

TDA VECTOR DINAMICO

/**

* **of** file vector_dinamico.h

* **brief** fichero cabecera del TDA vector dinamico

*

* se crea un vector con capacidad de crecer y decrecer

*

*/

ifndef _vectorDinamico_h

define _vectorDinamico_h

/**

* **brief** TDA Vector_Dinamico

*

* Una instancia **de** v del tipo de dato abstracto

* **de** Vector_Dinamico sobre el tipo **de** float es un

* array 1-D de un determinado tamaño **de** n , que

* puede crecer y decrecer a petición del usuario.

* Lo podemos representar como:

*

* $\{v[0], v[1], \dots, v[n-1]\}$

* donde

* $v[i]$ es el valor almacenado en la posición i del vector

*

* La eficiencia en espacio es **de** $O(n)$.

*

* Un ejemplo de uso puede verse en:

* **include** ejemplo_vector_dinamico.cpp

*

*/

```
class Vector_Dinamico {
```

private:

1. 2. 3.

→ Page rep Vector_Dinamico Rep del TDA Vector_Dinamico

✱

* 0 section inv Vector_Dinamico Invariante de Representation.

4

- Un objeto válido **de** r del TDA Vector_Dinamico debe cumplir

* - ∂C v. elementos ≥ 0

- **PC** v. datos apunta a una zona de memoria con

- ↳ capacidad para albergar ∞ elementos valores

* de tipo float

5

d) section f a Vector_Dinamico Función de abstracción

- * Un objeto válido de rep del TDA Vector_Dinamico

\mathbf{x} representa al vector de tamaño n

$$\{v.\text{datos}[0], v.\text{datos}[1], \dots, v.\text{datos}[v.\text{elementos}-1]\}$$

* /

float * datos; // ** < Apunta a los elementos del vector ** /

`int nElementos;` // * < Indica el número de elementos en el array
`int * pDatos;` // * > Indica la dirección de memoria de los datos

public:

//----- Constructores -----

/**

• @brief Constructor por defecto

• @param n indica el número de componentes iniciales

• reservados para el vector

• @note Este constructor también corresponde al de por defecto

*/

Vector_Dinamico (int n=0);

Vector_Dinamico (const Vector_Dinamico & original);

//----- Destructor -----

~Vector_Dinamico();

//----- Otras funciones -----

/**

• @brief Número de componentes del vector

• @return Devuelve el número de componentes que puede almacenar en cada instante el vector

• @see resize()

*/

int size() const;

/**

• @brief Acceso a un elemento

• @param i es la posición del vector donde está el elemento. $0 \leq i < \text{size}()$

• @return Devuelve la referencia al elemento. Por tanto puede usarse para almacenar un

↓ Valor en esa posición.

*/

float & operator [] (int i);

/*

↓ **brief** Acceso a un elemento de un vector constante

↓ **param** i es la posición del vector donde está

↓ el elemento. $0 \leq i < \text{size}()$

↓ **return** Devuelve la referencia al elemento. Se supone que el vector no se puede modificar y por tanto es acceso solo de lectura.

*/

const float & operator [] (int i) const;

/*

↓ **brief** Redimensión del vector

↓ **param** n es el nuevo tamaño del vector. $n \geq 0$

↓ **post** Los valores almacenados antes de la redimensión no se pierden (excepto los que se salen del nuevo rango de índices)

*/

void resize (int n);

/*

↓ **brief** Operador de asignación

*/

VectorDinamico & operator = (const VectorDinamico & original);

};

#endif /*_vectorDinamico_h_*/

Ejemplo: Clase vector dinámico (1/2).

vector_dinamico.h	vector_dinamico.cpp
<pre> #ifndef _vectorDinamico_h #define _vectorDinamico_h class Vector_Dinamico { private: float * datos; int nelementos; public: // ----- Constructores ----- Vector_Dinamico(int n); Vector_Dinamico(const Vector_Dinamico& original); // ----- Destructor ----- ~Vector_Dinamico(); // ----- Otras funciones ----- int size() const; float& operator[] (int i); const float& operator[] (int i) const; void resize(int n); Vector_Dinamico& operator= (const Vector_Dinamico& original); }; #endif /* _vectorDinamico_h */ </pre>	<pre> #include <cassert> #include <vector_dinamico.h> Vector_Dinamico::Vector_Dinamico(int n) { assert(n>=0); if (n>0) datos= new float[n]; nelementos= n; } /* ----- */ Vector_Dinamico::Vector_Dinamico (const Vector_Dinamico& original) { nelementos= original.nelementos; if (nelementos>0) { datos= new float[nelementos]; for (int i=0; i<nelementos;++i) datos[i]= original.datos[i]; } else datos=0; } </pre>

Ejemplo: Clase vector dinámico (2/2).

vector_dinamico.cpp	vector_dinamico.cpp
<pre> Vector_Dinamico::~Vector_Dinamico() { if (nelementos>0) delete[] datos; } int Vector_Dinamico::size() const { return nelementos; } float& Vector_Dinamico::operator[] (int i) { assert (0<=i && i<nelementos); return datos[i]; } const float& Vector_Dinamico::operator[] (int i) const { assert (0<=i && i<nelementos); return datos[i]; } Vector_Dinamico& Vector_Dinamico::operator= (const Vector_Dinamico& original) { if (this! = &original) { if (nelementos>0) delete[] datos; nelementos= original.nelementos; datos= new float[nelementos]; for (int i=0; i<nelementos;++i) datos[i]= original.datos[i]; } return *this; } </pre>	<pre> void Vector_Dinamico::resize(int n) { assert (n>=0); if (n!=nelementos) { if (n!=0) { float * nuevos_datos; nuevos_datos= new float[n]; if (nelementos>0) { int minimo; minimo= nelementos<n?nelementos:n; for (int i= 0; i<minimo;++i) nuevos_datos[i]= datos[i]; delete[] datos; } nelementos= n; datos= nuevos_datos; } else { if (nelementos>0) delete[] datos; datos= 0; nelementos= 0; } } } </pre>

```
#include <iostream>
```

```
/* Ejemplo de uso */
```

```
#include <vector_dinamico.h>
```

```
using namespace std;
```

```
void cargar_indices (vector_dinamico & v)
```

```
{  
    for (int i = 0; i < v.size(); ++i)  
        v[i] = i;  
}
```

```
float maximo (const vector_dinamico & v)
```

```
{  
    float max;  
    if (v.size() == 0) {  
        cerr << "Upps! máximo de ??? asignamos zero" << endl;  
        max = 0.0;  
    }  
    else {  
        max = v[0];  
        for (int i = 1; i < v.size(); ++i)  
            if (max < v[i])  
                max = v[i];  
    }
```

```
    return max;
```

```
}
```

```
int main()
```

```
{
```

```
    vector_dinamico vec;
```

```
cargar_indices (vec);
```

```
cout << "Maximo de " << vec.size() << "elementos:"  
      << maximo (vec) << endl;
```

```
vec.resize (10);
```

```
cargar_indices (vec);
```

```
cout << "Maximo de" << vec.size() << "elementos:"  
      << maximo (vec) << endl;
```

```
return 0;
```

```
{
```


TDA CONJUNTO DE REALES

```
/**
 * @file conjunto_reales.h
 * @brief Fichero Cabecera del TDA Conjunto Reales
 *
 */
#define _conjunto_reales_h
#define _conjunto_reales_h

#include <vector_dinamico.h>
#include <cassert>

/**
 * @brief TDA Conjunto Reales
 *
 * Una instancia de c del tipo de dato abstracto
 * @c Conjunto Reales es un conjunto de números de tipo float.
 *
 * El número de elementos del conjunto se denomina cardinal
 * o tamaño del conjunto. Un conjunto de tamaño cero se
 * denomina vacío.
 *
 * Lo podemos representar como
 *
 *  $\{e_1, e_2, e_3, \dots, e_n, e_n\}$ 
 *
 * donde n es el número de elementos del conjunto.
 *
 * La eficiencia en espacio es de  $O(n)$ 
 *
 */
```

class Conjunto_Reales {

private:

- /*
 - * @page repConjunto_Reales Rep del TDA Conjunto_Reales
 - *
 - * @section invConjunto_Reales Invariante de Representación
 - *
 - * Un objeto válido de rep del TDA Conjunto_Reales debe cumplir
 - * - @c rep.v.size() >= rep.nelementos
 - * - @c rep.nelementos >= 0
 - * - @c rep.v[i] < rep.v[j] para todo i, j tal que
 - * i < j < rep.nelementos
 - * @section faConjuntoReales Función de abstracción
 - *
 - * Un objeto válido de rep del TDA ~~Conjunto_Reales~~ representa al vector {rep.v[0], ..., rep.v[rep.nelementos]}

*/
Vector_Dinamico v; /* Almacena los elementos del conjunto */
int nelementos; /* Número de posiciones de v usadas */

- /*
 - * @brief Localizador de una posición en @c v
 - *
 - * @param val es el valor del elemento a localizar en la matriz.
 - * @retval pos, posición donde se encuentra el @a valor (si está) ó la posición donde debería insertarse (si no está)
 - * @return si el valor @a val está en el vector devuelve true
 - * @note La eficiencia es logarítmica (usa búsqueda binaria)

*/
bool posicion_elemento (int & pos, float val) const;

public:

Conjunto Reales(): elementos(0) {}

// Conjunto Reales (const Conjunto-Reales & c);

// ~Conjunto-Reales();

// Conjunto Reales & operator = (const Conjunto-Reales & c);

/**

* @brief Añadir un elemento

* @param f valor a insertar en el conjunto

* @return true si el número de elementos ha aumentado y
false si el elemento ya estaba en el conjunto.

*/

bool insertar(float f);

/**

* @brief Eliminar un elemento

* @param f valor a eliminar del conjunto

* @return true si el número de elementos ha disminuido y
false si el elemento no estuviera en el conjunto

*/

bool borrar(float f);

/**

* @brief Consultar la existencia de un elemento

* @param f valor a consultar en el conjunto

* @return true si el elemento está en el conjunto, false
en caso contrario

*/

*/

bool pertenece(float f) const { int pos;

return posicion elemento(pos, f); }

/**

* @brief Valor del i -ésimo elemento

*

* @param i indica el elemento del conjunto que queremos obtener

* @pre $0 \leq i < \text{size}()$

* @return devuelve el valor del i -ésimo elemento

*/

```
float elemento (int i) const { assert ( $0 \leq i$  &&  $i < \text{v.size}()$ );  
    return v[i]; }
```

/**

* @brief Conjunto vacío

* @return true si el conjunto está vacío ($\text{size}() == 0$)

*/

```
bool vacío() const { return elementos == 0; }
```

/**

* @brief Cardinal del conjunto

* @return Devuelve el número de elementos del conjunto

*/

```
int size() const { return elementos; }
```

```
};
```

```
#endif /* _conjunto_reales_h */
```

Ejemplo: Clase Conjunto_Reales (1/2).

conjunto_reales.h	conjunto_reales.cpp
<pre> #ifndef _conjunto_reales_h #define _conjunto_reales_h #include <vector_dinamico.h> #include <cassert> class Conjunto_Reales { private: Vector_Dinamico v; int nelementos; bool posicion_elemento(int& pos, float val) const; public: Conjunto_Reales(): nelementos(0) {} // Conjunto_Reales(const Conjunto_Reales& c); // ~Conjunto_Reales(); // Conjunto_Reales& operator= // (const Conjunto_Reales& c); bool insertar(float f); bool borrar(float f); bool pertenece(float f) const { int pos; return posicion_elemento(pos,f); } float elemento(int i) const { { assert(0<=i && i<=v.size()); return v[i]; } } bool vacio() const { return nelementos==0; } int size() const { return nelementos; } }; #endif /* _conjunto_reales.h */ </pre>	<pre> #include <cassert> #include <conjunto_reales.h> bool Conjunto_Reales::posicion_elemento(int& pos, float val) const { int izq=0, der=nelementos-1, centro; while (der-izq>=0) { centro=(izq+der)/2; if (val<v[centro]) der=centro-1; else if (val>v[centro]) izq=centro+1; else { pos=centro; return true; } } pos= izq; return false; } </pre>

Ejemplo: Clase Conjunto_Reales (2/2).

conjunto_reales.cpp	conjunto_reales.cpp
<pre> bool Conjunto_Reales::insertar(float f) { int pos; if (posicion_elemento(pos,f)) return false; else { if (v.size()==nelementos) if (v.size()==0) v.resize(1); else v.resize(2*v.size()); for (int j=nelementos; j>pos; --j) v[j]=v[j-1]; v[pos]= f; nelementos++; return true; } } </pre>	<pre> bool Conjunto_Reales::borrar(float f) { int pos; if (posicion_elemento(pos,f)) { nelementos--; for (int j=pos;j<nelementos;++j) v[j]=v[j+1]; if (nelementos<v.size()/4) v.resize(v.size()/2); return true; } else return false; } </pre>

T.DA VECTOR DISPERSO

/**
* *file* vector_disperso.h

* *brief* Archivo cabecera del TDA vector disperso

*
* Se crea un vector con múltiples elementos dispersos en un
* rango amplio de su índice
*
*/

#ifndef - vector_disperso.h

#define - vector_disperso.h

/**

*
*

* Una instancia *de* v del tipo de dato abstracto *de* Vector Disperso

* sobre el tipo *de* float es un array 1-D con índices enteros

* positivos sin limitación de rango.

* Este tipo de dato se diseña especialmente para los problemas

* en los que el vector almacena en la mayoría de sus posiciones

* un valor predeterminado *de* d , mientras que modifica un

* pequeño conjunto de ellas. Lo podemos representar como:

* $\{(i[0], v[0]), (i[1], v[1]), \dots, (i[n-1], v[n-1]), (x, d)\}$

*
*

* La eficiencia en espacio es *de* $O(n)$, donde *de* n es el

* número de posiciones del vector que almacenan un valor

* distinto de *de* d

*
*

*
*

*/

Class Vector_Disperso {

private:

/* \varnothing page repVector_Disperso Rep del TDA Vector_Disperso

* \varnothing section invVector_Disperso Invariante de la representación

*

* Un objeto válido rep del TDA Vector_Disperso debe cumplir

* - \varnothing rep.nelementos ≥ 0

* - \varnothing rep.reservados ≥ 0

* - \varnothing rep.datos apunta a una zona de memoria con

* capacidad para albergar \varnothing reservados valores de tipo
 \varnothing Elemento

* - \varnothing $0 \leq \text{rep.datos}[i].\text{indice} < \text{rep.datos}[j].\text{indice}$ para
todo \varnothing i, j , tal que \varnothing $0 \leq i < j < \text{rep.nelementos}$

* \varnothing section faVector_Disperso Función de abstracción

*

* Un objeto válido rep del TDA Vector_Disperso representa

* al vector:

* { (rep.datos[0].indice, rep.datos[0].valor), ... ,

* (rep.datos[rep.nelementos-1].indice, rep.datos[rep.nelementos-1]

* .valor), (*, rep.valor_por_defecto) }

*

* /

/**

✓ **Obtén** Posición en el vector

* El tipo **Elemento** almacena una pareja de valores que indica el índice y el valor correspondiente de una posición en el vector

*

*/

struct Elemento {

int indice; /* < Posición en el vector */

float valor; /* < Valor almacenado en la posición de índice del vector */

};

Elemento * datos; /* < Matriz de posiciones usadas en el vector */

int nelementos; /* < Número de elementos usados en la matriz de datos */

int reservados; /* < Número de posiciones reservadas en de datos */

float valor_por_defecto; /* < Valor en las posiciones fuera de de datos */

/**

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
20	0	0	0	0	0	0	0	0	50	0	0	0	0	-1	0	0	0	0	0	0	0	0	28	0

Vector disperso con 4 posiciones válidas:

índice $i=0$ con valor 20 || índice $i=14$ con valor -1

índice $i=9$ con valor 50 || índice $i=23$ con valor 28

Representación:

{ (0, 20), (9, 50), (14, -1), (23, 28), (0) }

Elemento 1

Elemento 2

Elemento 3

Elemento 4

valor por defecto

pos=0	pos=1	pos=2	pos=3	pos=4	pos=5	pos=6
0 20	9 50	14 -1	23 28			

datos →

índice: datos[pos].indice

valores: datos[pos].valor

nelementos = 4

reservados = 7

valor_por_defecto = 0.0

Ejemplo: Clase Vector Disperso (1/3).

<code>vector_disperso.h</code>	<code>vector_disperso.cpp</code>
<pre> #ifndef _vector_disperso_h #define _vector_disperso_h class Vector_Disperso { private: struct Elemento { int indice; float valor; }; Elemento * datos; int nelementos; int reservados; float valor_por_defecto; bool posicion_indice(int& pos, int i) const; void resize(int n); public: Vector_Disperso(float default_valor); Vector_Disperso(const Vector_Disperso& orig); ~Vector_Disperso(); Vector_Disperso& operator= (const Vector_Disperso& original); float get_default() const; float get(int i) const; void set(int i, float f); int num_no_default() const; void datos_posicion (int i, int &indice, float& valor) const; }; #endif /* _vector_disperso.h */ </pre>	<pre> bool Vector_Disperso::posicion_indice(int& pos, int i) const { int izq=0, der=nelementos-1, centro; while (der-izq>=0) { centro=(izq+der)/2; if (i<datos[centro].indice) der=centro-1; else if (i>datos[centro].indice) izq=centro+1; else { pos=centro; return true; } } pos= izq; return false; } void Vector_Disperso::resize(int n) { assert(n>=nelementos && n>0); Elemento * nuevos_datos= new Elemento[n]; for (int j= 0; j<nelementos ;j++) nuevos_datos[j]= datos[j]; delete[] datos; datos= nuevos_datos; reservados=n; } </pre>

Ejemplo: Clase Vector Disperso (2/3).

<code>vector_disperso.cpp</code>	<code>vector_disperso.cpp</code>
<pre> Vector_Disperso::Vector_Disperso(float defecto) { datos=new Elemento[1]; nelementos=0; reservados=1; valor_por_defecto= defecto; } Vector_Disperso::Vector_Disperso(const Vector_Disperso& orig) { valor_por_defecto= orig.valor_por_defecto; reservados= nelementos= orig.nelementos; if (nelementos>0) { datos= new Elemento[nelementos]; for (int i=0; i<nelementos;++i) datos[i]= orig.datos[i]; } else datos=0; } </pre>	<pre> Vector_Disperso::Vector_Disperso() { datos=0; nelementos=reservados=0; valor_por_defecto= 0.0; } Vector_Disperso::~Vector_Disperso() { if (reservados>0) delete[] datos; } Vector_Disperso& Vector_Disperso::operator= (const Vector_Disperso& orig) { if (this!= &orig) { if (reservados>0) delete[] datos; valor_por_defecto= orig.valor_por_defecto; reservados= nelementos= orig.nelementos; if (nelementos>0) { datos= new Elemento[nelementos]; for (int i=0; i<nelementos;++i) datos[i]= orig.datos[i]; } else datos=0; } return *this; } </pre>

Ejemplo: Clase Vector Disperso (3/3).

<i>vector_disperso.cpp</i>	<i>vector_disperso.cpp</i>
<pre> int Vector_Disperso::num_no_default() const { return nelementos; } void Vector_Disperso::datos_posicion(int i, int& indice, float& valor) const { assert (0<=i && i<nelementos); indice= datos[i].indice; valor= datos[i].valor; } float Vector_Disperso::get_default() const { return valor_por_defecto; } float Vector_Disperso::get(int i) const { int pos; if (posicion_indice(pos,i)) return datos[pos].valor; else return valor_por_defecto; } </pre>	<pre> void Vector_Disperso::set(int i, float f) { int pos; // Está en pos if (posicion_indice(pos,i)) { if (f!=valor_por_defecto) datos[pos].valor= f; else { // hay que eliminarlo nelementos= nelementos-1; for (int j=pos;j<nelementos;++j) datos[j]=datos[j+1]; if (nelementos<reservados/4) resize(reservados/2); } } else { // No está en el vector (estaría en pos) if (f!=valor_por_defecto) { if (nelementos==reservados) resize(reservados*2); for (int j=nelementos; j>pos; --j) datos[j]= datos[j-1]; datos[pos].indice= i; datos[pos].valor= f; ++nelementos; } } } </pre>