

Implementación y uso de TADs

Alberto Verdejo

Dpto. de Sistemas Informáticos y Computación
Universidad Complutense de Madrid

- R. Peña. *Diseño de Programas: Formalismo y Abstracción*. Tercera edición. Pearson Prentice-Hall, 2005.
Capítulo 5
- N. Martí Oliet, Y. Ortega Mallén y A. Verdejo. *Estructuras de datos y métodos algorítmicos: 213 ejercicios resueltos*. Segunda edición. Garceta, 2013.
Capítulos 1 y 2
- M. A. Weiss. *Data Structures and Algorithm Analysis in C++*. Fourth edition. Pearson, 2014.
Capítulo 1

Problema: Dado un número $x > 0$, se suman los cuadrados de sus dígitos, para obtener x_1 . Se realiza la misma operación a x_1 para obtener x_2 , y así sucesivamente hasta que ocurra una de las dos cosas siguientes:

- se llega a 1, y entonces se dice que el número es *feliz*, o
- nunca se llega a 1, y se dice entonces que el número es *infeliz*.

Problema: Dado un número $x > 0$, se suman los cuadrados de sus dígitos, para obtener x_1 . Se realiza la misma operación a x_1 para obtener x_2 , y así sucesivamente hasta que ocurra una de las dos cosas siguientes:

- se llega a 1, y entonces se dice que el número es *feliz*, o
- nunca se llega a 1, y se dice entonces que el número es *infeliz*.

Ejemplos:

- el 7 es feliz: $7 \rightarrow 49 \rightarrow 97 \rightarrow 130 \rightarrow 10 \rightarrow 1$

Problema: Dado un número $x > 0$, se suman los cuadrados de sus dígitos, para obtener x_1 . Se realiza la misma operación a x_1 para obtener x_2 , y así sucesivamente hasta que ocurra una de las dos cosas siguientes:

- se llega a 1, y entonces se dice que el número es *feliz*, o
- nunca se llega a 1, y se dice entonces que el número es *infeliz*.


Ejemplos:

- el 7 es feliz: $7 \rightarrow 49 \rightarrow 97 \rightarrow 130 \rightarrow 10 \rightarrow 1$
- el 2019 es feliz: $2019 \rightarrow 86 \rightarrow 100 \rightarrow 1$

Problema: Dado un número $x > 0$, se suman los cuadrados de sus dígitos, para obtener x_1 . Se realiza la misma operación a x_1 para obtener x_2 , y así sucesivamente hasta que ocurra una de las dos cosas siguientes:

- se llega a 1, y entonces se dice que el número es *feliz*, o
- nunca se llega a 1, y se dice entonces que el número es *infeliz*.

Ejemplos:

- el 7 es feliz: $7 \rightarrow 49 \rightarrow 97 \rightarrow 130 \rightarrow 10 \rightarrow 1$
- el 2019 es feliz: $2019 \rightarrow 86 \rightarrow 100 \rightarrow 1$
- el 38 es infeliz: $38 \rightarrow 73 \rightarrow 58 \rightarrow 89 \rightarrow 145 \rightarrow 42 \rightarrow 20 \rightarrow 4 \rightarrow 16 \rightarrow 37$


```
#include <iostream>
#include <set>
using namespace std;

int siguiente(int n) { // n > 0
    int suma = 0;
    while (n > 0) {
        int digito = n % 10;
        suma += digito * digito;
        n /= 10; // avance de dígito
    }
    return suma;
}

bool happy(int n) {
    set<int> cjto; // vacío inicialmente
    while (n != 1 && cjto.count(n) == 0) {
        cjto.insert(n);
        n = siguiente(n); // paso al siguiente
    }
    return n == 1;
}
```

- La idea fundamental de los *tipos abstractos de datos* (TADs) es separar de forma estricta la representación del uso de los datos de un tipo. Un tipo no solamente consta de un conjunto de valores sino también de un conjunto de operaciones para la creación, modificación y manipulación de dichos valores.
- Por una parte, tenemos la representación del tipo de datos en términos de tipos básicos o de otros tipos ya conocidos, así como la implementación sobre dicha representación de las operaciones asociadas al tipo de datos, que constituyen la *interfaz* de ese tipo.
- Por otra parte, la única forma de usar los datos del tipo abstracto es invocando adecuadamente las operaciones de la interfaz, de manera que nunca se accede a su representación interna, de ahí la abstracción.

El TAD de los conjuntos de números enteros puede tener las siguientes operaciones:

- conjunto vacío, `set`
- insertar un elemento, `void insert(int elem)`
- eliminar un elemento, `void erase(int elem)`
- averiguar si un elemento pertenece al conjunto,
`bool contains(int elem) const`
- averiguar si el conjunto es vacío, `bool empty() const`
- averiguar el cardinal del conjunto, `int size() const`

En conjunto.h:

```
#ifndef conjunto_h
#define conjunto_h

class set {
public:
    set(); // constructor
    ~set(); // destructor
    void insert(int e);
    bool contains(int e) const;
private:
    int contador;
    int capacidad;
    int * datos;
    void amplia();
};

#endif // conjunto_h
```

Caso de estudio: conjuntos

En conjunto.cpp:

```
#include "conjunto.h"

set::set() : contador(0), capacidad(8), datos(new int[capacidad]) {}

set::~set() { delete[] datos; }

void set::insert(int e) {
    if (contador == capacidad)
        amplia();
    datos[contador] = e;
    ++contador;
}

bool set::contains(int e) const {
    int i = 0;
    while (i < contador && datos[i] != e)
        ++i;
    return i < contador;
}
```

En conjunto.cpp:

```
void set::amplia() {  
    int * nuevos = new int[2*capacidad];  
    for (int i = 0; i < capacidad; ++i)  
        nuevos[i] = datos[i];  
    delete[] datos;  
    datos = nuevos;  
    capacidad *= 2;  
}
```

Todo en conjunto.h:

```
#include <stdexcept> // std::domain_error
#include <utility>    // std::move

template <class T>
class set {
public:
    set(); // constructor
    ~set(); // destructor
    void insert(T e);
    bool contains(T e) const;
    void erase(T e);
    bool empty() const;
    int size() const;
private:
    int contador;
    int capacidad;
    T * datos; // sin repeticiones
    void amplia();
};
```

```
template <class T>
set<T>::set() : contador(0), capacidad(8), datos(new T[capacidad]) {}
```

```
template <class T>
set<T>::~~set() { delete[] datos; }
```

```
template <class T>
void set<T>::insert(T e) {
    if (!contains(e)) {
        if (contador == capacidad)
            amplia();
        datos[contador] = e;
        ++contador;
    }
}
```

```
template <class T>
bool set<T>::contains(T e) const {
    int i = 0;
    while (i < contador && datos[i] != e)
        ++i;
    return i < contador;
}
```

```

template <class T>
void set<T>::amplia() {
    T * nuevos = new T[2*capacidad];
    for (int i = 0; i < capacidad; ++i)
        nuevos[i] = std::move(datos[i]);
    delete[] datos;
    datos = nuevos;
    capacidad *= 2;
}

```

```

template <class T>
void set<T>::erase(T e) {
    int i = 0;
    while (i < contador && datos[i] != e)
        ++i;
    if (i < contador) {
        datos[i] = datos[contador-1];
        --contador;
    } else
        throw std::domain_error("El elemento no está");
}

```

```
template <class T>
bool set<T>::empty() const {
    return contador == 0;
}
```

```
template <class T>
int set<T>::size() const {
    return contador;
}
```


En `main.cpp`:

```
#include "conjunto.h"
```

```
bool happy(int n) {  
    set<int> cjto; // vacío inicialmente  
    while (n != 1 && !cjto.contains(n)) {  
        cjto.insert(n);  
        n = siguiente(n); // paso al siguiente  
    }  
    return n == 1;  
}
```

```
void psicoanaliza(int n) {  
    cout << n << " es " << (happy(n) ? "feliz\n" : "infeliz\n");  
}
```

Conjuntos genéricos: instanciación con **string**

```
int main() {  
    set<string> cadenas;  
  
    cadenas.insert("hola");  
    if (cadenas.contains("hola")) cout << "SI\n";  
    else cout << "NO\n";  
  
    try {  
        cadenas.erase("adios");  
    } catch (domain_error const& de) {  
        cout << "ERROR: " << de.what() << '\n';  
    }  
  
    try {  
        cadenas.erase("goodbye");  
    } catch (exception const& e) {};  
  
    cadenas.erase("hola");  
    if (cadenas.contains("hola")) cout << "SI\n";  
    else cout << "NO\n";  
  
    return 0;  
}
```

```
template <class T>
class set {
public:
    set(); // constructor
    set(set<T> const& other); // constructor por copia
    set<T> & operator=(set<T> const& other); // operador de asignación
    ~set(); // destructor
    ...
private:
    ...
    void amplia();
    void libera();
    void copia(set<T> const& other);
};
```

```

// constructor
template <class T>
set<T>::set() : contador(0), capacidad(8), datos(new T[capacidad]) {}

// destructor
template <class T>
set<T>::~~set() { libera(); }

template <class T>
void set<T>::libera() { delete[] datos; }

// constructor por copia
template <class T>
set<T>::set(set<T> const& other) {
    copia(other);
}

```

```
// operador de asignación
template <class T>
set<T> & set<T>::operator=(set<T> const& other) {
    if (this != &other) {
        libera();
        copia(other);
    }
    return *this;
}
```

```
template <class T>
void set<T>::copia(set<T> const& other) {
    capacidad = other.capacidad;
    contador = other.contador;
    datos = new T[capacidad];
    for (int i = 0; i < contador; ++i)
        datos[i] = other.datos[i];
}
```

```
class punto {  
public:  
    punto(int x, int y) : _x(x), _y(y) {}  
    int x() const { return _x; }  
    int y() const { return _y; }  
    bool operator==(punto const& that) const {  
        return _x == that._x && _y == that._y;  
    }  
    bool operator!=(punto const& that) const {  
        return !(*this == that);  
    }  
    void print(std::ostream & out = std::cout) const {  
        out << '(' << _x << ',' << _y << ')';  
    }  
private:  
    int _x, _y; // coordenadas  
};  
  
inline ostream & operator<<(ostream & out, punto const& rhs) {  
    rhs.print(out);  
    return out;  
}
```

```

#include "punto.h"
#include <cassert>    // assert

class rectangulo {
private:
    punto origen, extremo;
public:
    rectangulo(punto ori, int ancho, int alto) :
        rectangulo(ori, { ori.x() + ancho, ori.y() + alto }) {}
    rectangulo(punto ori, punto ex) : origen(ori), extremo(ex) {
        assert(ori.x() < ex.x() && ori.y() < ex.y());
    }
    int area() const {
        return (extremo.x() - origen.x()) * (extremo.y() - origen.y());
    }
    int ancho() const {
        return extremo.x() - origen.x();
    }
    void print(std::ostream & out = std::cout) const {
        out << "de " << origen << " a " << extremo;
    }
}

```

```

    bool operator==(rectangulo const& that) const {
        return origen == that.origen && extremo == that.extremo;
    }
    bool operator!=(rectangulo const& that) const {
        return !(*this == that);
    }
    bool operator<(rectangulo const& that) const {
        return area() < that.area();
    }
    bool operator>(rectangulo const& that) const {
        return that < *this;
    }
};

inline std::ostream & operator<<(std::ostream & out,
                                rectangulo const& rhs) {
    rhs.print(out);
    return out;
}

```


Función genérica para buscar el máximo de un vector

```
#include <functional> // std::less
#include <cassert>     // assert

template <class Object, class Comparator = less<Object>>
Object const& find_max(vector<Object> const& v,
                      Comparator isLessThan = Comparator()) {
    assert(v.size() > 0);

    int maxIndex = 0;
    for (int i = 1; i < v.size(); ++i)
        if (isLessThan(v[maxIndex], v[i]))
            maxIndex = i;

    return v[maxIndex];
}

int main() {
    rectangulo r1({ 0, 0 }, 4, 5);
    rectangulo r2({ 2, 2 }, punto{ 4, 4 });
    rectangulo r3({ 0, 0 }, punto(8, 1));
    vector<rectangulo> vect { r2, r1, r3, { { 2, 0 }, { 12, 1 } } };
    cout << find_max(vect) << '\n';
}
```

```
class compara_ancho {  
public:  
    bool operator()(rectangulo const& uno, rectangulo const& otro) {  
        return uno.ancho() < otro.ancho();  
    }  
};
```

```
rectangulo const& masAncho = find_max(vect, compara_ancho());  
cout << masAncho.area() << '\n';
```

```
cout << find_max(vect,  
    [](rectangulo const& a, rectangulo const& b) {  
        return a > b;  
    })  
    << '\n';
```

- 1 - ¿A qué hora pasa el próximo tren?
- 2 - Una tarde de sábado
- 3 - El conjunto de Mandelbrot
- 4 - Evaluar un polinomio
- 5 - Los k elementos mayores

- ACR 139 - Números cubifinitos
- ACR 185 - Potitos
- ACR 232 - ¡Feliz no cumpleaños!

