Facultad de Informática

Informática de Gestión

Estructuras de Datos y de la Información

Segundo Parcial, Junio 2003

Ejercicio 1 (7 puntos)

Se desea diseñar un tipo abstracto de datos para representar un club de fans de un equipo de fútbol. El nivel de fanatismo de los fans se puede expresar como un número natural: cuanto más grande, mayor es su fanatismo. Cuando un fan nuevo desea entrar en el club, sólo puede hacerlo si su nivel de fanatismo es mayor o igual que el de algún miembro del club. Con el tiempo los fans pueden incrementar su nivel de fanatismo si el equipo gana muchos partidos o pueden decrementarlo desencantados por una mala racha. Cada cierto tiempo el presidente revisa el nivel de fanatismo de los fans pertenecientes al club y expulsa a todos aquellos que tengan un nivel de fanatismo excesivamente bajo o excesivamente alto, ya que desea miembros fervientes pero que no provoquen altercados. Se pide:

1. (3 puntos) Especificar algebraicamente el tipo abstracto de datos $CLUB$ con las operaciones pedidas a continuación (y las auxiliares que consideres necesarias) suponiendo disponible un tipo FAN que almacena la información sobre un fan con una operación de igualdad disponible. Se desea disponer de las siguientes operaciones:

   - *vacio*: crea un club de fans vacio.
   - $insertar(c, f, n)$: si f no pertenecía ya al club c lo inserta con nivel de fanatismo n siempre que cumpla las reglas de admisión; si ya pertenecía, el nuevo nivel de fanatismo de f pasa a ser n.
   - $pertenece(c, f)$: comprueba si f pertenece al club c.
   - $nivel(c, f)$: si f pertenece al club c devuelve su nivel de fanatismo.
   - $revisar(c, n, m)$: borra del club c a los fans con nivel de fanatismo k tal que $k < n$ o $k > m$.
   - $masfanaticos(f, n)$: obtiene la lista de los fans con un fanatismo mayor o igual a n.

   Las operaciones *insertar* y *nivel* deben ser parciales y el resto totales.

2. (4 puntos) Se define la siguiente clase C++ para implementar el TAD $CLUB$, supuesto que se dispone de un tipo $TFan$:

```
class TClub{
  private:
    TArbus<int,TLista<TFan> > club;
    ...

  public:
    TClub();
    bool pertenece(const TFan& f) const;
```

```
    TLista<TFan> masfanaticos(int n) const;
    void revisar(int n, int m);
    void insertar(const TFan& f, int n) throw (ENoAdmisible);
    int nivel(const TFan& f) throw (ENoPertenece);
    ...
}
```

Un club está representado por un árbol binario de búsqueda. El valor asociado a cada nivel es una lista de fans con ese nivel de fanatismo.

Se pide codificar las operaciones `pertenece`, `masfanaticos`, `revisar` e `insertar`, suponiendo que el resto de las operaciones (`nivel`, constructora por defecto etc) ya están definidas. Para ello puedes definir funciones o procedimientos auxiliares privados.

Estudiar el coste de `pertenece` y `masfanaticos` en términos del número de fans (ignorando las copias de estructuras). ¿Influiría en dicho coste el hecho de que el árbol estuviera equilibrado?

Utilizar las interfaces de los árboles binarios de búsqueda y de las listas que se muestran al final del enunciado.

Ejercicio 2 (3 puntos)

La pitonisa Lola utiliza las cartas de la baraja española (1 a 12 de oros, copas, espadas y bastos) para leer el futuro a sus clientes. Para ello pide al cliente que corte la baraja y que del montón que se ha quedado vaya sacando cartas de una en una en el orden que quiera y se las vaya dando hasta acabar con dicho montón. La pitonisa las va poniendo sobre la mesa siguiendo un ritual predeterminado antes de comenzar a interpretarlas. El ritual consta de dos partes: dada una sucesión C de cartas

- Paso 1: la transforma en una sucesión de cartas C' reemplazando cada sucesión de cartas consecutivas que no sean figura (10, 11 y 12) por su inversa.
- Paso 2: transforma C' en otra sucesión C'' tomando sucesivamente la primera carta de C', luego la última, luego la segunda, luego la penúltima etc.

Por ejemplo si $C = 1oros\ 2copas\ 7bastos\ 12espadas\ 10oros\ 5copas\ 6bastos$ entonces:

$$C' = 7bastos\ 2copas\ 1oros\ 12espadas\ 10oros\ 6bastos\ 5copas$$
$$C'' = 7bastos\ 5copas\ 2copas\ 6bastos\ 1oros\ \ 10oros\ 12espadas$$

Suponiendo que se dispone de un tipo $TCarta$ con una función booleana $esFigura$ que determina si la carta es una figura o no, se pide:

- Teniendo en cuenta que C y C'' son secuencias de cartas ($TSecuencia < TCarta >$) y cómo se lleva a cabo el segundo paso, decidir qué estructura resulta más adecuada utilizar para representar la sucesión intermedia C'.
- Codificar por separado cada uno de los pasos del ritual. Puedes utilizar como estructuras auxiliares las vistas en clase, que se muestran al final del enunciado.

Interfaces

```
template <class TElem>
class TPila {
  private:
    ...
  public:
     TPila( );
     TPila( const TPila<TElem>& );
     ~TPila( );
     TPila<TElem>& operator=( const TPila<TElem>& );

     void apila(const TElem&);

  // Lanza una excepcion EAccesoIndebido si la pila es vacia
     const TElem& cima( ) const throw (EAccesoIndebido);

  // Lanza una excepcion EAccesoIndebido si la pila es vacia
     void desapila( ) throw (EAccesoIndebido);
     bool esVacia( ) const;

}


template <class TElem>
class TCola {
  private:
     ...
  public:
     TCola( );
     TCola( const TCola<TElem>& c);
     ~TCola( );
     TCola<TElem>& operator=( const TCola<TElem>& c);

     void anadir(const TElem& x);

     // Lanza una excepcion EAccesoIndebido si la cola es vacia
     const TElem& primero( ) const throw (EAccesoIndebido);

     // Lanza una excepcion EAccesoIndebido si la cola es vacia
     void eliminar( ) throw (EAccesoIndebido);

     bool esVacia( ) const;
}
```

```
template <class TElem>
class TLista {
  private:   ...
  public:
     TLista( );
     TLista( const TLista<TElem>& );
     TLista (const TElem& );         // lista unitaria
     ~TLista( );
     TLista<TElem>& operator=( const TLista<TElem>& );

    void concatenar(const TLista<TElem>& l);
    void anadirppio(const TElem& e);
    void anadirfinal(const TElem& e);
    int longitud() const;
    //lanzan una excepcion si la lista es vacia
    const TElem& prim()  throw (EAccesoIndebido);
    const TElem& ult()  throw (EAccesoIndebido);
    void resto()  throw (EAccesoIndebido);
    void eult()  throw (EAccesoIndebido);
    bool esVacia() const;

    //lanzan una excepcion si n<=0 o n>longitud()
    const TElem& acceso(int n) throw (EAccesoIndebido);
    void modificar(int n,const TElem& e) throw (EAccesoIndebido);
    void borrar (int n) throw (EAccesoIndebido);

    //lanza una excepcion si n<=0 o n>longitud()+1
    void insertar (int n,const TElem& e) throw (EAccesoIndebido);
    int buscar(const TElem& e); //devuelve 0 si no esta
    bool esta(const TElem& e);
}

template <class TElem>
class TSecuencia {
  private:
   ...
  public:
     TSecuencia( );
     TSecuencia( const TSecuencia<TElem>& );
     ~TSecuencia( );
     TSecuencia<TElem>& operator=( const TSecuencia<TElem>& );

     void inserta(const TElem&);
     void borra( ) throw (EAccesoIndebido);
     const TElem& actual( ) const throw (EAccesoIndebido);
     void avanza( ) throw (EAccesoIndebido);
     void reinicia( );
     bool esFin( ) const;
     bool esVacio( ) const;
}
```

```
template<class TClave, class TValor>
class TArbus{
 private:
  ...
 public:
   TArbus();
   TArbus(const TArbus<TClave,TValor>&);
   ~TArbus();
   TArbus<TClave,TValor>& operator=(const TArbus<TClave,TValor>&);

   void inserta(const TClave&,const TValor&);
   void borra(const TClave&);
   const TValor& consulta(const TClave&) const throw (EClaveErronea);
   bool esta(const TClave&) const;
   bool esVacio() const;

   //consultoras del valor y la clave de la raiz si no es vacio
   const TValor& consultaVRaiz() const throw (EArbolVacio);
   const TClave& consultaCRaiz() const thorw (EArbolVacio);

   //devuelven los hijos izquierdo y derecho si no es vacio
   TArbus<TClave,TValor> hijoIz() const throw (EArbolVacio);
   TArbus<TClave,TValor> hijoDer() const throw (EArbolVacio);

   //construye un abb a partir de un hijo izquierdo,
   //un hijo derecho (ambos abbs), y una pareja clave-valor siempre
   //que se cumpla la propiedad de ordenacion exigida para los abbs
   //en caso contrario lanza una excepcion
   void enraizar(const TArbus<TClave,TValor>&,
                 const TClave &, const TValor &,
                 const TArbus<TClave,TValor>&) throw (ENoOrdenado);

   //devuelve la menor clave si existe
   const TClave& min() const throw (ENoExiste);

   //recorridos
   TLista<TValor> inorden() const;
   TLista<TValor> preorden() const;
   TLista<TValor> postorden() const;
}
```

```
template <class TElem>
class TCPrio
{ private:
    ...
  public:

  // Tamaño inicial por defecto
     static const int MAX = 64 - 1;

     TCPrio( int capacidad );
     TCPrio( const TCPrio<TElem>& );
     ~TCPrio( );
     TCPrio<TElem>& operator=( const TCPrio<TElem>& );


     void anyade(const TElem&);

     // Lanza la excepción ECPrioVacia si la cola está vacía
     const TElem& min( ) const throw (ECPrioVacia);

     // Lanza la excepción ECPrioVacia si la cola está vacía
     void quitaMin( ) throw (ECPrioVacia);

     bool esVacio( ) const;

}


template <class TClave, class TValor>
class TTabla {
  private:
    ...
  public:
     TTabla( );
     TTabla( const TTabla<TClave,TValor>& );
     ~TTabla( );
     TTabla<TClave,TValor>& operator=( const TTabla<TClave,TValor>& );

     void inserta( const TClave&, const TValor& );

     // si la clave no está, la tabla no se modifica
     void borra( const TClave& );

     // Lanza la excepción EClaveErronea si la tabla no contiene dicha clave
     const TValor& consulta( const TClave& ) const throw (EClaveErronea);
     bool esta( const TClave& ) const;
     bool esVacio( ) const;

}
```