Tema 4 Diseño e implementación de TADs Diseño e implementación de TADs

19/10/2021

Ejemplo de las fechas -> Date1A.cpp/.h y Date1B.cpp/.h

La orientación a objetos permite separar las cosas privadas. -> Date2.h/.cpp

Date2 d(30,10,2021) -> Llama al constructor

d.print();

**- Creación de operadores**  Importante para algunos ejercicios: Es definir que hace un operador para la clase en concreto: void operator++(); -> Hace que podamos hacer d++;

void Date2::operator++(){

// Asumimos que no hay años bisiestos

day++;

if (day > daysInMonth(month)){

day = 1;

 month++;

if (month > 12){

month = 1;

year++;

}

}

}

Es muy común tener que crear operadores de orden:

bool Date2::operator<(const Date2& other) const{

if (year < other.year) return true;

else if (year > other.year) return false;

else if (month < other.month) return true;

else if (month > other.month) return false;

else return day < other.day;

}

* **Const**  bool operator<(const Date2& other) const;

El ultimo const quiere decir que este método no modifica los atributos de la clase, es un método ***observador***.

* **Secuencia de inicialización** Sólo se pone en constructores. Sirven para darles un valor a lo que se crea en memoria justo cuando se crea (inicializaciones para cada atributo). Así no se inicializan por ejemplo a 0, y luego tenemos que meterle un valor, directamente se mete el valor.

Date2::Date2(int d,int m,int y) : ***day(d), month(m), year(y*)**{ //throws domain\_error

//day = d; month = m; year = y;

if (!correctDate()) throw domain\_error("Invalid date");

}

Cout << d; // No conoce el operador “<<” para el tipo de d. (Todo esto está en Date2)

* **Crear operador <<:** Es distinto a los otros operadores: los otros se pueden implementar como métodos, pero << no (porque no y punto a). Debe implementarse como ***función externa***. Su sintaxis es:

// Función externa a la clase

ostream& operator<<(ostream& out, const Date2& d){

out << d.day << "/" << d.month << "/" << d.year;

return out;

}

* No lleva el Date2::, podría ir escrita en el main, en lugar de en el .cpp. Es una función que coexiste con las clases.
* En principio no funcionaría, porque estaría intentando acceder a d.day (atributos privados), sin ser parte de la clase. Funciona porque se ha declarado **función amiga** de la clase. Esto no se debe usar mucho.

friend ostream& operator<<(ostream& out, const Date2& d);

Date3 es igual, pero cambia la implementación. Al tener sólo un dato la constructora se vuelve más difícil, algunos operadores son más fáciles, etc. (Echar un vistazo).

* Usar DateTests.cpp para hacer pruebas.

El **tipo 4** (date4) lo simplifica más. A parte de la fecha, añade la información int value. Esto hace que sea mucho más fácil observar y hacer operaciones. Lo malo es que cuanta más info, más complicado mantener toda la información. En este caso, siempre que se llame a una función de modificación, hay que modificar toda la información -> Las observadoras salen beneficiadas, las otras no. Conclusión: meter más información puede ser bueno, pero hay que comprobar si va a merecer la pena al conservar y transmitir esa información.

* ***Clasificación de Operaciones.*** Diapositiva 19.



Ahora un TAD más importante: esta no es la mejor implementación, solo lo mejor con lo que sabemos hasta ahora -> ***TAD CONJUNTO\*. (Set1)***

\*Un conjunto es un ente matemático que contiene unos valores. Se sabe los valores que están y los que no están. No hay repeticiones. Se pueden añadir, quitar y consultar valores.

**Operaciones**:

Constructor: Set();

Destructor: ~Set();

Void add(const T&); -> recibe un parámetro, el elemento a añadir

Void remove(const T&); -> Puede ser booleano(lo ha podido borrar o no = estaba o no)

Bool contains(const T&) const; -> Es observador

**VISTA ABSTRACTA:**

set<int> s;

s.add(5)

s.add(7)

s.add(5)

Da igual cuantas veces haga add(5),

no se va a añadir más veces

(no hay concepto de repeticiones).

**VISTA CONCRETA:**

[……..] -> El constructor crea un array dinámico de X enteros (X una constante que definamos).

nElem = 0; -> Siguiente índice si ocupar

capacidad = 10;

T\* array -> puntero al array que contiene los datos

Cuando hacemos **add(5)**-> nElem = 1

[5,…]

**Add(7)** -> nElem = 2;

[5,7,….]

**Add(3)** -> nElem = 3;

[5,7,3,…]

**Add(5)** ->nElem = 4;[5,7,3,5,…]

* **Ampliar un array:** Crear un array más grande, copiar el array lleno ahí, y borrarlo.

T\* viejo = array;

Capacidad \*= 2;

Array = new T[capacidad];

for(int i = 0; I < nelem)

array[i] = viejo[i];

delete[] viejo;

* **Void Add()** tiene complejidad lineal en el caso peor: dependiendo del nº de elementos (cuando está lleno). Pero en promedio es constante, ya que casi siempre es constante.

----- no he entendido el remove() xd ----- bueno mas o menos si

* **Contains()** Llama a buscar desde 0 hasta el final.
* Todos tienen complejidad lineal. PERO la complejidad NO es lineal en el nº de elementos del conjunto. Es lineal en n’ (elementos totales que se han añadido) que son los elementos del array que incluyen las posibles repeticiones. Por lo tanto solo es viable si estamos seguros de que no habrá muchas repeticiones. Hay que hacer una segunda representación evitando las repeticiones: *Set2*.

Vamos a hacer lo mismo de antes pero ahora evitaremos repeticiones.

* Constructor: El mismo.

***\*Invariante de la representación\**** concepto que hay que saber y a veces escribir. En este caso, el invariante de la rep. Es que no hay repeticiones. Debo construir los métodos de forma que esto siempre se cumpla.

* En Add() habrá que añadir un condicional, si con Contains() encuentra el nº, no lo añade.
* Remove() es más fácil, ya que simplemente borra el elemento cuando lo encuentra, y se trae el último (ya que no hay orden). (lo mismo de antes sin bucle).
* **Complejidad**: Add es lineal en n, siendo n el nº de elementos del conjunto. Remove también es lineal. Es más eficiente que lo de antes ya que no se busca hasta el final: es lineal en una n más pequeña. Contains() también lineal: búsqueda.

En general esta forma es mejor, pero depende de lo que vaya a hacer el programa.

Si ahora lo ordenamos, las búsquedas van a ser más rápidas. Pero, hay que mantener el orden y eso aumentará el coste -> Remove() seguiría siendo lineal: Saber si un elemento está o no, será **logarítmico,** pero desplazar todo el vector para cubrir ese hueco ya sería **lineal**.

* *Set.h* -> TAD set con array dinámico ordenado y sin repeticiones.

La invariante de representación es que no hay repeticiones y está ordenado.

* binSearch() -> Las búsquedas son binarias.
* Add() tiene una complejidad logarítmica (buscar que no esté), seguida de una lineal, por lo que es **lineal.**
* Remove(), explicado arriba.