Práctica 6

Implementación de estructuras lineales de datos /2

6.1 Reserva dinámica de memoria

Todas las variables utilizadas hasta ahora, sean simples o estructuradas, tienen una característica común: son *estáticas*. Es decir, la memoria necesaria para ubicar la variable ha sido reservada mediante la declaración de la variable antes de comenzar la ejecución del programa. Las variables *dinámicas* se crean y se destruyen durante la ejecución del programa.

A diferencia de las variables estáticas, las variables dinámicas no tienen *nombre*, es decir, no pueden ser referenciadas por un nombre dado por el programador, en vez de eso, las variables dinámicas van a ser referenciadas de forma indirecta a través de un nuevo tipo de variable que denominaremos *puntero*.

```
tipo
    ptro_T: ref T;
ftipo
var
    p:  ptro_T;
fvar
```

La variable estática `p` es un puntero y puede ser utilizada para referenciar a variables dinámicas del tipo `T`. Es posible reservar variables dinámicas del tipo `T` utilizando la operación `reserva`:

```
reserva(p)
```

De esta forma se obtiene una variable (sin nombre) a la que se puede acceder a través de la variable `p`. El contenido de la variable `p` es ahora la dirección en memoria de la variable dinámica obtenida mediante la operación `reserva`.

El operador `ref` aplicado a un puntero da acceso a la variable apuntada por este. Así por ejemplo, en el algoritmo siguiente

```
tipo
    pEntero:  ref entero;
ftipo
var
    p:  pEntero;
```

```
fvar
...
reserva(p);
...
p ref:= 5;
...
p ref:= p ref + 1;
...
```

las asignaciones mostradas modifican el valor de la variable apuntada (referenciada) por `p`, pero no el valor de `p`.

La operación `libera(p)` destruye la variable dinámica apuntada por `p` y devuelve el espacio ocupado por esta, al espacio libre de la memoria, de manera que pueda ser reutilizado para otras variables.

6.2 Colas

Una *cola* es una organización de datos en la que el elemento más antiguo en la cola es el primer elemento a salir.

6.2.1 El TAD Cola

La especificación del tipo abstracto de datos cola es la siguiente:

Tipo: tCola (tElem)
Incluye: booleano
Operaciones:
nuevaCola: $\longrightarrow$ tCola
pideTurno: tCola, tElem $\longrightarrow$ tCola
avance: tCola $\longrightarrow$ tCola
primero: tCola $\longrightarrow$ tElem
esNula: tCola $\longrightarrow$ booleano
Ecuaciones: $\forall$x, y: tElem; $\forall$c: tCola;
primero(pideTurno(pideTurno(c, x), y)) $\equiv$ primero(pideTurno(c, x))
primero(pideTurno(nuevaCola, x)) $\equiv$ x
avance(pideTurno(pideTurno(c, x), y)) $\equiv$
$\equiv$ pideTurno(avance(pideTurno(c, x)), y)
avance(pideTurno(nuevaCola, x)) $\equiv$ cNula
esNula(pideTurno(c, x)) $\equiv$ **falso**
esNula(nuevaCola) $\equiv$ **cierto**
Ecuaciones de error:
avance(nuevaCola) $\equiv$ **error**
primero(nuevaCola) $\equiv$ **error**

6.2.2 El módulo *colas*

A continuación vamos a construir un módulo llamado *colas*, que exporte el tipo abstracto de datos *tCola* visto en la sección anterior. Vamos a realizar una implementación dinámica. Para ello utilizaremos los siguientes tipos.

```
tipo
    celda = tupla
        e:  tElem;
        s:  ref celda;
    ftupla;
    tCola = tupla
        i, f:  ref celda;
    ftupla;
ftipo
```

El campo `i` apunta al elemento más antiguo almacenado en la cola, salvo si la cola es nula que apunta a `nil`. Cada elemento en la cola apunta al introducido inmediatamente después, salvo el último introducido que apunta a `nil`. El campo `f` apunta al elemento último almacenado en la cola, salvo si la cola es nula que apunta a `nil`.

De acuerdo con esta representación, el módulo que encapsula las colas es como sigue

```
modulo colas;
    importa
        //nombre del modulo donde se define el tipo tElem
    fimporta
    exporta
        tipo
            tCola;
        ftipo
            //Modifica:  inicializa la cola c
        accion nuevaCola(sal c:tCola);
            //Entrada:  c una cola de tElem y e un tElem
            //Modifica:  Almacena e como ultimo elemento de c
        accion pideTurno(e/s c:tCola;ent e:tElem);
            //Entrada:  c una cola de tElem
            //Requisitos:  La cola no esta vacia
            //Modifica:  Elimina el elemento mas antiguo de la cola
        accion avance(e/s c:tCola);
            //Entrada:  c una cola de tElem
            //Requisitos:  La cola no esta vacia
            //Salida:  Copia en e elemento mas antiguo de la cola
        accion primero(ent c:tCola;sal e:tElem);
            //Entrada:  c una cola de tElem
            //Salida:  Indica si c tiene elementos
        funcion esNula(p:tCola)dev b:booleano;
    fexporta
    implementacion
        tipo
            tCola = tupla
                i, f:ref celda;
            ftupla;
            celda = tupla
                e:tElem;
                s:ref celda;
            ftupla;
        ftipo
        accion nuevaCola(sal c:tCola);
            c.i:= nil;
            c.f:= nil
        faccion;
        accion pideTurno(e/s c:tCola;ent e:tElem);
```

```
            var
                q:ref celda;
            fvar
            reserva(q);
            q ref.e:= e;
            q ref.s:= nil;
            si c.f = nil ⟶
                c.i:= q
            [] c.f ≠ nil ⟶
                c.f ref.s:= q
            fsi;
            c.f:= q
        faccion;
        accion avance(e/s c:tCola);
            var
                q:ref celda;
            fvar
            q:= c.i;
            c.i:= c.i ref.s;
            si c.i = nil ⟶
                c.f:= nil
            [] c.i ≠ nil ⟶
                continuar
            fsi;
            libera(q)
        faccion;
        accion primero(ent c:tCola;sal e:tElem);
            e:= c.i ref.e
        faccion;
        funcion esNula(c:tCola)dev b:booleano;
            b:= (c.i = nil);
            dev b
        ffuncion;
    fimplementacion
fmodulo
```

Se puede observar que la implementación anterior es extraordinariamente eficaz en lo referente a recursos de tiempo. Todas las operaciones tienen lugar en tiempo constante.

6.2.3 Traducción a lenguaje C

Las reglas generales para pasar un módulo de nuestro lenguaje algorítmico a C se han visto ya en sesiones anteriores, por lo que no volveremos a incidir en ello. El único elemento novedoso es la utilización de reserva dinámica de espacio, por lo que a continuación se muestran las instrucciones de C para la reserva, utilización y liberación de variables dinámicas:

```
tipo
    ptrTElem = ref tElem;          typedef tElem * ptrTElem;
ftipo
var
    p: ptr_tElem;                  ptrTElem p;
fvar
reserva(p)                         p = malloc(sizeof(tElem));
```

libera(p)	`free(p);`
p ref:= p ref + x	`*p = *p + x;`

La operación `malloc(tamaño)` actúa de modo muy similar a la operación `reserva(p)` del lenguaje algorítmico: crea una nueva variable dinámica del tamaño que se le indique. Si no hay memoria disponible `p` toma el valor `NULL`, el equivalente en C al `nil` algorítmico. La memoria que devuelve `malloc` no se inicializa, es decir, contiene valores descontrolados.

La operación `free(p)` actúa como la operación `libera(p)` del lenguaje algorítmico: destruye la variable dinámica apuntada por `p` y devuelve al espacio libre de la memoria, el espacio ocupado por esta, de manera que pueda ser reutilizado para otras variables.

Para poder utilizar ambas funciones debe haberse importado la libreria standard de c, es decir, debe haberse incluido `<stdlib.h>`[1].

Finalmente, vemos la implementación dinámica en C del módulo `colaDeTElems`[2]. El fichero `colaDeTElems.h` sería:

```
#ifndef FFF_COLA_DE_TELEMS_H
#define FFF_COLA_DE_TELEMS_H
#include <stdbool.h> // para el tipo booleano
#include ‘‘TElem.h’’ // para el tipo tElem
typedef struct nodoDeColaDeTElems {
    tElem e;
    struct nodoDeColaDeTElems *s;
} NodoDeColaDeTElems;
typedef struct colaDeTElems{
    NodoDeColaDeTElems *i;
    NodoDeColaDeTElems *f;
} ColaDeTElems;
void nuevaColaDeTElems(ColaDeTElems *);
void pideTurnoColaDeTElems(ColaDeTElems *, tElem);
void avanceColaDeTElems(ColaDeTElems *);
void primeroColaDeTElems(ColaDeTElems, tElem *);
bool esNulaColaDeTElems(ColaDeTElems);
#endif
```

El fichero `colaDeTElems.c` sería:

```
#include <stdlib.h> // para conocer exit y malloc
#include ’’colaDeTElems.h’’ //
void errorColaDeTElems(char s[]){
    printf(’’\n\n\nERROR en el módulo colas:  %s \n’’, s);
    while (true)
        exit(-1);
}
void nuevaColaDeTElems(ColaDeTElems *c){
    c->i = NULL;
    c->f= NULL;
}
void pideTurnoColaDeTElems(ColaDeTElems *c, tElem x){
    NodoDeColaDeTElems * q;
    if ((q=malloc(sizeof(NodoDeColaDeTElems)))==NULL)
        errorColaDeTElems(’’no hay memoria para pideTurno’’);
```

[1] Esta librería contiene otras funciones para gestionar la memoria dinámica. Si quiere saber cuales son y como se usan, escriba `man malloc` en la consola.

[2] No repito las especificaciones

```
    q->e = x;
    q->s = NULL;
    if (c->f == NULL)
        c->i = q;
    else
        c->f->s=q;
    c->f=q;
}
bool esNulaColaDeTElems (ColaDeTElems c){
    return (c.i == NULL);
}
void avanceColaDeTElems(ColaDeTElems *c){
    NodoDeColaDeTElems * q;
    if (esNulaColaDeTElems(*c))
        errorColaDeTElems(''avanzando en cola nula'');
    q= c->i;
    c->i = c->i->s;
    if (c->i == NULL)
        c->f = NULL;
    free(q);
}
void primeroColaDeTElems (ColaDeTElems c, tElem *x){
    if (esNulaColaDeTElems(c))
        errorColaDeTElems(''primero en cola nula'');
    *x = c.i->e; //cuidado con el tipo tElem
}
```

6.3 Actividad a realizar

Diseñar un programa que simule el funcionamiento de las cabinas de un peaje de autopista. Se supone que el peaje consta de cinco cabinas, y que el tiempo necesario para el cobro está distribuido uniformemente, siendo este de entre 15 y 30 segundos para las cabinas 1 y 2, de entre 15 y 45 segundos para las cabinas 3 y 4, y de entre 30 y 60 segundos para la cabina 5. Los coches llegan al peaje de acuerdo a una distribución exponencial, con un tiempo medio entre llegadas de t_c segundos, siendo este un valor a fijar por el usuario en la simulación con objeto de poder representar distintas situaciones de tráfico. Cada cabina del peaje tiene una cola distinta, cuando un coche llega al peaje se incorpora a una de las tres colas con menos coches, con una probabilidad de 0.6 para la más corta, 0.3 para la segunda más corta y 0.1 para la tercera más corta. El programa debe simular el funcionamiento del peaje durante tres horas (10800 segundos) y debe obtener el tiempo medio (en segundos) de espera (incluido el tiempo de cobro) y la longitud (número de coches) máxima de la cola en cada una de las cabinas, el tiempo medio de espera en el peaje y el número total de coches servidos.

Se trata de dar una solución modular correcta, por lo que se propone desarrollar los módulos cuya especificación se da en parte a continuación.

1. Módulo `reloj`. Es la implementación de un reloj que cuenta segundos. Su interfaz es el siguiente:

```
modulo reloj;
    exporta
        tipo
```

```
            Reloj;
        ftipo
            // Salida:  Pone a cero la cuenta del reloj r
        accion aCero(sal r:Reloj);
            // Incrementa en uno la cuenta del reloj r
        accion tic(e/s r:Reloj);
            ///Devuelve el valor en segundos de la cuenta del reloj r
        funcion instante(r:Reloj) dev i:entero;
    fexporta
```

2. Módulo `ruleta`. Contiene una serie de operaciones que permiten generar números aleatorios de acuerdo a distintas distribuciones y probabilidades. Puede tener la forma que se presenta a continuación. Se están usando las funciones `srand` y `rand` para generar números aleatorios. Para usarlas es necesario incluir `stdlib.h`. Para establecer la semilla del generador de números aleatorios se emplea la función `time`, para lo que es necesario incluir `time.h`. Finalmente, para log debe incluir `math.h`. Fijese en lo que hacen estas funciones antes de usarlas y recuerde como funciona el símbolo `/` en C.

```
modulo ruleta;
    exporta
            // Inicia la generacion de numeros aleatorios
        accion iniciarRuleta();
            //implementa una distribucion exponencial de media tmedio
        funcion distribucionExponencial(media:  entero) dev t:  entero;
            // implementa una distribucion lineal entre min y max
        funcion distribucionLineal(min, max:entero)dev t:entero;
            // elige entre tres opciones con prob.  dadas
        funcion eleccionCon3Probabilidades(maxp,medp,minp:real)dev n:entero;
    fexporta
    implementacion
        accion iniciarRuleta();
            srand(time(NULL));
        faccion;
        funcion distribucionExponencial(media:  entero) dev t:entero;
            var
                x:  real;
            fvar
            x:= rand()/RAND_MAX; // numero aleatorio en [0, 1)
            t:= entera(-log(1-x)*media);
            mientras t = 0 hacer
                x:= rand()/RAND_MAX;
                t:= entera(-log(1-x)*media);
            fmientras;
            dev t
        ffuncion;
        funcion distribucionLineal(min, max:entero)dev t:  entero;
            var
                x:  real;
            fvar
            x:= rand()/RAND_MAX;
            t:= entera((max - min) * x) + min;
            dev t
        ffuncion;
        funcion eleccionCon3Probabilidades(maxp,medp,minp:real) dev n:entero;
```

```
            var
                x:  real;
                elec:  tabla[1..3] de real;
            fvar
            elec[1]:=maxp;
            elec[2]:=medp+maxp;
            elec[3]:=1;
            x:= rand()/RAND_MAX;
            n:= 1;
            mientras (elec[n] < x) y (n < 3) hacer
                n:= n+1
            fmientras;
            dev n
        ffuncion;
    fimplementacion
fmodulo
```

3. Módulo `colaDeEnteros`. Es la implementación del TAD Cola cuando los elementos a almacenar son de tipo entero. La especificación de este módulo (y su implementación dinámica) aparecen con anterioridad; recuerde que tElem es entero y llame ColaDeEnteros al tipo exportado.

4. Módulo `cabinas`. Es la implementación del tipo `Cabina` que simula el funcionamiento de una cabina del peaje. La parte de exportaciones del modulo es la que sigue:

```
tipo
    Cabina;
ftipo
    // Inicia la cabina sin coches esperando
accion iniciarCab(e/s cab:  Cabina, ent tmin, tmax:  entero);
    // Pone en el instante r un coche en espera en la cabina
accion encolarCoche(e/s cab:  Cabina, ent r:  Reloj);
    // Da el numero de coches esperando en la cabina
funcion cuantosCoches(ent cab:  Cabina) dev n:entero;
    // Simula el servicio de un coche en la cola de la cabina
    // en el instante r
accion servCabina(e/s cab:  Cabina, ent r:  Reloj);
```

   La implementación de la acciones del módulo pueden tener la forma que se presenta a continuación:

```
tipo
    Cabina = tupla
        nCoches:  entero; // coches esperando en la cabina
        maxCoches:entero; // maximo de coches esperando
        servidos:entero; // coches servidos
        totalEsperado:  entero; // tiempo total de esperas
        proxServ:  entero; // instante proximo servicio
        minServ:  entero; // tiempo mínimo del servicio
        maxServ:  entero; // tiempo máximo del servicio
        colaCoches:  ColaDeEnteros; // cola de coches
    ftupla;
ftipo
accion iniciarCab(e/s cab:  Cabina, ent tmin, tmax:entero);
```

```
    cab.nCoches:= 0; cab.maxCoches:= 0; cab.servidos:= 0;
    cab.totalEsperado:= 0; cab.proxServ:= 0;
    cab.minServ:= tmin; cab.maxServ:= tmax;
    nuevaColaDeEnteros(cab.colaCoches)
faccion;
accion contarCoche(e/s cab:  Cabina,ent r:Reloj);
    si cab.nCoches = 0 ⟶
        cab.proxServ:= instante(r)+distribucionLineal(cab.minServ, cab.maxServ);
    [] cab.nCoches ≠ 0 ⟶
        continuar
    fsi;
    cab.nCoches:= cab.nCoches +1;
    si cab.nCoches > cab.maxCoches ⟶
        cab.maxCoches:= cab.nCoches
    [] cab.nCoches ≤ cab.maxCoches ⟶
        continuar
    fsi
faccion;
accion encolarCoche(e/s cab:  Cabina, ent r:  Reloj);
    contarCoche(cab, r);
    pideTurnoColaDeEnteros(cab.colaCoches, instante(r))
faccion;
funcion cuantosCoches(ent cab:  Cabina) dev n:entero;
    n:= cab.nCoches;
    dev n
ffuncion;
accion servCabina(e/s cab:  Cabina, ent r:Reloj);
    var
        x:entero;
    fvar
    si cab.proxServ ≠ instante(r) ⟶
        continuar
    [] cab.proxServ = instante(r) ⟶
        cab.servidos:= cab.servidos + 1;
        primeroColaDeEnteros(cab.colaCoches, x);
        avanceColaDeEnteros(cab.colaCoches);
        cab.totalEsperado:= cab.totalEsperado + (instante(r) - x);
        cab.nCoches:= cab.nCoches - 1;
        si cab.nCoches = 0 ⟶
            cab.proxServ:= 0
        [] cab.ncoches <> 0 ⟶
            cab.proxServ:= instante(r)+distribucionLineal(cab.minServ,
                    cab.maxServ)
        fsi
    fsi
faccion;
```

5. Módulo `peajes`. Es la implementación del tipo `Peaje` que simula el funcionamiento de un peaje de autopista con `NCAB` cabinas. Exporta lo que se indica a continuación:

```
tipo
    Peaje;
ftipo
    // Inicia el peaje
    accion iniciarPeaje(e/s p:Peaje);
    // Encola un nuevo coche en una cabina
```

```
accion guardaCola(e/s p:Peaje; ent ncab:  entero; ent r:  Reloj);
    // Elige cabina entre las tres menos ocupadas
    funcion eligeCabina(p:Peaje) dev n:entero;
    // Simula una ronda de servicio de las cabinas
accion rondaCabinas(e/s p:Peaje,ent r:Reloj);
```

La implementación de la acciones del módulo pueden tener la forma que se presenta a continuación:

```
const
    NCAB = 5; // numero de cabinas
    tipoCobro:  tabla[1..NCAB] de intervalo = ((min:15, max:30), (min:15,
            max:30), (min:15, max:45), (min:15, max:45), (min:30, max:60));
    PROB_MAX=0.6;
    PROB_MEDIA= 0.3;
    PROB_MIN = 0.1;
fconst
tipo
    Peaje = tabla [1..NCAB] de Cabina;
    pareja = tupla
        can, cab:entero;
    ftupla;
    taux = tabla [1..NCAB] de pareja;
    intervalo = tupla
        min, max:  entero;
    ftupla;
ftipo
accion ordenar(e/s t:taux);
    // implemente alguno de los métodos de ordenación de tablas
faccion;
accion guardaCola(e/s p:  Peaje; ent ncab:  entero, ent r:  Reloj);
    encolarCoche(p[ncab], r)
faccion;
accion iniciarPeaje(e/s p:Peaje);
    var
        i:entero;
    fvar
    para i:= 1 hasta NCAB hacer
        iniciarCab(p[i], tipoCobro[i].min, tipoCobro[i].max)
    fpara
faccion;
funcion eligeCabina(p:Peaje) dev n:entero;
    var
        x, i:entero;
        t:  taux;
    fvar
    para i:= 1 hasta NCAB hacer
        t[i].can:= cuantosCoches(p[i]);
        t[i].cab:= i
    fpara
    ordenar(t);
    x:= eleccionCon3Probabilidades(PROB_MAX, PROB_MEDIA, PROB_MIN);
    n:= t[x].cab;
    dev n
ffuncion;
```

```
accion rondaCabinas(e/s p:  Peaje, ent r:  Reloj);
    var
        i:entero;
    fvar
    para i:= 1 hasta NCAB hacer
        servCabina(p[i], r)
    fpara
faccion;
```

6. Módulo `controles`. Es la implementación del tipo `Control` que simula el funcionamiento de un control de autopista que se compone de un peaje e información sobre el tráfico. El módulo exporta:

```
tipo
    Control;
ftipo
    // Inicia el control
accion iniciarControl(e/s c:  Control);
    // Simula la llegada de un coche al control
accion llegaCoche(e/s c:  Control, ent r:Reloj);
    // Simula la marcha de un coche del control
accion marchaCoche(e/s c:  Control, ent r:Reloj);
```

La implementación de la acciones del módulo pueden tener la forma que se presenta a continuación:

```
tipo
    Frecuencia = tupla
        tLlegada, tMedio:entero;
    ftupla;
    Control = tupla
        peaje:  Peaje;
        trafico:  Frecuencia;
    ftupla;
ftipo
accion iniciarControl(e/s c:  Control);
    // solicitar el tiempo medio entre llegadas (en segundos)
    c.trafico.tLlegada:= distribucionExponencial(c.trafico.tMedio);
    iniciarPeaje(c.peaje)
faccion
accion llegaCoche(e/s c:  Control, ent r:  Reloj);
    var
        n:entero;
    fvar
    si instante(r) = c.trafico.tLlegada ⟶
        n:= eligeCabina(c.peaje);
        guardaCola(c.peaje, n, r);
        c.trafico.tLlegada:= instante(r)+distribucionExponencial(c.trafico.tMedio)
    [] r ≠ c.trafico.tLlegada ⟶
        continuar
    fsi
faccion
accion marchaCoche(e/s c:  Control, ent r:  Reloj);
    rondaCabinas(c.peaje, r)
faccion
```

7. Algoritmo `simPeaje`. Es el programa principal que simula el peaje de la autopista. El algoritmo puede ser el siguiente:

```
// tiempo para la simulación 3 horas = 10800 segundos
const
    TSIM = 10800;
fconst
accion iniciarSimulacion(e/s c:  Control);
    iniciarRuleta();
    iniciarControl(c)
faccion;
accion siguienteIteracion(e/s c:  Control; ent r:  Reloj);
    llegaCoche(c, r);
    marchaCoche(c, r)
faccion;
var
    r:  Reloj;
    c:  Control;
fvar
aCero(r);
iniciarSimulacion(c);
mientras instante(r) ≠ TSIM hacer
    tic(r);
    siguienteIteracion(c, r)
fmientras;
mostrarResultados(c)
```

8. En todo el enunciado faltan algunas acciones y funciones. Se trata de aquellas que permiten implementar `mostrarResultados`. Escriba las acciones y funciones necesarias para dicha acción de acuerdo a los módulos diseñados y a los principios de modularización estudiados. La acción `mostrarResultados` debe dar el tiempo medio (en segundos) de espera y la longitud (número de coches) máxima de la cola en cada una de las cabinas, el tiempo medio de espera en el peaje y el número total de coches servidos.