Ejercicios del Tema 11

1. En el campo del *bigdata* es habitual manejar vectores de dimensiones muy grandes (p.e., vectores de 264 elementos). Afortunadamente, la mayor parte de los elementos de dichos vectores son 0. Implementa el TAD *BigVector*, que permitirá representar vectores cuyos índices pueden ser números naturales arbitrarios (en la práctica, unsigned long), y cuyos valores serán números reales (double). El TAD debe soportar las siguientes operaciones:

- crear: Crea un vector disperso *V* en el que todos los componentes valen 0
- valorDe(V, i): Encuentra el valor del elemento *i* del vector *V*
- ponValor(V, i, val): Fija el valor del elemento *i* del vector *V* a *val*
- productoEscalar(V0, V1): Calcula el producto escalar de *V0* y *V1*
- suma(V0, V1): Calcula la suma de *V0* y *V1*

2. Se desea crear un motor de búsqueda para documentos de texto, de forma que, tras indexar muchos documentos, sea posible recuperar aquellos que contengan una cierta palabra. Implementa las siguientes operaciones en un TAD *Buscador*:

- crear: Crea un buscador vacío, sin documentos.
- indexar(*d*): Añade un documento al buscador, indexando el mismo por su contenido. Asigna, así mismo, un identificador único al documento.
- buscar(*t*): Recupera todos aquellos documentos cuyo contenido contenga el término *t.* El resultado es una lista de identificadores de los documentos encontrados.
- consultar(*id*): Recupera el contenido del documento cuyo identificador es *id*.

3. **(Extraído del examen extraordinario de febrero 2010)** Se desea definir un TAD Agencia que represente a una agencia de viajes y permita realizar gestiones básicas. El TAD debe ofrecer las siguientes operaciones:

- crea: crea una agencia vacía.

- aloja(c, h): modifica el estado de la agencia alojando a un cliente *c* en un hotel *h*. Si c ya tenía antes otro alojamiento, éste queda cancelado. Si *h* no estaba dado de alta en el sistema, se le dará de alta.
- desaloja(c): modifica el estado de una agencia desalojando a un cliente c del hotel que éste ocupase. Si *c* no tenía alojamiento, el estado de la agencia no se altera.
- alojamiento(c): permite consultar el hotel donde se aloja un cliente *c*, siempre que éste tuviera alojamiento. En caso de no tener alojamiento produce un error.
- listado_hoteles(): obtiene una lista ordenada de todos los hoteles que están dados de alta en la agencia.
- huespedes(h): permite obtener la lista ordenada de clientes que se alojan en el hotel *h*. Dicha lista será vacía si no hay clientes en el hotel.

Se pide:

a) Obtener una representación eficiente del tipo utilizando los TADs vistos en la asignatura.

b) Implementar todas las operaciones indicando el coste de cada una de ellas.

4. (Adaptación del examen de junio 2010) Una confederación hidrográfica gestiona el agua de ríos y pantanos de una cuenca hidrográfica. Para ello debe conocer los ríos y pantanos de su cuenca, el agua embalsada en la cuenca y en cada pantano. También se permite trasvasar agua entre pantanos del mismo río o de distintos ríos dentro de su cuenca.

Se pide diseñar e implementar un TAD que permita a la confederación hidrográfica gestionar la cuenca. En particular, se piden las siguientes operaciones con los siguientes comportamientos:

- crear: Creación de una cuenca hidrográfica vacía.
- insertar_rio(r): Añade un río. La cuenca no puede tener dos ríos que se llamen igual. Operación no definida si ya existe el río *r* en la cuenca.
- insertar_pantano(r, p, n1, n2): Añade un pantano al río *r*, con nombre *p* (un río no puede tener dos pantanos que se llamen igual). La capacidad máxima del pantano es de *n1* Hm^3. Además, lo llena con *n2* Hm^3 (si *n2* > *n1* entonces lo llena completamente). Operación no definida si ya existe algún pantano de nombre *p* en el río *r* o no existe el río *r*.
- embalsar(r, p, n): Carga *n* Hm^3 en el pantano *p* del río *r*. Si se excede la capacidad del pantano, éste se llena completamente. Operación no definida si el río o el pantano no existen.

- embalsado_pantano(r, p): Devuelve la cantidad de agua embalsada en el pantano *p* del río *r*. Operación no definida si el río o el pantano no existen.

- embalsado_cuenca(r): Devuelve la cantidad de agua embalsada en el río *r*. Operación no definida si el río no existe.

- transvasar(r1, p1, r2, p2, n): Transvasa *n* Hm^3 del pantano *p1* en *r1* al pantano *p2* en *r2*. Si *n* colma a *p2*, se ajusta a lo que queda para llenar *p2*. Si este valor excede el volumen de *p1*, se trasvasa únicamente la totalidad de dicho volumen. Si el volumen de agua a transvasar n es negativo, se cambia el sentido del transvase (es decir, se pasa el agua del pantano destino al pantano origen). Operación no definida si alguno de los ríos o pantanos no existen.

5. (Extraído del examen parcial de septiembre 2016) Un comerciante ha decidido instalar en su almacén un sistema para recibir pedidos por internet y gestionarlos. El sistema cuenta con una tabla en la que se almacenan todos los objetos disponibles en el almacén junto con información sobre el precio, peso y número de unidades disponibles de cada uno de ellos. Ya se tienen implementadas operaciones para dar de alta nuevos objetos, y para modificar las características de un objeto existente.

```
typedef string Objeto;
typedef struct {
    float precio;
    float peso;
    int numUnidades;
} InfoObjeto;
class Pedidos {
    public:   //operaciones para dar de alta objetos y modificarlos ...
    private:  DiccionarioHash<Objeto,InfoObjeto> objetos;
}
```

El sistema debe almacenar información sobre el pedido (lista de objetos) que debe entregarse cada día y la dirección de envío. Sólo da tiempo a entregar un pedido al día. No se pueden enviar objetos de los que no haya unidades.

La clase *Almacen* debe contar con las siguientes operaciones:

- nuevoPedido: Recibe una lista con los objetos que un cliente solicita, la fecha en la que desea recibir la mercancía y la dirección de envío. La operación debe almacenar el pedido asociándole la fecha en la que será entregado. En caso de que la fecha deseada ya esté asignada, se asignará la fecha más próxima posterior a la solicitada que no tenga todavía asignado ningún pedido.

- preparaPedidos: Dadas dos fechas, devuelve una lista, ordenada por fecha, con información sobre los envíos que se realizarán entre las dos fechas dadas, ambas incluidas. Cada elemento de la lista contendrá la fecha de entrega, la lista de objetos a enviar y la dirección de envío.

Se puede hacer uso de un tipo *Fecha* que dispone de una operación para obtener la siguiente fecha a una dada.

Se pide:

a) Diseñar una representación que permita implementar las operaciones pedidas de forma que el acceso a la información almacenada sea eficiente. Indicar qué operaciones adicionales ha de tener disponibles el tipo *Fecha* para poder usarlo en la representación propuesta.

b) Implementar la función nuevoPedido.

c) Implementar la función preparaPedidos.

6. (Extraído del examen final de junio 2016) Los ingenieros de la empresa Apel están actualmente diseñando el software para su nuevo reproductor de música *iPud*, el cual debe permitir añadir y eliminar canciones, añadir una canción existente a la lista de reproducción, avanzar a la siguiente canción, obtener el tiempo total de la lista de reproducción, y obtener una lista con las canciones escuchadas recientemente.

En particular, el TAD *IPud* debe contar con las siguientes operaciones:

- create: Crea un iPud vacío.
- addSong(S, A, D): Añade la canción S (string) del artista A (string) con duración D (int) al iPud. Si ya existe una canción con el mismo nombre la operación dará error.
- addToPlaylist(S): Añade la canción S al final de la lista de reproducción. Si la canción ya se encontraba en la lista entonces no se añade (es decir, la lista no tiene canciones repetidas). Si la canción no está en el iPud se devuelve error.
- deleteSong(S): Elimina todo rastro de la canción S. Si la canción no existe la operación no tiene efecto.
- play: La primera canción de la lista de reproducción abandona la lista de reproducción y se registra como reproducida. Si la lista es vacía la acción no tiene efecto.
- current: Devuelve la primera canción de la lista de reproducción. Si es vacía se devuelve error.
- totalTime: Obtiene la suma de las duraciones de las canciones que integran la lista de reproducción actual. Si es vacía se devuelve 0.

- recent(N): Obtiene la lista con las N últimas canciones que se han reproducido (mediante la operación play), de la más reciente a la más antigua. Si el número de canciones reproducidas es menor que N se devolverán todas. La lista no tiene repeticiones, de manera que si una canción se ha reproducido más de una vez sólo figurará la reproducción más reciente.

Se puede asumir que las canciones quedan unívocamente determinadas por su nombre (string).

Se ha elegido la siguiente representación para el TAD *IPud* usando los TADs vistos en clase:

```
class SongInfo {
  string artist;
  int duration;
  bool inPlaylist; //indica si esta en la lista de reproduccion
  bool played; //indica si esta en la lista de reproducidas
  SongInfo(): inPlaylist(false), played (false) {}
} SongInfo;

class IPud{
  public:
    //... las operaciones pedidas...
  private:
    DiccionarioHash<string, SongInfo> songs;
    Lista<string> playlist; //la lista de reproduccion, sin repeticiones
    Lista<string> played; //las ya reproducidas, sin repeticiones
    int duration; // duración de la lista de reproducción
}
```

Se pide:

a) Implementar cada una de las operaciones indicadas y estimar su coste con la representación anterior.

b) Modificar la representación anterior para que todas las operaciones tengan coste constante, excepto *recent* que no debe superar el coste lineal en N.

7. Se desea implementar el planificador de procesos del sistema operativo `Guindows`. Cada proceso está unívocamente identificado por un identificador de proceso (un entero sin signo), el nombre de usuario que lo ha lanzado (string), una descripción (string), y una ruta al ejecutable (string). La política de planificación será FIFO (es decir, los procesos se ordenan para su ejecución por orden de llegada).

EL desarrollo de este planificador se plantea como un TAD con las siguientes operaciones:

- crea: Crea un planificador vacío.

- an_proceso(*U,D,R*): pid. Añade un proceso lanzado por el usuario *U*, con descripción *D* y con ruta *R*. Devuelve su identificador único.
- pid siguiente(): Devuelve el identificador del primer proceso en espera de ejecución (error en caso de que no haya procesos en espera)
- ejecuta(): Ejecuta el siguiente proceso. Como resultado, el proceso se elimina de la lista y del sistema de planificación. Si no hay procesos, la operación no tiene efecto.
- detalles(pid):proc. Devuelve la información asociada a un proceso, dado su identificador (en caso de que el proceso no exista, señala un error)
- termina(pid). Termina un proceso, eliminándolo del sistema y de la lista de planificación (en caso de que el proceso no exista, no tiene efecto).
- num_procesos. Devuelve el número de procesos pendientes de ejecución.

Desarrolla este TAD, asegurando la máxima eficiencia en la implementación de las operaciones. Estima el coste de las mismas.

8. (Extraído del examen final de junio 2017) Nos han encargado implementar un sistema para la gestión de la admisión en el Servicio de Urgencias de un Hospital. Cuando un paciente llega al servicio, se le toman los datos y se le asigna un código de identificación único (un número entero no negativo). A partir de ahí, el paciente espera a ser atendido. El orden de atención da prioridad a los pacientes por orden de llegada. Una vez atendido un paciente, sus datos se eliminan del sistema. Así mismo, en cualquier momento un paciente puede desistir de ser atendido. En este caso, en el control de salida se le solicita su número de identificación, y se elimina todo rastro de él del sistema.

La implementación del sistema se deberá realizar como un TAD GestionAdmisiones con las siguientes operaciones:

- crea(): Operación constructora que crea un sistema de gestión de admisiones vacío.
- an_paciente(codigo, nombre, edad, sintomas): añade al sistema un nuevo paciente con código de identificación *codigo*, con nombre *nombre*, con edad *edad* y con una descripción de síntomas *sintomas*. En caso de que el código esté duplicado, la operación señala un error "Paciente duplicado".
- info_paciente(codigo, nombre, edad, sintomas): *nombre*, *edad* y *sintomas* devuelven la información correspondiente al paciente con código *codigo*. En caso de que el código no exista, levanta un error "Paciente inexistente".
- siguiente(codigo): almacena en *codigo* el código del siguiente paciente a ser atendido. En caso de que no haya más pacientes, esta operación levanta un error "No hay pacientes".
- hay_pacientes(): devuelve true si hay más pacientes en espera, y false en otro caso.

- elimina(codigo): elimina del sistema todo el rastro del paciente con código *codigo*. Si no existe tal paciente, la operación no tiene efecto.

Dado que este es un sistema crítico, la implementación de las operaciones debe ser lo más eficiente posible. Por tanto, debes elegir una representación adecuada para el TAD, implementar las operaciones y justificar la complejidad resultante.

9. (Extraído del examen parcial de junio 2017) Nos han encargado implementar un sistema para la gestión de la admisión en el Servicio de Urgencias de un Hospital. Cuando un paciente llega al servicio, se le toman los datos, se le asigna un código de identificación único (un número entero no negativo), y también se le asigna un nivel de urgencia, dependiendo de su estado (leve, normal, o grave). A partir de ahí, el paciente espera a ser atendido. El orden de atención da prioridad a los pacientes graves sobre los normales, y a los normales sobre los leves. Una vez atendido un paciente, sus datos se eliminan del sistema. Así mismo, en cualquier momento un paciente puede desistir de ser atendido. En este caso, en el control de salida se le solicita su número de identificación, y se elimina todo rastro de él del sistema.

La implementación del sistema se deberá realizar como un TAD GestionAdmisiones con las siguientes operaciones:

- crea(): Operación constructora que crea un sistema de gestión de admisiones vacío.
- an_paciente(codigo, nombre, edad, sintomas, gravedad): Añade al sistema un nuevo paciente con código de identificación *codigo*, con nombre *nombre*, con edad *edad*, con una descripción de síntomas *sintomas*, y con código de gravedad *gravedad*. En caso de que el código esté duplicado, la operación señala un error "Paciente duplicado".
- info_paciente(codigo, nombre, edad, sintomas): Almacena en *nombre*, *edad* y *síntomas* la información correspondiente del paciente con código *codigo*. En caso de que el código no exista, levanta un error "Paciente inexistente".
- siguiente(codigo, gravedad): Almacena en *codigo* y *gravedad*, respectivamente, el código y la gravedad del siguiente paciente a ser atendido. Como se ha indicado antes, se atiende primero a los pacientes graves, después a los de nivel de gravedad normal, y por último a los leves. Dentro de cada nivel, los pacientes se atienden por orden de llegada. En caso de que no haya más pacientes, esta operación levanta un error "No hay pacientes".
- hay_pacientes(). Devuelve true si hay más pacientes en espera, y false en otro caso.
- elimina(codigo): Elimina del sistema todo el rastro del paciente con código *codigo*. Si no existe tal paciente, la operación no tiene efecto.

Dado que este es un sistema crítico, la implementación de las operaciones debe ser lo más eficiente posible. Por tanto, debes elegir una representación adecuada para el TAD, implementar las operaciones y justificar la complejidad resultante.