printf("Cola de ");
171                 if(i == 0)
172                     printf("preferentes (%d): ", Size(&S.colas[i]));
173                 else if(i == 1)
174                     printf("clientes    (%d): ", Size(&S.colas[i]));    //Utilizamos este ciclo para imprimir que tipo de cliente es a partir de su "prioridad"
```

INSTITUTO POLITÉCNICO NACIONAL

ESCOM

Practica 3

```
175             else if(i == 2)
176                 printf("usuarios    (%d): ", Size(&S.colas[i]));
177             for(j = 1; j <= Size(&S.colas[i]); j++){    //Este ciclo se va a repetir segun cuantos clientes haya formados en la cola
178                 if(Size(&S.colas[i]) > 10 && j >= 6 && j <= (Size(&S.colas[i]) - 5)){   //Solo mostramos en pantalla hasta 10 clientes, y utilizamos ...<-- Si hay mas
179                     printf("... <-- ");
180                     j = Size(&S.colas[i]) - 5;
181                 }else{
182                     e = Element(&S.colas[i], j);
183                     printf("%d", e.ID);
184                     if(j < Size(&S.colas[i]))
185                         printf(" <-- ");
186                 }
187             }
188             printf("\n\n");
189         }
190         for(i = 0; i < S.n_cajeros; i++){
191             e = S.cajeros[i].persona;
192             printf("Cajero %d: ", i + 1);
193             if(S.cajeros[i].ocupado){
194                 printf("ocupado: (ID: %d, tipo: ", e.ID);
195                 switch(e.tipo){
196                     case 0:{
197                         printf("preferente)");
198                         break;
199                     }
200                     case 1:{
201                         printf("cliente)");
202                         break;
203                     }
204                     case 2:{
205                         printf("usuario)");
206                         break;
207                     }
208                 }
209             }else
```

```
                    printf("desocupado");
                printf("\n");
            }
            printf("\nTiempo actual: %dms\n\n\n", tiempo_actual);
            refrescar = 0;
        }
    }
    return 0; //FIN
}


/*
    Desocupar Cajeros (DesocuparCajeros): recibe <- simulacion *S; retorna -> int cuantos;
    DesocuparCajeros(*S)
    Efecto: Dada una simulación, pone en FALSE la propiedad de todos los cajeros.
*/
int LiberaCajeros(simulacion * S){
    int i, cuantos = 0;
    for(i = 0; i < S->n_cajeros; i++){
        if(S->cajeros[i].ocupado)
            cuantos++;
        S->cajeros[i].ocupado = FALSE;
    }
    return cuantos;
}

/*
    Revisar Cajero Libre (RevisarCajeroLibre): recibe <- simulacion *S; retorna -> -1;
    RevisarCajeroLibre (*S)
    Efecto: Dada una simulación, verifica qué cajero está disponible.
    Si hay varios, devuelve el primero que encuentra. Si no hay ninguno, devuelve -1
*/
int CajeroEstaLibre(simulacion * S){
    int i;
    for(i = 0; i < S->n_cajeros; i++){
        if(S->cajeros[i].ocupado == FALSE){
            return i;
        }
    }
    return -1;
}

/*
    Formar Persona (FormarPersona): recibe <- int n_persona; recibe <- int tipo; recibe <- simulacion *S;
    Efecto: Dada una persona y su tipo de acuerdo a la jerarquía de atención del
    banco, la encola en la cola de acuerdo a su tipo en la simulación.
*/

void FormarPersona(int n_persona, int tipo, simulacion * S){
    elemento persona;
    persona.ID = n_persona;
    persona.tipo = tipo;
    Queue(&S->colas[tipo], persona);
    return;
}
/*
    Pasar Persona (PasarPersona): recibe <- int n_persona; recibe <- int tipo; recibe <- int donde; recibe <- simulacion *S
    PasarPersona (n_persona, tipo, donde, *S)
    Efecto: Dada una persona, su tipo y una posición de cajero válida,
    la pone como la persona atendida en ese cajero y lo pone como ocupado.
*/

void PasarPersona(int n_persona, int tipo, int donde, simulacion * S){
    elemento persona;
    persona.ID = n_persona;
    persona.tipo = tipo;
    S->cajeros[donde].persona = persona;
    S->cajeros[donde].ocupado = TRUE;
    if(tipo == 2)
        S->clientes_y_preferentes = 0;
    else
```

```
281         S->clientes_y_preferentes++;
282     return;
283 }
284 
285 /*
286     Persona Puede Pasar (PerosnaPuedePasar): recibe <- int tipo: recibe <- simulacion *S; retorna -> int disponible;
287     PerosnaPuedePasar (tipo, simulacion, *S)
288     Efecto: Dado un tipo de persona y la simulación, determina si esa persona puede pasar a algún cajero,
289     respetando las políticas del banco. Si sí puede pasar, devuelve la posición del cajero candidato, si no
290     devuelve -1.
291 */
292 int PuedePasar(int tipo, simulacion * S){
293     int disponible = CajeroEstaLibre(S);
294     if(disponible == -1) return -1;
295     if(S->clientes_y_preferentes == 5){
296         if((tipo == 0 || tipo == 1) && !Empty(&S->colas[2]))
297             return -1;
298         return disponible;
299     }
300     return disponible;
301 }
302 
303 /*
304     Procesar Llegada Persona (ProcesarLlegadaPersona): recibe <- int n_persona; recibe <- int tipo; recibe <- si,ulacion *S;
305     ProcesarLlegadaPersona (n_persona, tipo, *S)
306     Efecto: Dada una persona, su tipo y la simulación, determina
307     si la persona podría pasar directamente a algún cajero disponible
308     o se tendría que formar a su cola determinada.
309 */
310 void EjecutaPersonaXD(int n_persona, int tipo, simulacion * S){
311     int pos = PuedePasar(tipo, S);
312     if(Empty(&S->colas[tipo]) && pos != -1){
313         PasarPersona(n_persona, tipo, pos, S);
314     }else{
315         FormarPersona(n_persona, tipo, S);
316     }
317     return;
318 }
319 
320 /*
321     Procesar Colas (ProcesarColas): recibe <- simulacion *S; retorna -> int cuantas;
322     ProcesarColas (*S)
323     Efecto: Esta función se ejecuta cada que es tiempo de atención,
324     y determina si las personas esperando al frente de cada cola pueden pasar a algún cajero.
325     Si sí pueden, las pasa y las desencola.
326 */
327 int EjecutaColas(simulacion * S){
328     int i = 0, pos, cuantas = 0;
329     elemento persona;
330     for(i = 0; i < 3; i++){
331         if(!Empty(&S->colas[i])){
332             persona = Front(&S->colas[i]);
333             pos = PuedePasar(persona.tipo, S);
334             if(pos != -1){
335                 PasarPersona(persona.ID, persona.tipo, pos, S);
336                 Dequeue(&S->colas[i]);
337                 cuantas++;
338             }
339         }
340     }
341     return cuantas;
342 
343 }
```

REFERENCIAS

Cairó, O. y Guardati, S. (2002). Estructuras de Datos, 2da. Edición. McGraw-Hill. Deitel P.J.

Deitel H.M. (2008) Cómo programar en C++. 6ª edición.Prentice Hall. Joyanes, L. (2006).

Programación en C++: Algoritmos, Estructuras de datos y objetos. McGraw-Hill.

Anonimo, "Colas", 1 de mayo de 2010, Programacion avanzada, 1.