EDA, 2014/15

8 de mayo de 2015

Examen final (Febrero 2010)

Se desea definir un tipo abstracto de datos *Agencia* para representar una agencia hotelera. El TAD estará parametrizado respecto a la información de los clientes y a la información de los hoteles. Se deben 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 el cliente c, siempre que éste tuviera alojamiento. En caso de no tener alojamiento, produce un error.
- *listado_hoteles():* Obtiene un lista de todos los hoteles que están dados de alta en la agencia, ordenados según el orden definido en el tipo de parámetro.
- *huespedes(h):* Permite obtener el conjunto de clientes que se alojan en un hotel dado. Dicho conjunto sera vacío si no hay clientes en el hotel.

Se pide:

(i) Obtener una representación eficiente del tipo utilizando estructuras de datos conocidas.

(ii) Implementar todas las operaciones, indicando el coste de cada una de ellas. La operación *huespedes* debe producir un lista de clientes en lugar de un conjunto.

EX. FINAL - FEBRERO 2010

- Agencia

TreeMap <Hotel, clientes>
HashMap <Cliente, hotel>

```
template <class H, class C>
class Agencia {
    private:
    TreeMap <H, List <C>> _hoteles;
    HashMap < C, H) _clientes;
    public:
    agencia(); // Constructor
void aloja (const C &c, const H &h);
void desaloja (const C &c);
const H alojamiento(const C & c);
Lista<H> listado_hoteles();
List <C> huespedes(const H &h);
};
template < class H, class C>
agencia <H,C> :: agencia() {};
```

```
template <class H, class C>
agencia <H, C>:: aloja (const C & c, const H &h) {
    if (!_hoteles.contains (h)) {
        _hoteles.insert (h, List <C>);
    }
    if (_clientes.contains <c>) {
        H hotel =- clientes.at (c); // Hotel en el que esta
```

(1)

(1)

```
lista <L> :: Iterator i = aux.begin()
bool encontrado = false;
while (!encontrado && i != aux.end()) {
        if (c == it.elem()) {
                encontrado = true;
                aux.erase(it);
        }
        else {
                it.next();
        }
}
-clientes.insert (c,h);
lista <C> clientes = _hoteles.at(h);
_clientes.push_back(cliente);
_hoteles.insert(h, clientes);
```

```
template <class H, class C>
agencia <H,C> :: desaloja (const C &c) {
  if (_clientes.contains(c)) {
      H hotel = _clientes.at(hotel);
     -clientes.erase(c);
      list <C> aux ~~clientes~~ = _hoteles.at(hotel);
      bool encontrado = false;
      while (!encontrado && it != aux.end()) {
          if(it.elem() == hotel) {
                encontrado = true;
                aux.erase(it);
          }
          else {
                it.next();
          }
      _hoteles.insert(hotel, aux);
```

```
template <class H, class C>
const H
agencia<H,C> :: alojamiento (const C &c){

    if(!clientes.contains(c)){
        throw NoHayCliente();
    }

    return _cliente.at(c); // Me da el hotel
                           desde este el cliente
```

```
template <class h, class C>
lista <H>  listado_hoteles(){
    lista <H> lhoteles;
    TreeMap <H,C> :: Iterador it = _hoteles.begin();
    while (it != _hoteles.end()){
        lhoteles.push_back(it.key());
        it++;
    }

  return lhoteles;
}
```

```
template <class H, class C>
lista <C> agencia<C,H> :: huespedes (const H &h){
    if(_hoteles.contains(h)
        return _hoteles.at(h);
    }
    else{
        throw NoHayHotel();
    }
```