Soluciones - práctico 4

Multiestructuras

EJERCICIO 5

El ejercicio maneja los siguientes conceptos: pasajero, aeropuerto, vuelo y viaje. Del pasajero se conoce su nombre (el cual lo identifica) y sus reservas. Los aeropuertos son a su vez identificados por un código de tres letras. De los aeropuertos salen los diversos vuelos, los cuales se identifican mediante un número de tres dígitos. Dichos vuelos poseen los aeropuertos de salida y llegada, así como también los respectivos horarios. Un vuelo para una fecha dada identifica un viaje, con una determinada capacidad, un cierto número de reservas y una cola de espera.

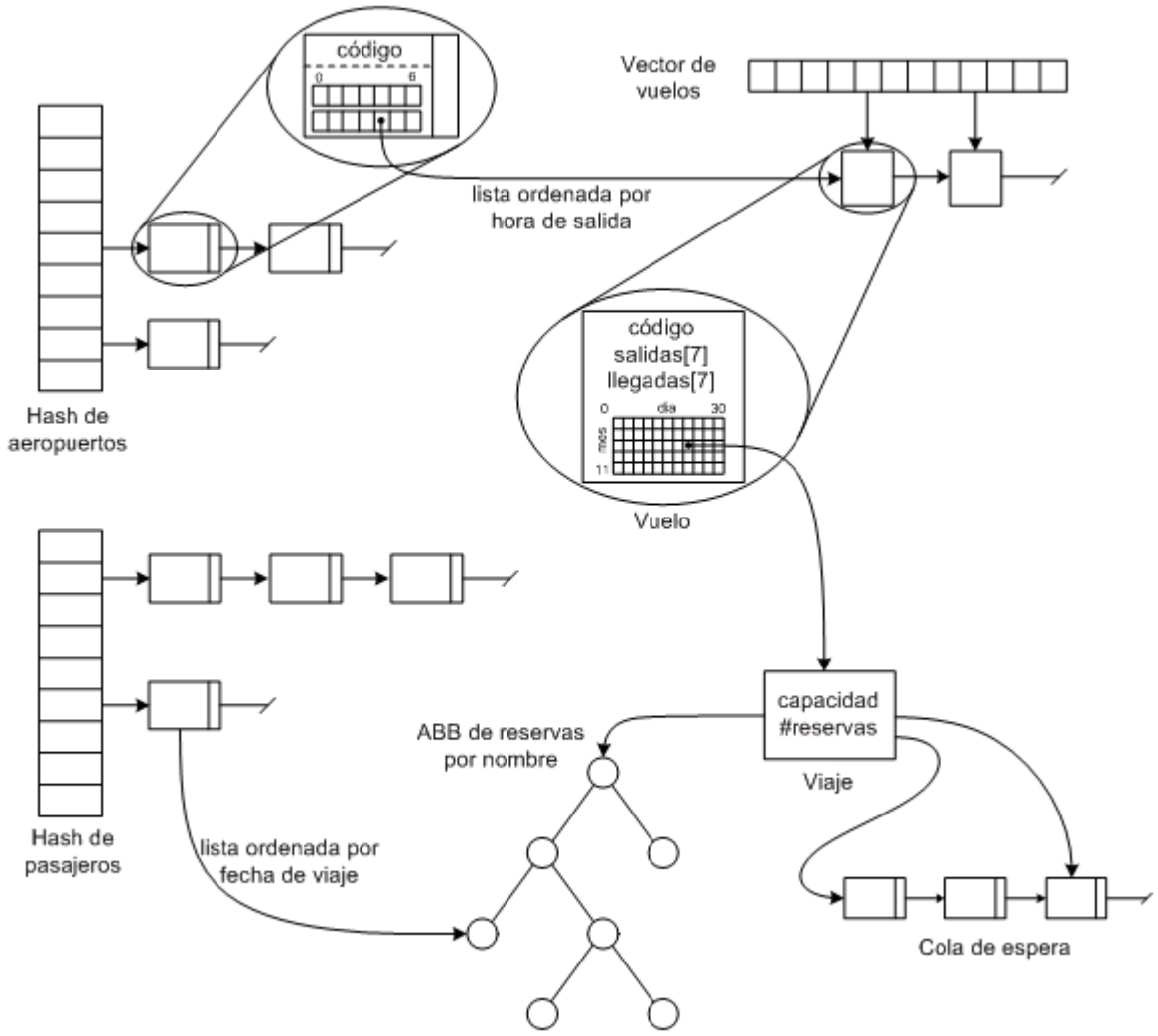
A continuación se muestra como cada operación restringe la estructura, comenzando por aquellas que condicionan más fuertemente la misma:

- **`horarios`** requiere la impresión, para un número de vuelo dado, de todas las salidas semanales que tiene ordenado por día de la semana (indicando horas y aeropuertos de salida y llegada). Este requerimiento debe ser satisfecho en orden constante en el peor caso, de donde resulta la necesidad de un vector de vuelos (obsérvese que son solo números de tres dígitos). A su vez, el nodo correspondiente al vuelo debe almacenar las horas y aeropuertos de salida y llegada, para lo cual se eligen dos arreglos de estructuras (cada una con hora y aeropuerto) para los 7 días de la semana, uno para las salidas y otro para las llegadas.
- **`libre`** requiere la indicación de la disponibilidad de lugares en un viaje dado en orden constante en el peor caso. Dado que un viaje está identificado por un número de vuelo y una fecha, y además el nodo correspondiente al vuelo es alcanzable a través de un array (ver operación anterior), solo resta identificar el viaje mediante la fecha. Para ello se utiliza, dentro de cada vuelo, una matriz de viajes donde se codifican los meses como filas y los días como columnas. De esta forma, el acceso a un viaje dados el vuelo y la fecha puede ser realizado en orden constante en el peor caso. A modo de completar el requerimiento, la capacidad así como también el número actual de reservas, deben ser mantenidos en el nodo del viaje.
- **`pasajeros`** requiere la impresión, en orden lineal en la capacidad del viaje en el peor caso, de todos los pasajeros con lugar reservado en dicho viaje. Dado que dicho listado debe estar ordenado alfabéticamente, el almacenamiento de las reservas en cada viaje se realiza en un ABB de reservas por nombre del pasajero.
- **`queViajes`** debe imprimir en orden constante promedio, la lista de viajes que un pasajero dado tiene reservados. Esto resulta en la necesidad de una estructura de hash por nombre del pasajero, y una lista de reservas a nivel del mismo. Puesto que la impresión debe estar ordenada por fecha de vuelo, se exige a la lista la misma condición (obsérvese que existen 10 reservas como máximo). Para cada una de las mencionadas reservas se debe mantener el viaje correspondiente (vuelo y fecha) y el estado de la misma (efectiva o en espera). Dado que a través de vuelo y fecha pueden accederse aeropuertos y horarios en orden constante, dicha información no es necesaria en estos nodos.
- **`asignacion`** debe, dados el viaje y un pasajero, intentar hacer una reserva para el pasajero en dicho viaje o, en caso de no haber lugar, ponerlo en la cola de espera. El acceso tanto al árbol de reservas como a la lista de espera debe realizarse a través del viaje, al cual puede llegarse en orden constante en el peor caso (ver más arriba). Si existe lugar (verificable en orden constante), debe darse el alta de la nueva reserva al árbol de

reservas, lo cual puede realizarse en orden logarítmico en promedio. En caso contrario, el pasajero debe ser enviado a la lista de espera, para lo cual es necesario un enlace al último elemento (una recorrida no es capaz de garantizar el orden pedido). Por último, debe agregarse la reserva (ya sea efectiva o en lista de espera) a la lista de reservas del pasajero. Dado que este último puede ser accedido en orden constante en promedio y la cantidad de reservas por pasajero es acotada, la inserción ordenada por fecha no modifica el orden total de la operación.

- **`cancelacion`** requiere borrar la reserva de un pasajero para un viaje dado, en orden logarítmico promedio. Dado que se puede asumir que dicho pasajero no se encuentra en lista de espera, y el acceso al nodo del viaje puede realizarse en orden constante peor caso, el borrado de la reserva está dado justamente por un procedimiento de orden logarítmico. Por otra parte, dado el número limitado de reservas por pasajero, la baja de la reserva en la lista del pasajero no incrementa dicho orden. A su vez, el primer pasajero en la lista de espera está siendo apuntado desde el mismo viaje, y en caso de haber uno, éste es movido al árbol (nuevamente en orden logarítmico), cambiando el estado de la reserva.
- **`salidas`** por su parte requiere que dado un aeropuerto mediante su código y un día de la semana, se retorne una lista de vuelos que salen dicho día del mencionado aeropuerto en orden constante en promedio. Resulta entonces la necesidad de un hash de aeropuertos por código, donde cada nodo mantenga un arreglo de listas de salidas, siendo el índice el día de la semana. Por otra parte, dichas listas deben estar ordenadas por hora de salida. Dado que existen un promedio de 700 vuelos, la copia de la lista para ser retornada no influye en el orden promedio de la operación.
- **`llegadas`**, al igual que el procedimiento anterior, implica mantener un arreglo de listas de llegadas ordenadas por hora, siendo el índice el día de la semana. Nuevamente vale la aclaración del promedio de vuelos diarios, a modo de justificar la conservación del orden frente a la copia de la lista para retornar.

La estructura propuesta se presenta en la siguiente figura.



4

El archivo cabecera se detalla a continuación, donde se asume la función weekDay que retorna, dada una fecha a través de mes y día, un entero entre 0 y 6 indicando el día de la semana correspondiente a la misma.

```
#ifndef __AIRPORT_H__
#define __AIRPORT_H__

#define MAX_CODE          4
#define MAX_AIRPORTS     20
#define MAX_PASSENGERS   60
#define MAX_NAME         30

typedef struct _airport_node *Airport;
typedef struct _fly_node *ListaVuelos;

typedef struct {
   int month;
   int day;
} date_t;

struct _fly_node {
   /* Datos del vuelo */
   int fly;
   int hour, min;

   /* Enlace en la lista */
   ListaVuelos next;
};

/*
 * Se asume la función weekDay() que dada una fecha retorna
 * el día de la semana correspondiente (como un entero entre 0 y 6)
 */

int weekDay( date_t date );

Airport crearAeropuerto();

void horarios( int fly_number, Airport c );

bool libre( date_t d, int fly, Airport c );

void pasajeros( date_t d, int fly, Airport c );

void queViajes( char *pass_name, Airport c );

void asignacion( date_t d, int fly_number, char *pass_name, Airport c );

void cancelacion( date_t d, int fly, char *pass_name, Airport c );

ListaVuelos salidas( int day, char *airport, Airport c );

ListaVuelos llegadas( int day, char *airport, Airport c );

void destruirAeropuerto(Airport &a);

#endif
```

La correspondiente implementación se presenta más abajo.

```
#include <stdlib.h>

#include <stdio.h>
#include <string.h>
#include <assert.h>
#include "Airport.h"


/***************************************/
/** Definición de estructuras privadas **/

typedef struct _fly *fly_t;
typedef struct _trip *trip_t;
typedef struct _ticket *ticket_t;
typedef struct _passenger *passenger_t;
typedef struct _airport *airport_t;

typedef struct _fly_point {
   int  hour;
   int  min;
   char airport[MAX_CODE];
} point_t;

struct _fly {
   /* Datos propios del vuelo */
   int number;
   point_t departure[7];
   point_t arrival[7];
   trip_t  trip[12][31];

   /* Enlaces en lista por dia */
   fly_t next_departure;
   fly_t next_arrival;
};

typedef enum { RESERVED, WAITING } ticket_type;

struct _ticket {
   /* Informacion del ticket */
   int fly_number;
   date_t date;
   ticket_type type;
   passenger_t passenger;

   /* Enlace en ABB, espera y reservas */
   ticket_t left, right;
   ticket_t next_wait;
   ticket_t next_trip;
};

struct _trip {
   /* Capacidad y ABB de reservas */
   int capacity, count;
   ticket_t reservations;

   struct {
      /* Primero y ultimo en espera */
      ticket_t first;
```

```
      ticket_t last;
   } wait_list;
};

struct _passenger {
   char name[MAX_NAME];
   ticket_t tickets;
   passenger_t next;
};

struct _airport {
   char code[MAX_CODE];
   fly_t departures[7];
   fly_t arrivals[7];
   airport_t next;
};

struct _airport_node {
   fly_t fly[1000];                          /* Vector de vuelos    */
   passenger_t passenger[MAX_PASSENGERS];    /* Hash de pasajeros   */
   airport_t airport[MAX_AIRPORTS];          /* Hash de aeropuertos */
};


/************************************/
/** Definición de funciones de hash **/

#define airport_hash(s)     hash(s, MAX_AIRPORTS)
#define passenger_hash(s)   hash(s, MAX_PASSENGERS)

static int hash( char *name, int max ){
   int r = 0;

   /* Ejemplo de función de hash */
   while (*name) r += *name++;
   return (r % max);
}


/************************************/
/** Definición de funciones privadas **/

static passenger_t get_passenger( Airport c, char *name ){
   int bucket = passenger_hash(name);
   passenger_t p = c->passenger[bucket];
   bool found = false;

   while (!found && p != NULL) {
      if (!strcmp(p->name, name))
         /* Debe devolverse el p actual */
         found = true;
      else p = p->next;
   }

   return p;
}

static airport_t get_airport( Airport c, char *code ){
   int bucket = airport_hash(code);
   airport_t p = c->airport[bucket];
   bool found = false;
```

```
   while (!found && p != NULL) {
      if (!strcmp(p->code, code))
         /* Debe devolverse el p actual */
         found = true;
      else p = p->next;
   }

   return p;
}

static trip_t get_trip( Airport c, int fly_number, date_t date ){
   fly_t fly;
   trip_t trip;

   fly = c->fly[fly_number];
   assert(fly != (fly_t) NULL);
   trip = fly->trip[date.month - 1][date.day - 1];

   return trip;
}

static void print_tickets( ticket_t t ){
   if (t != NULL) {
      print_tickets(t->left);
      printf("%s\n", t->passenger->name);
      print_tickets(t->right);
   }
}

static bool greater( date_t d1, date_t d2 ){
   return (d1.month > d2.month ||
           (d1.month == d2.month && d1.day > d2.day));
}

static void add_ticket( ticket_t t, passenger_t p ){
   /* Obtiene posicion inicial */
   ticket_t *q = &(p->tickets);
   bool found = false;

   while (!found && *q != NULL) {
      if (greater((*q)->date, t->date))
         /* Debe devolverse el q actual */
         found = true;
      else q = &((*q)->next_trip);
   }

   /* Agrega el ticket */
   t->next_trip = *q;
   *q = t;
}


static void del_ticket( ticket_t t, passenger_t p ){
   ticket_t *q = &(p->tickets);
   bool found = false;

   while (!found && *q != NULL) {
      if (*q == t) {
         *q = t->next_trip;
         found = true;
```

```
      }
      else q = &((*q)->next_trip);
   }
}

static void add_reservation( ticket_t t, ticket_t *root ){
   if (*root == NULL) {
      /* Agrega el ticket como hoja */
      t->left = t->right = NULL;
      *root = t;
   }
   else {
      if (strcmp(t->passenger->name, (*root)->passenger->name) < 0)
         /* Debe agregarse en subárbol izquierdo */
         add_reservation(t, &((*root)->left));
      else
         /* Debe agregarse en subárbol derecho */
         add_reservation(t, &((*root)->right));
   }
}
```

```
static ticket_t del_min( ticket_t *root ){
   ticket_t p;

   if ((*root)->left == NULL) {
      /* La raiz es el minimo! */
      p = *root;
      *root = p->right;
   }
   else p = del_min(&((*root)->left));

   return p;
}

static ticket_t del_reservation( char *passenger, ticket_t *root ){
   int r;
   ticket_t ticket = NULL;
   ticket_t min;

   if (*root != NULL) {
      if ((r = strcmp((*root)->passenger->name, passenger)) < 0) {
         /* Elimina ticket en el subárbol izquierdo */
         ticket = del_reservation(passenger, &((*root)->left));
      }
      else if (r > 0) {
         /* Elimina ticket en el subárbol derecho */
         ticket = del_reservation(passenger, &((*root)->right));
      }
      else if ((*root)->right == NULL) {
         /* Sin derecho! Cambia por izquierdo... */
         ticket = *root;
         *root = ticket->left;
      }
      else {
         /* Enlaza el minimo del derecho */
         ticket = *root;
         min = del_min(&(ticket->right));
         min->left = ticket->left;
         min->right = ticket->right;
         *root = min;
      }
   }

   return ticket;
}
```

```
/************************************/
/** Definición de funciones públicas **/

Airport crearAeropuerto(){
        Airport a = new _airport_node;

        for (int i = 0; i < 1000; i++)
                a->fly[i]=NULL;

        for (int i = 0; i < MAX_PASSENGERS; i++)
                a->passenger[i]=NULL;

        for (int i = 0; i < MAX_AIRPORTS; i++)
                a->airport[i]=NULL;

        return a;
}

void asignacion( date_t d, int fly_number, char *pass_name, Airport c ){
   trip_t trip;
   ticket_t ticket;

   /* Obtiene nodo del pasajero y viaje */
   passenger_t passenger = get_passenger(c, pass_name);
   assert(passenger != (passenger_t) NULL);
   trip = get_trip(c, fly_number, d);
   assert(trip != (trip_t) NULL);

   /* Construye nuevo ticket */
   ticket = (ticket_t) malloc(sizeof(struct _ticket));
   assert(ticket != (ticket_t) NULL);
   ticket->fly_number = fly_number;
   ticket->date = d;
   ticket->passenger = passenger;

   if (trip->count < trip->capacity) {
      /* Agrega ticket al árbol */
      ticket->type = RESERVED;
      add_reservation(ticket, &(trip->reservations));
      trip->count++;
   }
   else {
      /* Agrega ticket a lista de espera */
      ticket->type = WAITING;
      ticket->next_wait = NULL;

      if (trip->wait_list.last == NULL) trip->wait_list.first = ticket;
      else trip->wait_list.last->next_wait = ticket;
      trip->wait_list.last = ticket;
   }

   /* Otorga ticket al pasajero */
   add_ticket(ticket, passenger);
}

void cancelacion( date_t d, int fly, char *pass_name, Airport c ){
   trip_t trip;
   ticket_t ticket;
   passenger_t passenger;

   /* Obtiene nodo del pasajero */
```

```
   passenger = get_passenger(c, pass_name);
   assert(passenger != (passenger_t) NULL);

   /* Obtiene el viaje y borra reserva */
   trip = get_trip(c, fly, d);
   assert(trip != (trip_t) NULL);
   ticket = del_reservation(pass_name, &(trip->reservations));
   assert(ticket != (ticket_t) NULL);

   /* Borra ticket del pasajero */
   del_ticket(ticket, passenger);
   free(ticket);

   if (trip->wait_list.first) {
      /* Agrega primero en espera como reservado */
      add_reservation(trip->wait_list.first, &(trip->reservations));
      trip->wait_list.first->type = RESERVED;

      /* Actualiza lista de espera */
      trip->wait_list.first = trip->wait_list.first->next_wait;
      if (!trip->wait_list.first) trip->wait_list.last = NULL;
   }
   else trip->count--;
}

ListaVuelos salidas( int day, char *airport, Airport c ){
   fly_t fly;
   airport_t ap;
   ListaVuelos list;
   ListaVuelos *p = &list;

   /* Obtiene el aeropuerto */
   ap = get_airport(c, airport);
   assert(ap != (airport_t) NULL);
   fly = ap->departures[--day];

   while (fly != NULL) {
      /* Crea un nuevo nodo para el vuelo */
      *p = (ListaVuelos) malloc(sizeof(_fly_node));
      assert(*p != (ListaVuelos) NULL);
      (*p)->fly = fly->number;
      (*p)->hour = fly->departure[day].hour;
      (*p)->min = fly->departure[day].min;

      /* Avanza al proximo vuelo */
      p = &((*p)->next);
      fly = fly->next_departure;
   }

   /* Finaliza la lista */
   *p = NULL;

   return list;
}

ListaVuelos llegadas( int day, char *airport, Airport c ){
   fly_t fly;
   airport_t ap;
   ListaVuelos list;
   ListaVuelos *p = &list;
```

```
   /* Obtiene el aeropuerto */
   ap = get_airport(c, airport);
   assert(ap != (airport_t) NULL);
   fly = ap->arrivals[--day];

   while (fly != NULL) {
      /* Crea un nuevo nodo para el vuelo */
      *p = (ListaVuelos) malloc(sizeof(_fly_node));
      assert(*p != (ListaVuelos) NULL);
      (*p)->fly = fly->number;
      (*p)->hour = fly->arrival[day].hour;
      (*p)->min = fly->arrival[day].min;

      /* Avanza al proximo vuelo */
      p = &((*p)->next);
      fly = fly->next_arrival;
   }

   /* Finaliza la lista */
   *p = NULL;

   return list;
}

void horarios( int fly_number, Airport c ){
   int i;

   /* Obtiene el vuelo con el numero dado */
   fly_t fly = c->fly[fly_number];
   assert(fly != (fly_t) NULL);

   for (i = 0; i < 7; i++)
      if (*fly->departure[i].airport) {
         printf("%d ==> %2d:%02d - %s\t%2d:%02d - %s\n", i,
                fly->departure[i].hour, fly->departure[i].min,
                fly->departure[i].airport,
                fly->arrival[i].hour, fly->arrival[i].min,
                fly->arrival[i].airport);
      }
}

bool libre( date_t d, int fly, Airport c ){
   /* Obtiene nodo del viaje */
   trip_t t = get_trip(c, fly, d);
   assert(t != (trip_t) NULL);

   return (t->count < t->capacity);
}

void pasajeros( date_t d, int fly, Airport c ){
   /* Obtiene nodo del viaje */
   trip_t t = get_trip(c, fly, d);
   assert(t != (trip_t) NULL);

   printf("Reservas para %3d - %2d/%02d:\n", fly, d.day, d.month);
   printf("--------------------------\n");
   print_tickets(t->reservations);
   printf("--------------------------\n");
}

void queViajes( char *pass_name, Airport c )
```

```
{
   ticket_t t;
   /* Obtiene datos del pasajero */
   passenger_t passenger = get_passenger(c, pass_name);
   if (passenger == NULL) return;
   t = passenger->tickets;

   while (t != NULL) {
      int wDay = weekDay(t->date);
      fly_t fly = c->fly[t->fly_number];
      assert(fly != (fly_t) NULL);

      printf("%3d\t%s\t%2d/%02d - %s\t%2d/%02d - %s\n", t->fly_number,
             (t->type == RESERVED ? "RESERVED" : "WAITING "),
             t->date.day, t->date.month, fly->departure[wDay].airport,
             t->date.day, t->date.month, fly->arrival[wDay].airport);

      /* Avanza a siguiente reserva */
      t = t->next_trip;
   }
}

/*
   Nota: queda como ejercicio para el estudiante la implementación de la
   operación destructora.
*/
```