Universidad de Huelva

Estructuras de Datos II
(I.T. Informática de Gestión)
(I.T. Informática de Sistemas)

Convocatoria de Junio de 2005

La Rábida, 17 de junio de 2005

NOTAS

1. Se podrán definir funciones o acciones auxiliares para resolver los problemas pedidos
2. Sólo se pueden utilizar los Tipos Abstractos de Datos que se proporcionan en el Anexo. Además, para cada uno, sólo es posible usar los métodos indicados.
3. Es obligatorio incluir comentarios en los algoritmos que escribas indicando qué se hace en los bloques de código más relevantes.

Cuestión 1 — 1 punto

Partiendo de la especificación algebraica del TAD Árbol Binario (ver Anexo), escribir las ecuaciones necesarias para definir la semántica de las siguientes operaciones:

(a) elementosMenores: elemento arbin → conjunto<elemento>
// devuelve un conjunto con los elementos más pequeños que el elemento que se pasa por parámetro

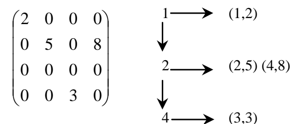
(b) esEquilibrado: arbin → booleano
// devuelve verdadero si el árbol está equilibrado en el sentido AVL. Recuerda que la función altura
// es parcial y sólo está definida para árboles no vacíos

Cuestión 2 — 1,75 puntos

Una ***matriz dispersa*** es una matriz de grandes dimensiones (por ej., 1000 x 1000) en la que la mayor parte de sus elementos valen cero. La forma más simple de implementar una matriz es usando un array de dos dimensiones. Sin embargo, de esta manera se desperdicia mucho espacio en memoria, dado que la mayor parte de las posiciones del array serán cero. Siguiendo con el ejemplo anterior, necesitaríamos 1.000.000 de posiciones para almacenar, pongamos, 100 valores no nulos.

Con el fin de alcanzar una gestión más eficiente de la memoria, se puede implementar una matriz dispersa usando listas. En ese caso, tendríamos una **lista ordenada** con los números de fila de la matriz que almacenan valores distintos de cero y, para cada fila, la lista de pares formados por el número de columna y el elemento almacenado, cuando es distinto de cero, **ordenada por número de columna**. La principal ventaja de esta representación es que **no se almacenan los ceros**.

A continuación aparece un ejemplo de matriz dispersa de enteros y su representación usando estas listas:



La especificación y la representación escogidas para realizar la implementación del TAD Matriz Dispersa son:

```
template <typename T>
class MatrizDispersa{
    public:
        MatrizDispersa (int n, int m) throw (DimensionErroneaExcepcion)
        /* Si n ó m son negativos, lanza la excepción DimensionErroneaExcepcion. En caso contrario, crea una matriz
           dispersa de n filas y m columnas. */
        void asignar (int i, int j, const T& e) throw (DimensionErroneaExcepcion)
        /* Si i < 1 ó i > numFil o j < 1 ó j > numCol lanza la excepción DimensionErroneaExcepcion. En otro caso,
           asigna el elemento e a la fila i columna j. */
    private:
        int numFil, numCol;
        Lista<Fila<T> > matriz;
}
```

Se pide que implementes el método **asignar**, que inserta un valor en la posición dada (*i*, *j*) de la matriz. Recuerda que como no se almacenan ceros en las listas de columnas, la asignación de un 0 a una posición que almacene un valor no nulo supone la eliminación del par correspondiente de la lista de columnas y, si fuera la última columna, la eliminación también de la fila. Por ejemplo, en la figura anterior, la ejecución de *asignar(2, 4, 0)* supone la eliminación del par (4, 8) de la lista de columnas de la fila 2. Sin embargo, al realizar *asignar(4, 3, 0)* se deben eliminar tanto el par (3, 3) de la lista de columnas como la fila 4 de la lista de filas.

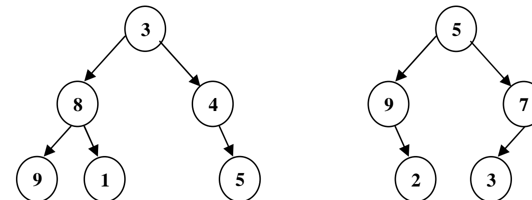
Las clases Fila y Columna del Anexo, representan los objetos almacenados respectivamente en cada una de las listas usadas para la implementación del TAD Matriz Dispersa. La interfaz del TAD Lista también está disponible en el Anexo. **Es obligatorio el uso de iteradores** para realizar los recorridos y búsquedas en las listas.

Cuestión 3 — 1 punto

Realizar una función booleana que, dados dos árboles binarios, indique si el primero es *inferior* al segundo. Diremos que un árbol binario es inferior a otro, si los elementos del primero, en los nodos coincidentes en posición, son menores que los del segundo.

```
template <typename T>
bool esInferior(const Arbin<T>& a1, const Arbin<T>& a2)
```

Por ejemplo, en estos dos árboles, el primero es inferior al segundo:

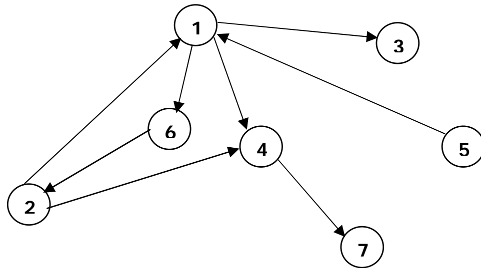


Cuestión 4 — 1,25 puntos

Dado un grafo *G* y dos vértices del mismo, *u* y *v*, escribe una función que devuelva cuál es la longitud media de todos los caminos simples que hay entre *u* y *v* en *G*. La *longitud de un camino* se define como el número de vértices que forman dicho camino menos uno.

float longMedia(**const** Grafo& g, Vertice u, Vertice v)

En el grafo representado en la figura, la longitud media de los caminos entre los vértices 1 y 7 es **3**, ya que, hay dos caminos (1-4-7 y 1-6-2-4-7) de longitudes 2 y 4 respectivamente.



NOTA: Se aconseja el uso de la **recursividad** para resolver esta cuestión.

Anexo

(A) Especificación algebraica de los géneros **arbin** y **conjunto**

```
espec arbolesBinarios
    usa booleanos, enteros
    parámetro formal
        género elemento
        operaciones
            _≤_ , _≥_, _<_ , _>_ , _==_ , _≠_ :
                elemento elemento → booleano
    fpf
    género arbin
    operaciones
        Δ: → arbin
        _ < _ , _ >: elemento arbin arbin → arbin
        parcial raíz: arbin → elemento
        parcial subIzq, subDer: arbin → arbin
        vacío?: arbin → booleano
        parcial altura: arbin → entero
        ecuaciones
            …
fsepec
```

```
espec conjuntos
    usa booleanos, naturales
    parámetro formal
        género elemento
    fpf
    genero conjunto
    operaciones
        ∅: → conjunto
        poner, quitar: elemento conjunto → conjunto
        _ ∪ _ , _ ∩ _ : conjunto conjunto → conjunto
        _ ∈ _ : elemento conjunto → booleano
        esVacío: conjunto → booleano
        cardinal: conjunto → natural
        ecuaciones
            …
fsepec
```

(B) Definición de las clases **Fila** y **Columna**

```
template <typename T>
class Fila{
    public:
        Fila (int i, Lista<Columna<T > > l);
        int getFila ();
        Lista<Columna<T> >& getCols();
    private:
        int f;
        Lista<Columna<T> > cols;
}
```

```
template <typename T>
class Columna{
    public:
        Columna (int j, int v);
        int getColumna ();
        const T& getDato();
        void setDato(const T& v);
    private:
        int c;
        T valor;
}
```

(C) Definición de la clase **Lista**

```
template <typename T>
class Lista {
  public:
    class Iterador {
      friend class Lista<T>;
      public:
        void avanzar(const Lista& L)
                throw(PosicionErroneaExcepcion);
        const T& observar(const Lista& L) const
                throw(PosicionErroneaExcepcion);
        int posicion(const Lista& L) const;
        bool operator!=(const Iterador & der) const;
        bool operator==(const Iterador & der) const;
      private:
        …
    };
    Lista();
    Lista(const Lista& list);
    bool esVacia() const;
    void anadirIzq(const T& objeto);
    void anadirDch(const T& objeto);
    void eliminarIzq() throw(ListaVaciaExcepcion);
    void eliminarDch() throw(ListaVaciaExcepcion);
    const T& observarIzq() const
        throw(ListaVaciaExcepcion);
    const T& observarDch() const
        throw(ListaVaciaExcepcion);
    // Métodos propios de la lista con iterador
    void insertar(const Iterador& it, const T& objeto);
    void eliminar(const Iterador& it)
        throw(PosicionErroneaExcepcion);
    void modificar(const Iterador& it, const T& objeto)
        throw(PosicionErroneaExcepcion);
    const T& observar(const Iterador& it) const
        throw(PosicionErroneaExcepcion);
    Iterador principio() const;
    Iterador final() const;
    ~Lista();
  private:
    …
};
```

(D) Definición de la clase **Arbin**

```
template <typename T>
class Arbin {
  public:
    Arbin();
    Arbin(const T& objeto, const Arbin& ai, const Arbin& ad);
    Arbin(const Arbin& a);
    Arbin& operator=(const Arbin& a);
    const T& getRaiz() const throw(ArbolVacioExcepcion);
    Arbin subIzq() const throw(ArbolVacioExcepcion);
    Arbin subDer() const throw(ArbolVacioExcepcion);
    bool esVacio() const;
    int altura() const throw(ArbolVacioExcepcion);
  private:
    …
};
```

(E) Definición de las clases **Conjunto** y **Grafo**

```
template <typename T>
class Conjunto {
  public:
        Conjunto();
        bool esVacio() const;
        int cardinalidad() const;
        void anadir(const T& objeto);
        void eliminar(const T& objeto);
        const T& quitar();
        bool pertenece(const T& objeto) const;
  private:
        …
};
```

```
class Grafo {
  public:
    Grafo(int n);
    int numVertices() const;
    Void insertarArista(const Arista& a);
    const ConjAristas& aristas() const;
    const ConjVertices& adyacentes(Vertice v) const;
    ~Grafo();
  private:
    ...
};
typedef int Vertice;
typedef Conjunto<Arista> ConjAristas;
typedef Conjunto<Vertice> ConjVertices;
```