Estructuras de Datos y Algoritmos

Grados en Ingeniería Informática, de Computadores y del Software (grupo A)

Examen Final, 3 de Junio de 2014.

1. **(3 puntos)** Alonso Rodríguez tiene que hacer la compra de la semana. Ha hecho una lista de n productos que quiere comprar. En su barrio hay m supermercados en cada uno de los cuales se dispone de todos esos productos. Pero como es un comprador compulsivo no quiere comprar más de tres productos en cada uno de los supermercados ya que así pasa más tiempo comprando (se puede suponer que $n \leq 3m$). Dispone de una lista de precios de los productos en cada uno de los supermercados. Se pide diseñar un algoritmo de vuelta atrás que permita a Alonso decidir cómo hacer la compra de forma que compre todo lo que necesita y que el coste total sea mínimo.

2. **(3 puntos)** Dado un árbol binario, en cuya raíz se encuentra situado un tesoro y cuyos nodos internos pueden contener un dragón o no contener nada, se pide diseñar un algoritmo que nos indique la hoja del árbol cuyo camino hasta la raíz tenga el menor número de dragones. En caso de que existan varios caminos con el mismo número de dragones, el algoritmo devolverá el que se encuentre más a la izquierda de todos ellos.

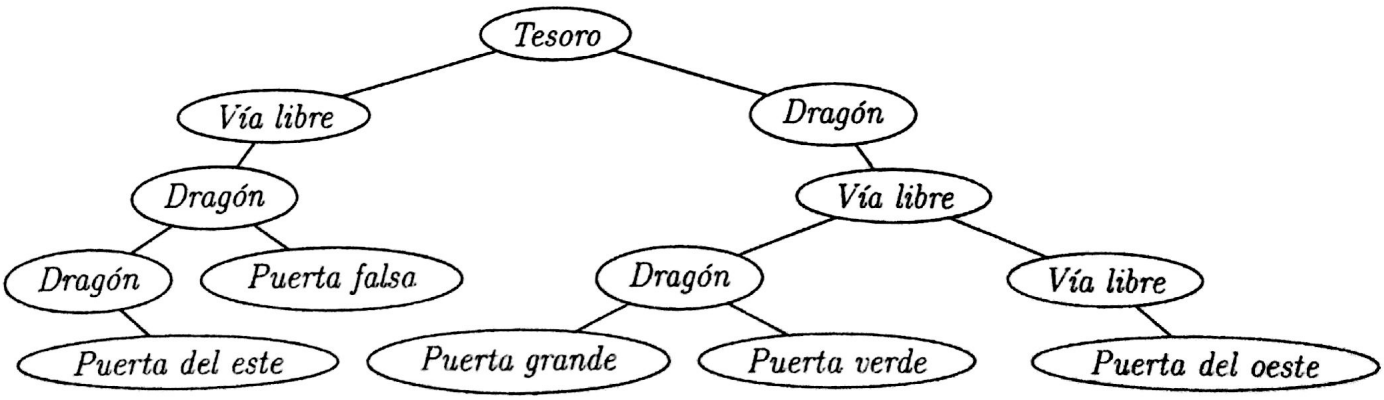
   Para ello implementar una función que reciba un árbol binario cuyos nodos almacenan cadenas de caracteres:

   - La raíz tiene la cadena *Tesoro*
   - Los nodos internos tienen la cadena *Dragón* para indicar que en el nodo hay un dragón o la cadena *Vía libre* para indicar que no hay dragón.
   - En cada hoja se almacena un identificador que no puede estar repetido.

   y devuelva el identificador de la hoja del camino seleccionado. El árbol tiene como mínimo un nodo raíz y un nodo hoja diferente de la raíz. La operación no se implementa como parte de ningún TAD.

   El coste de la operación implementada debe ser $\mathcal{O}(n)$.

   Por ejemplo, dado el siguiente árbol el algoritmo devolverá la cadena de caracteres *Puerta falsa.*



3. **(4 puntos)** Eres un prestigioso diseñador de videojuegos, y estás trabajando ahora en uno llamado *EspacioMatic*, en el que habrá naves espaciales con distintos módulos (motores, cabinas, escudos, láseres, etcétera). Necesitarás implementar, como poco, las siguientes operaciones:

   - *EspacioMatic*: inicializa el sistema de juego.
   - *nuevaNave*: añade una nueva nave espacial (vacía) al sistema, con el identificador numérico proporcionado. Si se intenta usar un identificador ya existente produce error. No devuelve nada.
   - *equipaNave*: dado el identificador de una nave, un nombre de módulo (una cadena como "motor", "cabina", "láser", etcétera) y un nivel de funcionalidad (un entero ≥ 1), añade el módulo correspondiente a esa nave con el nivel indicado. Si esa nave ya tenía ese módulo, se suma el nuevo nivel al anterior (esto permite reparar módulos de naves). No devuelve nada.
   - *estropeaNave*: dado el identificador de una nave, y un nombre de módulo, resta 1 al nivel de ese módulo en esa nave (asumiendo que tuviese un nivel > 0). Devuelve `true` si el módulo existía y tenía un nivel positivo antes de hacer la resta, ó `false` si ha sido imposible restar nivel ya que el módulo no existía o estaba ya a 0.
   - *navesDefectuosas*: devuelve una lista de identificadores de naves que tienen uno o más módulos completamente estropeados (con un nivel de 0).
   - *modulosNave*: dado el identificador de una nave, devuelve una lista con los nombres de los módulos que tiene equipados (tengan el nivel que tengan), ordenada alfabéticamente.

   Desarrolla en C++ una implementación de la clase *EspacioMatic* basada en otros TADs conocidos, optimizando la complejidad temporal de las operaciones. En cada operación, indica también, de forma razonada, su complejidad.

JUNIO 2014

① Supermercado ($n \leq 3m$)

- n productos
- m supermercados
- no más de 3 productos/super
- Precios
- Coste total mínimo

Planteamiento

La solución es un vector de n posiciones (una por producto) que guarde en qué super lo compramos.

- Requisitos explícitos

$(X_1, \dots X_n)$ $\quad 0 \leq X_i < m$

- Requisitos implícitos
  - El mismo super no está repetido más de 3 veces.
  - Coste total mínimo.

- Parámetros

(int) - N (nº de productos) (Global)

(int) - M (nº de supers) (Global)

-(float) precios [N][M] (Global)
fila columna

- int sol [ ]
- int k etapa
- int solOpt [ ]
- float costeAct, costeMin.
- int marcas [M]

marcas se inicializa a 0 y cuando compramos en el super i hacemos marcas[i]++

```
void compra (int sol [], int k, int solOpt [],
             float & costeAct, float & costeMin, int marcas [ ]){
  for (int super = 0 ; super < M; super++){
    sol [k] = super;
    costeAct = costeAct + precio [k] [super]
    marcas [super]++;
    if(marcas [super] <= 3)   // es válida
      if (k == N-1)
        if (costeAct < costeMin){
          costeMin = costeAct;
          copia sol (solOpt, sol);
        }
      else{
        compra (sol, k+1, solOpt, costeAct, ....);
      }
    después del if  costeAct = costeAct - precio [k] [super]
                    marcas [super] --;
  }
```

25/5/15 SLF

ESPACIOMATIC (EJ. 3 - Ex. Junio 2014)

Estructura de datos

Nave
- Módulos
  - Nombre (string)
  - Nivel (int)
- Identificador (int)

HashMap con clave Identificador de la nave y valor un TreeMap cuyas claves sean nombres de módulos y cuyos valores son los niveles.

```
class EspacioMatic {
  private:
    HashMap <int, TreeMap <string, int >>
        -naves;
    HashMap <int, Lista <string>>          TreeMap <string, int>>
        -defectuosas;

  public:
    EspacioMatic();
    void nuevaNave (const int &id);
    void equipaNave (const int &id, const string &mod,
                     const int &nivel);
    void estropeaNave (const int &id, const string &mod);
    lista <int> navesDefectuosas();
    lista <string> modulosNave (const int &id);
};
```

```
void EspacioMatic::EspacioMatic(){};

void EspacioMatic::nuevaNave(const int &id){
    if(_naves.contains(id)){
        throw naveExistente();
    }
    _naves.insert(id, new TreeMap<string,int>());
}

void EspacioMatic::equipaNave(const int &id, _nave
                              const string &mod, const int &nivel){

    if(_naves.contains(id){ //Si está la nave
    TreeMap<string,int>modulos = _naves.at(id)
*   if(!modulo.contains(mod){
        Si no está lo insertamos
        modulos.insert(mod, 0) //Inicializamos a 0

    _modulos.insert(mod, modulos.at(mod) + nivel); //insert. modulos
    _naves.insert(id, modulos); //Actualizamos naves

* if(modulos.at(mod) == 0)   // Está defectuoso y  nivel 0,
    if(~~!defectuosos.at(id).contains(mod)~~){ presuponemos que está
      TreeMap<string, int>
      _modDef = _defectuosos.at(id);
      _modDef.erase(mod); //Lo quitamos
      _defectuoso.insert(id, _modDef); //Lo insertamos en defectuosos
    if(_defectuosos.at(id.empty())
        _defectuosos.erase(id); // Si ya no está defectuosa, la quitamos de defectuosos
```

25/5/15 SLF

bool EspacioMatic :: estropeaNave (const int & id,
const string & mod);

if (!naves.contains(id)) // No existe la nave cte
throw NoExisteNave();
if (!naves.at(id).contains(mod)) cte
return false; // La nave no tiene el mod
TreeMap <string, int> modulos = _naves.at(id); cte
if (modulos.at(mod) == 0) // Modulo defect los
return false;
Int nivel = modulos.at(mod); los
modulos.insert(mod, nivel - 1); los // Decrementamos el nivel
_naves.insert(id, modulos); // Actualizamos las naves
cte
if (nivel - 1 == 0) // Si el modulo queda defectuoso
if (!_defectuosos.contains(id)) cte
modulos = _defectuosos.at(id); cte
modulos.insert(mod, 0); los
_defectuosos.insert(id, modulos); cte
else
TreeMap <string, int> mod2; // Creamos uno nuevo (modulo)
mod2.insert(mod, 0); // Insertamos cte
_defectuosos.insert(id, mod2); // Actualizamos
cte
return true;
}

Complejidad:
O(log n)

```
Lista <int> EspacioMatic::NavesDefectuosas(){
    Lista <int> l;                                          cte
    HashMap <int, TreeMap <string, int>>::const_Iterator it=
                                        _defectuosas.cbegin();
    while (it != _defectuosas.cend()){                      cte
        l.push_back(it.key());                              cte
        it++;                                               cte
    }
    return l;
}
```

// Iterador al principio

// lo metemos al final de la lista

vuelta (n = datos)

Complejidad: O(n)

```
Lista <string> EspacioMatic::modulosNave(const int &id){
    Lista <string> l;
    TreeMap <string, int>::const_Iterator it;

    if(!_naves.contains(id))                                cte
        throw NoExisteNave();

    it = _naves.at(id).cbegin();                            cte
    while (it != _naves.at(id).cend()){
        l.push_back(it.key);
        it++;
    }
```

O(n)

Como antes

25/5/15

MULTAMATIC

Planteamiento de las estruct. de datos:

Para los tramos:

template < class T (tramos), class C (cámaras) >

```
struct _infoTramo{
    C camIni;
    C camFin;
    int seg;
    Lista<String> _cochesMultados;   // Matrículas
};
HashMap<T, _infoTramo> _tramos;
```

```
class MultaMatic {
private:
    struct infoTramo;
    struct infoCoche;
public:
    insertaTramo
    fotoEntrada
    fotoSalida
    multasPorMatricula
    multasPorTramos
```

```
struct _infoCoche{
    T tramo;
    int nMultas;
    time_t instanteEntrada;
    bool enTramo;
};
```

```
HashMap<string (matrícula), _infoCoche> _coches;
HashMap<C, T> _cámaras; // Asocia cámaras a tramos
HashMap<T, infoTramo> _tramos;
```

~~void insertatramo~~

```
void Multallatic <T, C> insertaTramo (const T & t,
        const C & cini,
        const C & Cfin,
        Const int & minSec) {

  if (-tramos. contains (t)) {
        throw loContiene ();
  }
                if (-camaras.contains(cini) || -camaras.contains (fin)) {
  else {                                    throw ErrorCamara ();
    infoTramo miTramo;
    miTramo. CamaraIni = Cini;
    miTramo. CamaraFin = CFin;
    miTramo. TiempoTr = minSec;

  -Tramo . insert (t, miTramo);
  - Camaras. insert (cini, t);
  - Camaras. insert (cfin, t);
```

```
template < class T, class C >
void Multallatic < T, C > :: InfoEntrada (const C & cam,
                                const string & matricula,
                                time_t time) {

  if (!- camaras. contains (cam))
  |   Throw Excepcion;
   infoCoche Aux;
  aux.tramo = - camaras. at (cam);
  aux.TiempoEntrada = time;
if ( -coches. contains (matricula))
   aux.Multas = = -coches. at (matricula). nMultas;

else
    aux. NMultas = 0;
    aux. enTramo = true;

  -coches. insert (matricula, aux);
}
```

1/6/15 SLF

```
template <class T, class C>
void MultaMatic :: FotoSalida (const C& cam,
                               const string & matricula,
                               time_t time){

  if (!_camaras.contains(cam)){
        throw NoEsta();
  }
// if (!_coches.contains(matricula)){
//       throw NoEstaCoche();          // No le podemos poner multa si que el coche no esta
// }
  if (_coches.contains(matricula)){
    InfoCoches aux = _coches.at(matricula);
    InfoTramo auxt = _tramos.at(_camaras.at(cam));
    if (aux.entramo == true && _camaras.at(cam) != auxtramo)
      if((time - aux.tiempoEntrada) > auxt.tiempoMin))
        aux.nMultas++;
        auxt.multados.pushback(matricula);
        aux.entramo = false;
    _coches.insert(matricula, aux);
    _tramos.insert(_tramos.at(_camaras.at(cam)), auxt);
  }
  else
    throw error();
```

// Actualizo el tramo sólo si ponemos multa nueva

```
template <class T, class C>
int MultaMatic<T,C>::MultasPorMatricula(const string &matricula) const
{
    if(!_coches.contains(matricula))
        throw CocheNoExiste();

    return _coches.at(matricula).nMultas;
}
```

```
template <class T, class C>
Lista<string> MultaMatic<T,C>::MultasPorTramo(const T&t) const
{
    if(!_tramos.contains(t))
        throw TramoNoExiste();

    return _tramos.at(t).multados;
```