Aplicaciones de tipos abstractos de datos

Alberto Verdejo

Dpto. de Sistemas Informáticos y Computación
Universidad Complutense de Madrid

Sistema de control de bostezos

Severino del Pino, profesor de arameo de la Universidad Imponente, ha detectado problemas de aburrimiento entre su numeroso alumnado.

Su colega Tadeo de la Tecla, del Departamento de Informática, ha ofrecido ayudarle diseñando un sistema informático de control de bostezos. Tadeo propone un sistema con las siguientes operaciones:

- crear un sistema de bostezos vacío,
- registrar un nuevo bostezo en el sistema,
- borrar del sistema todos los bostezos registrados de un alumno dado,
- consultar el número de bostezos de un alumno registrados en el sistema, y
- calcular la lista de todos los alumnos que tengan tres o más bostezos registrados, ordenada por alumnos.

Sistema de control de bostezos. Implementación

```
template <class alumno>
class bostezos {
private:
   std::map<alumno, int> tabla;
public:
   bostezos() {}

   void bostezar(alumno const& a) {  // O(log A)
      ++tabla[a];
   }

   int cantidad(alumno const& a) const {  // O(log A)
      auto it = tabla.find(a);
      if (it == tabla.end())
         return 0;
      else
         return it->second;
   }
```

Sistema de control de bostezos. Implementación

```
    void borrar(alumno const& a) {  // O(log A)
        tabla.erase(a);
    }

    std::vector<alumno> lista_negra() const {  // O(A)
        std::vector<alumno> resultado;
        for (auto const& par : tabla) {
            if (par.second >= 3)
                resultado.push_back(par.first);
        }
        return resultado;
    }
};
```

Sistema de control de faltas

Al profesor Severino le ha encantado el sistema diseñado, pero ahora cree que su funcionalidad está algo limitada, porque también le gustaría registrar otros tipos de faltas que sus alumnos suelen cometer: tirarle tizas, hacer volar aviones de papel, escapar por la ventana, etc.

Además le gustaría poder decidir en cada momento cuál es el número de faltas que ha tenido que cometer como mínimo un alumno para aparecer en la lista negra.

Extender el sistema adecuadamente.

Sistema de control de faltas. Implementación

```
#define clave first
#define valor second

template <class alumno, class falta>
class bostezos_y_mas {
private:
   using info_alumno = std::pair<int, std::unordered_map<falta, int>>;
   std::map<alumno, info_alumno> tabla;
public:
   bostezos_y_mas() {}

   void registrar(falta const& f, alumno const& a) {  // O(log A)
      info_alumno & info = tabla[a];
      ++info.first;
      ++info.second[f];
   }
```

Sistema de control de faltas. Implementación

```
void borrar_falta(falta const& f, alumno const& a) {  // O(log A)
   auto itA = tabla.find(a);
   if (itA != tabla.end()) {
      auto itF = itA->valor.second.find(f);
      if (itF != itA->valor.second.end()) {
         itA->valor.first -= itF->valor;
         if (itA->valor.first > 0) {
            itA->valor.second.erase(itF);
         } else {
            tabla.erase(itA);
         }
      }
   }
}

void borrar(alumno const& a) {  // O(log A)
   tabla.erase(a);
}
```

Sistema de control de faltas. Implementación

```
int cantidad_falta(falta const& f, alumno const& a) const {  // O(log A)
   auto itA = tabla.find(a);
   if (itA != tabla.end()) {
      auto itF = itA->valor.second.find(f);
      if (itF != itA->valor.second.end()) {
         return itF->valor;
      } else
         return 0;
   } else
      return 0;
}

int cantidad(alumno const& a) const {  // O(log A)
   auto itA = tabla.find(a);
   if (itA == tabla.end())
      return 0;
   else
      return itA->valor.first;
}
```

Sistema de control de faltas. Implementación

```
    std::vector<alumno> lista_negra_falta(falta const& f, int n) const { // O(A)
        std::vector<alumno> resultado;
        for (auto const& par : tabla) {
            auto itF = par.valor.second.find(f);
            if (itF != par.valor.second.end() && itF->valor >= n)
                resultado.push_back(par.clave);
        }
        return resultado;
    }

    std::vector<alumno> lista_negra(int n) const {  // O(A)
        std::vector<alumno> resultado;
        for (auto const& par : tabla) {
            if (par.valor.first >= n)
                resultado.push_back(par.clave);
        }
        return resultado;
    }
};
```

Consultorio médico

La dirección del Hospital Central de Fanfanisflán quiere informatizar su consultorio médico por medio de un sistema que permita realizar al menos las siguientes operaciones:

- generar un consultorio vacío, sin ninguna información,
- dar de alta a un nuevo médico que antes no figuraba en el consultorio,
- hacer que un paciente se ponga a la espera para ser atendido por un médico, el cual debe estar dado de alta en el consultorio,
- consultar el paciente a quien le toca el turno para ser atendido por un médico, el cual debe estar dado de alta y debe tener algún paciente que le haya pedido consulta,
- atender al paciente que le toque por parte de un médico que debe estar dado de alta y tener algún paciente que le haya pedido consulta, y
- reconocer si hay o no pacientes a la espera de ser atendidos por un médico, el cual debe estar dado de alta.

Consultorio médico. Implementación

```
using medico = std::string;
using paciente = std::string;

class consultorio : private std::unordered_map<medico, std::queue<paciente>> {
public:
   void alta_medico(medico const& m) {  // O(1)
      auto res = insert({ m, std::queue<paciente>() });
      if (!res.second)
         throw std::domain_error(m + " repetido");
   }

   void pedir_consulta(paciente const& p, medico const& m) {  // O(1)
      auto it = find(m);
      if (it == end())
         throw std::domain_error(m + " no existe");
      else
         it->valor.push(p);
   }
```

Consultorio médico. Implementación

```
bool tiene_pacientes(medico const& m) const {  // O(1)
   auto it = find(m);
   if (it == end())
      throw std::domain_error(m + " no existe");
   else
      return !it->valor.empty();
}

paciente const& siguiente_paciente(medico const& m) const {  // O(1)
   auto it = find(m);
   if (it == end())
      throw std::domain_error(m + " no existe");
   else if (it->valor.empty())
      throw std::domain_error(m + " sin pacientes");
   else
      return it->valor.front();
}
```

Consultorio médico. Implementación

```
    void atender_consulta(medico const& m) {  // O(1)
        auto it = find(m);
        if (it == end())
            throw std::domain_error(m + " no existe");
        else if (it->valor.empty())
            throw std::domain_error(m + " sin pacientes");
        else
            it->valor.pop();
    }
};
```

Consultorio médico 2

Los pacientes a veces se cansan de esperar y abandonan la consulta antes de ser atendidos, lo cual causa inconvenientes después cuando les llega el turno. Se desea añadir al sistema de consultorio anterior la siguiente operación:

- registrar que un paciente que estaba esperando a ser atendido por un médico deja de estarlo antes de que le llegue el turno.

Consultorio médico 2. Implementación

```
class consultorio2  {
private:
   std::unordered_map<medico, std::list<paciente>> meds;
   using info_paciente = std::unordered_map<medico,
                                            std::list<paciente>::iterator>;
   std::unordered_map<paciente, info_paciente> pacs;

public:
   void alta_medico(medico const& m) {  // O(1)
      auto res = meds.insert({ m, std::list<paciente>{} });
      if (!res.second)
         throw std::domain_error(m + " repetido");
   }
```

Consultorio médico 2. Implementación

```
void pedir_consulta(paciente const& p, medico const& m) {  // O(1)
   auto itM = meds.find(m);
   if (itM == meds.end())
      throw std::domain_error(m + " no existe");
   else {
      auto & ip = pacs[p];
      auto itMP = ip.find(m);
      if (itMP != ip.end())
         throw std::domain_error(p + " repite con " + m);
      else {
         itM->valor.push_back(p);
         ip[m] = --itM->valor.end();
      }
   }
}
```

Consultorio médico 2. Implementación

```
bool tiene_pacientes(medico const& m) const {  // O(1)
   auto it = meds.find(m);
   if (it == meds.end())
      throw std::domain_error(m + " no existe");
   else
      return !it->valor.empty();
}

paciente const& siguiente_paciente(medico const& m) const {  // O(1)
   auto it = meds.find(m);
   if (it == meds.end())
      throw std::domain_error(m + " no existe");
   else if (it->valor.empty())
      throw std::domain_error(m + " sin pacientes");
   else
      return it->valor.front();
}
```

Consultorio médico 2. Implementación

```
void atender_consulta(medico const& m) {  // O(1)
   auto it = meds.find(m);
   if (it == meds.end())
      throw std::domain_error(m + " no existe");
   else if (it->valor.empty())
      throw std::domain_error(m + " sin pacientes");
   else {
      pacs[it->valor.front()].erase(m);
      it->valor.pop_front();
   }
}
```

Consultorio médico 2. Implementación

```
    void abandonar_consulta(paciente const& p, medico const& m) {
        auto itM = meds.find(m);
        if (itM == meds.end())
            throw std::domain_error(m + " no existe");
        else {
            auto itP = pacs.find(p);
            if (itP == pacs.end())
                throw std::domain_error(p + " no pidio consulta");
            else {
                auto & ip = itP->valor;
                auto itMP = ip.find(m);
                if (itMP == ip.end())
                    throw std::domain_error(p + " no espera a " + m);
                else {
                    itM->valor.erase(itMP->valor);
                    ip.erase(m);
                }
            }
        }
    }
};
```

Videoclub

El videoclub *Me lo veo todo* desea gestionar los alquileres de películas mediante un sistema informático. El sistema debe incluir al menos las siguientes operaciones:

- crear un videoclub vacío,
- añadir una copia de una película haciendo posible su alquiler,
- incluir a una persona como nuevo socio del videoclub, suponiendo que no lo es,
- establecer que un socio alquila una copia de una película, suponiendo que hay copias disponibles de la película,
- devolver una copia de una película por parte de un socio que la tenía alquilada,
- eliminar todas las copias de una película, siempre que hayan sido devueltas,
- determinar si una persona es socio del videoclub,
- calcular el número de copias existentes de una película (alquiladas o no),
- calcular el número de copias alquiladas de una película, y
- determinar si una persona tiene alquilada una copia de una película.

Videoclub. Implementación

```
using socio = std::string;
using pelicula = std::string;

class videoclub {
private:
   using info_peli = std::pair<int, int>;   // < núm_copias, alquiladas >
   std::unordered_map<pelicula, info_peli> pelis;
   using alquiladas_t = std::unordered_set<pelicula>;
   std::unordered_map<socio, alquiladas_t> socios;
public:
   void insertar_peli(pelicula const& pel) {  // O(1)
      ++pelis[pel].first;
   }

   void alta_socio(socio const& soc) {  // O(1)
      auto res = socios.insert({ soc, alquiladas_t() });
      if (!res.second)
         throw std::domain_error("el socio " + soc + " ya existe");
   }
```

Videoclub. Implementación

```
void alquilar(socio const& soc, pelicula const& pel) {  // O(1)
   auto itS = socios.find(soc);
   if (itS != socios.end()) {
      if (itS->valor.count(pel) > 0)
         throw std::domain_error("el socio " + soc + " ya tiene " + pel);
      else {
         auto itP = pelis.find(pel);
         if (itP != pelis.end()) {
            if (itP->valor.first == itP->valor.second)
               throw std::domain_error("no hay copias disponibles de " + pel);
            else {
               ++itP->valor.second;
               itS->valor.insert(pel);
            }
         } else
            throw std::domain_error("la peli " + pel + " no existe");
      }
   } else
      throw std::domain_error("el socio " + soc + " no existe");
}
```

Videoclub. Implementación

```
void devolver(socio const& soc, pelicula const& pel) {  // O(1)
   auto itS = socios.find(soc);
   if (itS != socios.end()) {
      if (itS->valor.count(pel) == 0)
         throw std::domain_error("el socio " + soc + " no tiene " + pel);
      else {
         --pelis[pel].second;
         itS->valor.erase(pel);
      }
   } else
      throw std::domain_error("el socio " + soc + " no existe");
}
```

Videoclub. Implementación

```
void borrar_peli(pelicula const& pel) {  // O(1)
   auto itP = pelis.find(pel);
   if (itP != pelis.end()) {
      if (itP->valor.second > 0)
         throw std::domain_error("hay copias alquiladas de " + pel);
      else {
         pelis.erase(pel);
      }
   } else
      throw std::domain_error("la peli " + pel + " no existe");
}

bool es_socio(socio const& soc) const {  // O(1)
   return socios.count(soc) > 0;
}
```

Videoclub. Implementación

```
    unsigned int copias(pelicula const& pel) const {  // O(1)
        auto itP = pelis.find(pel);
        if (itP != pelis.end())
            return itP->valor.first;
        else
            return 0;
    }

    unsigned int alquiladas(pelicula const& pel) const {  // O(1)
        auto itP = pelis.find(pel);
        if (itP != pelis.end())
            return itP->valor.second;
        else
            return 0;
    }

    bool tiene_alquilada(socio const& soc, pelicula const& pel) const {  // O(1)
        auto itS = socios.find(soc);
        return (itS != socios.end()) && (itS->valor.count(pel) > 0);
    }
};
```

Problemas

- 34 - Carnet por puntos 1
- 35 - Carnet por puntos 2
- 36 - Un consultorio médico
- 37 - Autoescuela
- 38 - iPud
- 39 - Venta de libros por internet