Todo sobre TADs

Teoría sobre los TADs

Ejercicios sobre teoría de TADs

Ejercicios prácticos sobre TADs

TAD Hipermercado

Un hipermercado dispone para el pago de las compras de 50 cajas numeradas del 1 al 50. El número de cajas que permanecen abiertas a lo largo del dia es variable, de tal forma que en cada momento hay entre 10 y 50 cajas en funcionamiento, dependiendo de la afliuencia de clientes (en las horas y fechas puntas hay más cajas abiertas que en momentos de menor clientela), aunque la media de cajas abiertas es de 48. El horario de venta se cubre con dos turnos de trabajo al día y en cada turno se supone que hay suficientes cajeros para atender la demanda máxima que se pueda producir y además algunos suplentes para sustituir a los que necesiten descansar. Para cada caja hay que almacenar el número de caja, el identificador del cajero (3 digitos) y la facturacion realizada desde el comiezo del turno de trabajo.

A) Escribe la especificación del TAD LíneaCajas con las siguientes operaciones:

- Crear una linea de cajas cerradas.
- Abrir una caja determianda.

- Cerrar una caja determianda y devolver la facturación realizada en ella desde la última vez que se abrió.
- Cobrar a un cliente el importe de su compra en una caja.
- Sustituir al cajero de una caja determianda. La caja continua funcionando.
- Cambiar el turno de trabajo y devolver la recaudación total que hay en las cajas en el momento del cambio. En esta operación puede que el total de cajas, así como las cajas concretas de uno y otro turno no coincidan.
- Cerrar todas las cajas y devolver la recaudacion total que hay en ellas en el momento del cierre.

B) Realiza el TAD teniendo en cuenta lo siguiente:

- El tiempo de ejecución de cerrar una caja, cobrar una compra y susituir a un cajero debe de ser constante.
- Justifica razonadamente la estructura de datos elegida en términos de eficiencia en tiempo y espacio.

Sabemos que un hipermercado contiene cajas, estas cajas estan enumeradas del 1 al 50, esto nos indica que podemos hacer uso de un vector de cajas donde cada posición del vector será una caja completamente diferente.
Esto hará que las operaciones sean de O(1) poder acceder a una caja para poder introducirle los datos si esta está abierta o no.

La caja se identificará del resto por su número de caja (posición en dicho vector), identificador del cajero (char) y la facturación (double).

Especificación de las operaciones del TAD LineaCajas

```
//Postcondicion: Crea una linea de cajas cerradas
  LineaCajas();
//Postcondicion: Abrir una caja determinada
```

```
  void AbrirCaja(int , const char&);
//Postcondicion: Cierra una caja determinada y devuelve la facturacion
  double CerrarCaja(int );
//Postcondicion: Cobra a un cliente un importe
  void Cobrar(int , double);
//Postcondicon: Cambia al cajero de una caja determinada
  void CambiarCajero(int, const char&);
//Postcondicion: Cambia el turno de una caja y devuelve recaudacion de
las cajas que cambian de turno
  double CambioTurno();
//Postcondicion: Cierra todas las cajas y devuelve la recaudación total
  double CerrarCajas();
```

Implementación del TAD LineaCajas

```
#include <iostream>

class LineaCajas{
  public:
    typedef struct Caja{
      int numerocaja_;
      char[3] idcajero_;
      bool abierta_;
      double recaudacion_;
    };
    //metodos de la clase LineaCajas
    LineaCajas();
    void AbrirCaja(int, char&);
    double CerrarCaja(int);
    void Cobrar(int, double);
    void CambiarCajero(int, const char&);
    double CambioTurno();
    double CerraraCajas();
    //no hace falta destructor ya que vector tiene el suyo propio
  private:
    //vector de 50 cajas
```

```
    Caja cajas[50];
    double facturaciontotal_;
};
//constructor
LineaCajas::LineaCajas():reacudaciontotal_(0.0){
  //creamos el vector de 50 cajas cerradas y sin recaudacion
  for (unsigned i =0 ; i< 50; i++){
    cajas[i].numerocaja_ = i+1;
    cajas[i].abierta_ = false;
    cajas[i].recaudacion_ = 0.0;
  }
}

void LineaCajas::AbrirCaja(int numCaja, char& idCajero){
  //buscamos la caja introducida y comprobar que no está abierta para
asignarle un cajero
    if(numCaja >= 1 && numCaja <= 50 && !caja[numCaja-1].abierta){
//hemos encontrado la caja y está cerrada
      cajas[numCaja-1].idcajero_ = idCajero;
      cajas[numCaja-1].abierta = true;
    }
  }


double LineaCajas::CerrarCaja(int numCaja){
  //Buscamos la caja y comprobamos si está abierta para cerrarla para
cerrarla
    if(numCaja >= 1 && numCaja <= 50 && caja[numCaja-1].abierta){
      //creamos una variable double auxiliar, para guardar el dinero
recaudado.
      double fact = 0.0;
      fact += cajas[numCaja-1].recaudacion_;
      cajas[numCaja-1].recaudacion_ = 0.0;
      cajas[numCaja-1].abierta_ = false;
      return fact;
    }else{return 0.0;} // si la caja no se puede cerrar
```

```
}

void LineaCajas::Cobrar(int numCaja, double importe){
  //buscamos la caja y si está abierta para poder cobrar al cliente
  if(numCaja >= 1 && numCaja<=50 && cajas[numCaja-1].abierta){
    cajas[numCaja-1].recaudacion += importe;
    facturaciontotal_ += importe;
  }
}

void LineaCajas::CambiarCajero(int numCaja, const char& idCajero){
  //buscamos la caja abierta para cambiar el cajero
  if(numCaja >= 1 && numCaja <= 50 && cajas[numCaja-1].abierta){
    cajas[numCaja-1].idcajero_ = idCajero;
  }
}

double LineaCajas::CambioTurno(){
  //recorremos el vector de cajas y las que estan abiertas pasan a
cerrada y devuelve la recaudacion
  //creamos una variable que almacene la recaudacion total
  double factotal = facturaciontotal_;;
  for(unsigned i =0; i<50; i++){
    cajas[i].abierta_ = false;
    cajas[i].recaudacion_ = 0.0;
  }
  facturaciontotal_ = 0.0;  //reiniciamos el total facturado
  return factotal;
}

double LineaCajas::CerrarCajas(){
   double factotal = facturaciontotal_;;
  //vamos recorriendo el vector cerrando todas las cajas y obteniendo la
facturacion total
  for(unsigned i =0 ; i<50; i++){
    cajas[i].abierta_ = false;
```

```
      cajas[i].recaudacion_ 0.0;
    }
    facturaciontotal_ =0.0;
    return factotal;
  }
```

TAD Consultorio

La dirección de un hospital quiere informatizar su consultorio médico con un programa que realice las siguientes operaciones:

- Generar un consultorio vacío.
- Dar de alta a un nuevo médico.
- Dar de baja a un médico.
- Poner a un paciente en la lista de espera de un médico.
- Consultar al paciente a quien le toca el turno para ser atendido.
- Atender al paciente que le toque por parte de un médico.
- Comprobar si un médico determinado tiene o no pacientes en espera.

Diseña la ED para representar el TAD e implementar las operaciones anteriores

Un consultorio está formado por médicos que atienden a unos pacientes por orden de llegada, por tanto vamos a hacer uso de los TAD Lista y Cola ambos dinámica debido a que no sabemos el tamaño total del consultorio. Hacemos uso del TAD Lista debido que vamos a tener una secuencia de médicos identificados con id, por tanto, vamos a poder acceder a cualquier médico a través de ese id. Por otro lado, los pacientes estarán contenidos en una cola definidos por un turno que es la posicion en la cola, contra antes pidas cita en la consulta, antes será atendido.
Entendemos atender al paciente como eliminarlo de la cola y poner a un paciente en la lista

de espera como insertar el paciente en dicha cola.

Sea la especificacion del TAD Cola:

La cola es una secuencia de elementos en donde las operaciones de inserción y eliminación de elemantos se realizan en los extremos, se insertan elementos en el fin de la cola y se eliminan en el frente.

```
typedef <typename T> class Cola{
  public:
    Cola():inicio(nullptr),fin(nullptr){}; //ctro.
    Cola(const Cola<T>& c); //ctor copia.
    Cola<T>& operator =(const Cola<T>& c); //operador de asignacion.
    void push(const T& elemento);
    void pop();
    const T&elemento frente() const;
    bool vacia() const;
    ~Cola();
  private:
  typedef struct nodo{
    nodo* sig;
    T elto;
    nodo(const T& e, nodo* p= nullptr):elto(e),sig(p){}
  };
  nodo* inicio, fin;
  void copiar(const Cola<T>& c);
};
```

Sea la especificación del TAD Lista:

La lista es una secuencia de elementos en donde podemos realizar las operaciones de inserta, eliminar y acceder a cualquier elemento de la misma indicando una posicion concreta de la misma.

```
typedef <typename T> class Lista{
  public:
```

```
    Lista():L(nullptr){};
    Lista(const Lista<T>& l); //ctor copia.
    Lista<T>& operator = (const Lista<T>& l);
    void insertar(const T& x, posicion p);
    void eliminar (posicion p);
    const T& elemento(posicion p) const;
    bool vacia() const;
    posicion primera() const;
    posicion fin() const;
    posicion siguiente(posicion p);
    posicion anterior(poscion p);
    ~Lista();
  private:
    typedef struct nodo{
      nodo* sig;
      T elto;
      nodo(const T& e;nodo* p=nullptr):elto(e),sig(p){}
    };
  nodo* L; //nodo cabecera;
  void copiar(const Lista<T>& l);
};
```

Especificación de las operaciones del TAD Consultorio

```
//Postcondicion: Crea un consultorio vacio
  Consultorio();
//Precondicion: Medico no está en el consultorio
//Postcondicion: Introduce un medico dando un id
  void AñadirMedico(unsigned );
//Precondicion: Medico está en el consultorio
//Postcondicion: Elimina el medico y su cola del consultorio dado un id
  void EliminarMedico(unsigned);
//Precondicion: Existe el medico y el paciente no está en la lista de
espera
//Postcondicon: Introduce el paciente en el consultorio de un medico
determinado
```

```
  void IntroducePaciente(unsigned , const Paciente& );
//Precondicion: Medico existe en el consultorio
//Postcondicion: Devuelve el paciente que va a ser atendido
  Paciente ConsultarPaciente(unsigned ) const;
//Precondicion: Medico existe en el consultorio
//Postcondicion: Atiende al paciente (frente) --> lo elimina
  void AtenderPaciente(unsigned );
//Precondicion: Medico existe en el consultorio
//Postcondicion: devuelve true si tiene pacientes(cola no vacia).
  bool TienePacientes(unsigned) const;
```

Implementación del TAD Consultorio

```
#include <iostream>
#include "Lista.hpp"
#include "Cola.hpp"
using namespace std;

//creamos la estructura paciente
typedef struct Paciente{
  unsigned numero_;
};

class Consultorio{
  public:
    //estructura de medico
    typedef struct Medico{
      unsigned id_;
      //cola de pacientes de un medico
      Cola<Paciente>lista_espera;
    };
    //metodos de la clase
    Consultorio(){};
    void AñadirMedico(unsigned );
    void EliminarMedico(unsigned );
    void IntroducePaciente(unsigned , const Paciente& );
```

```
    Paciente ConsultarPaciente(unsigned ) const;
    void AtenderPaciente(unsigned );
    bool TienePacientes(unsigned) const;
    //creamos un método específico para comprobar si existe o no el
médico.
    bool ExisteMedico(unsigned) const;
    ~Consultorio();
  private:
  //Lista de los medicos del consultorio
  Lista<Medico>consultorio;
};
typedef typename Lista<Medico>::posicion posicion;

//Implementacion del método específico del TAD
bool Consultorio::ExisteMedico(unsigned idmed)const{
  //definimos p como la primera posicion de la lista de medicos
  posicon p = consultorio.primera();
  for(p;p!=consultorio.fin(); p = consultorio.siguiente(p)){
//recorremos la lista
    //comprobamos si coinciden los ids
    if(consultorio.elemento(p).id_ == idmed)return true;
  }
  return false; //Si no lo encuentra
}

void Consultorio::AñadirMedico(unsigned idmed){
  //comprobamos si no existe el médico cuyo id vamos a introducir
  if(!ExisteMEdico(idmed)){//si no existe lo insertamos en la lista en
la ultima pos
    consultorio.insertar(med,consultorio.fin());
  }
}

void Consultorio::EliminarMedico(unsigned idmed){
  //comprobamos si existe el medico para eliminarlo
  if(ExisteMedico(idmed)){
```

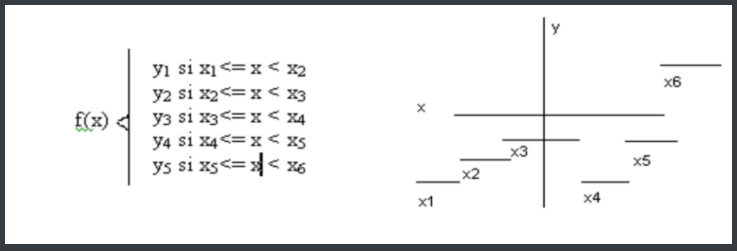
```
    consultorio.eliminar(idmed);
  }
}
void Consultorio::IntroducePaciente(unsigned idmed ,const Paciente& pac)
{
  //comprobamos que existe el medico para poder ingresar el paciente
  if(ExisteMedico(idmed)){
    //Existe, lo buscamos e introducimos
    posicion p = consultorio.primera();
    for(p;p!=consultorio.fin(), p = consultorio.siguiente(p)){
      if(consultorio.elemento(p).id_ == idmed){
        consultorio.elemento(p).lista_espera.push(pac);
      }
    }
  }
}
Paciente Consultorio::ConsultarPaciente(unsigned idmed ) const{
  //verificamos si existe medico y devolvemos el que va a ser atendido
(frente).
  if(ExisteMedico(idmed)){
    //Existe, lo buscamos y devolvemos
    posicion p = consultorio.primera();
    for(p; p!=consultorio.fin(); p = consultorio.siguiente(p)){
      if(consultorio.elemento(p).id_ == idmed){
        return consultorio.elemento(p).lista_espera.frente();
      }
    }
  }
}
void Consultorio::AtenderPaciente(unsigned idmed){
  if(ExisteMedico(idmed)){
    //existe, lo buscamos y eliminamos el paciente
    posicion p = consultorio.primera();
    for(p;p!=consultorio.fin();p= consultorio.siguiente(p)){
      if(consultorio.elemento(p).id_ == idmed){
        consultorio.elemento(p).lista_espera.pop();
```

```
      }
    }
  }
}

bool Consultorio::TienePacientes(unsigned idmed) const{
  if(ExisteMedico(idmed)){
    //existe, lo buscamos y eliminamos el paciente
    posicion p = consultorio.primera();
    for(p;p!=consultorio.fin();p= consultorio.siguiente(p)){
      if(consultorio.elemento(p).id_ == idmed){
        return consultorio.elemento(p).lista_espera.vacia();
      }
    }
}
Consultorio::~Consultorio(){
  //por cada medico eliminamos su cola de pacientes
  posicion p = consultorio.primera();
  for(p; p!= consultorio.fin(); p = consultorio.siguiente(p)){
    consultorio.elemento(p).lista_espera.~Cola(); //destructor de cada
cola
  }
  consultorio.~Lista(); //destructor del consultorio
}
```

TAD Escalonada

Una función f:R ->R es escalonada cuando consiste en una sucesión de funciones constantes definidas en subintervalos disjuntos contiguos, es decir, f puede definirse mediante condiciones de la forma f(x) = y_i si x_i<= z <= x_{i+1}, doonde los valores y_i son distintos para subintervalos adyacentes. Cada uno de los puntos (x_i, y_i) en los que la función cambia de valor se llama salto o escalón.



Especificar un TAD para las funciones escalonadas con un numero finito de saltos, que incluta las siguientes operaciones:

- Crear una función constante $f(x) = y_i$ definida a partir de x_i
- Añadir un nuevo salto en el punto(x,y) a una función. Si ya existe un salto en la coordenada x, se sustituye por el nuevo.
- Eliminar el escalón definido en el subintervalo al que pertenece la coordenada x. El escalón anterior al eliminado se prolonga hasta el siguiente.
- Calcular el valor de una función en un punto dado.
- Calcular el valor minimo de una función escalonada.
- Calcular el valor maximo de una función escalonada.
- Hacer una translación de una funcion w unidades horizontales y z unidades verticales, siendo w y z números reales.
- Destruir una función.

Sabemos por el enunciado que una funcion escalonada es la sucesión de escalones en un subintervalo definido. Por tanto, haremos uso del TAD Lista ya que dando la posicion del eje x podremos acceder a cualquier trazo pudiendo así realizar las operaciones que nos indica el enunciado.

Sea la especificación del TAD Lista:

Una Lista es una secuencia de elementos en donde podemos realizar operaciones de inserción, eliminacion y acceso a cualquier elemento indicando una posicion de la lista.

```
typedef <typename T> class Lista{
  public:
    Lista():L(nullptr){};//ctor
    Lista(const Lista<T>& l);//ctor copia
    Lista<T>& operator =(const Lista<T>& l); //operador de asignacion
    void insertar(const T& x, posicion p);
    void eliminar (posicion p);
    const T& elemento(poscion p);
    bool vacia() const;
    posicion primera() const;
    posicion fin()const;
    posicion siguiente(posicion p);
    posicion anterior(posicion p);
    ~Lista();
  private:
  typedef struct nodo{
    nodo* sig;
    T& elto;
    nodo(const T& e, nodo* p=nullptr):elto(e),sig(p){};
  }
  nodo* L; //nodo cabecera
  void copiar(const Lista<T>& l);
};
```

Especificación de los métodos de TAD Escalonada

```
//Postcondicion: Crea una funcion escalonada
  Escalonada();
//Postcondicon: Añade un salto/escalon en el punto(x,y), si hay uno lo
sobreescribe
  void AñadirSalto(const Salto&);
//Precondicion: Escalonada no vacia
//Postocidicion: Elimina el escalon en el subintervalo definido
  void EliminaSalto(const Salto& );
//Precondicion: Escalonada no vacia
```

```
  //Postcondicion: Devuelve el valor de una funcion dado el punto
    double ValorFuncion(double)const;
  //Precondicion: Escalonada no vacia
  //Postcondicion: Devuelve el valor minimo de la funcion
    double MinimoValor()const;
  //Precondicion: Escalonada no vacia
  //Postcondicion: Devuelve el valor maximo de la funcion
    double MaximoValor()const;
  //Precondicion: Escalonada no vacia
  //Postcondicion: realiza la translacion de una funcion w y z unidades.
    void Translacion(double, double);
  //Postcondicion: Elimina la funcion escalonada
    ~Escalonada();
```

Implementacion de los métodos del TAD Escalonada

```
#include <iostream>
#include "Lista.hpp"
using namespace std;

class Escalonada{
  public:
  typedef struct Salto{
      double inicio_, fin_, altura;
      Salto(double i=0.0, double f=0.0, double
a=0.0):incio_(i),fin_(f),altura_(a){}
    };
  //metodos de la clase
    Escalonada(const Salto& s);
    void AñadirSalto(const Salto& s);
    void EliminaSalto(const Salto& s);
    double ValorFuncion(double)const;
    double MinimoValor()const;
    double MaximoValor()const;
    void Translacion(double, double);
    ~Escalonada();
```

```
  private:
    double x_,y_,z_,w_, valor_;
    Lista<Salto>funcion_escalonada; //lista de los saltos de la funcion
};
typedef typename Lista<Salto>::posicion posicion;

Escalonada::Escalonada(const Salto& s){
  //Creamos la funcion escalonda con el primer salto
  funcion_escalonada.insertar(s, funcion_escalonada.primera());
}

void Escalonada::AñadirSalto(const Salto& s){
  //definimos p como la primera posicion de nuestra lista
  posicion p = funcion_escalonada.primera();
  //buscamos en todos los saltos si el inicio de los saltos de la lista
  //concuerda con el inicio del salto introducido
  for(p;p!=funcion_escalonada.fin();p=funcion_escalonada.siguiente(p)){
    if(funcion_escalonada.elemento(p).inicio_ != s.inicio_){
      //como los inicios son distintos, insertamos el trazo sin
problemas
        funcion_escalonada.insertar(s,p);
    }else{ //los inicios son iguales, debemos de sustituir la funcion,
pero vamos a comprobar

funcion_escalonada.insertar(salto(funcion_escalonada.elemento(funcion_es
calonada.fin()).fin_,x_,y_));
    }
  }
}

void Escalonada::EliminarSalto(const Salto& s){
  //definimos p como la primera posicion de nuestra lista
  posicion p = funcion_escalonada.primera();
  for(p;p!=funcion_escalonada.fin();p=funcion_escalonada.siguiente(p)){
    if(funcion_escalonada.elemento(p).inicio_ == s.inicio_){
        funcion_escalonada.eliminar(p);
```

```
    }
  }
}

double Escalonada::ValorFuncion(double x)const{
  //buscamos el salto que concuerde con dicha posicion
  posicion p = funcion_escalonada.primera();
  for(p;p!=funcion_escalonada.fin();p=funcion_escalonada.siguiente(p)){
    if(funcion_escalonada.elemento(p).inicio_ >= x &&
funcion_escalonada.elemento(p).fin_ <= x){
      //estamos en el intervalo, devolvemos la altura
        return funcion_escalonada.elemento(p).altura_;
    }
  }
  return 0; //si no encuentra nada.
}

double Escalonada::MinimoValor() const{
  //recorremos la funcion y vamos comprarndo las alturas
  posicion p = funcion_escalonada.primera();
  double minimo = funcion_escalonada.elemento(p).altura_;
  for(p;p!=funcion_escalonada.fin();p=funcion_escalonada.siguiente(p)){
    if(funcion_escalonada.elemento(p).altura_ < minimo){
      minimo = funcion_escalonada.elemento(p).altura_;
    }
  }
  return minimo;
}

void Escalonada::Translacion(double w, double z){
  //vamos a desplazar la funcion w unidades horizontales, z verticales.
  //es decir sumamos w a la variable inicio_ y sumamos z a la variable
fin_
  posicion p = funcion_escalonada.primera();
  for(p;p!=funcion_escalonada.fin();p = funcion_escalonada.siguiente(p))
{
```

```
    //a cada salto vamos sumando los valores
    funcion_escalonada.elemento(p).inicio_ += w;
    funcion_escalonada.elemento(p).fin_ += z;
  }
}

double Escalonada::MaximoValor() const{
  //igual que MinimoValor
  posicion p = funcion_escalonada.primera();
  double maximo = funcion_escalonada.elemento(p).altura_;
  for(p; p!= funcion_escalonada.fin(); p=
funcion_escalonada.siguiente(p)){
    if(funcion_escalonada.elemento(p).altura_ > maximo){
      maximo = funcion_escalonada.elemento(p).altura_;
    }
  }
  return maximo;
}

Escalonada::~Escalonada(){
  funcion_escalonada.~Lista();
}
```