

TEMA 1 - DISEÑO MODULAR

1.1- Abstracción y especificación: diseño modular

Qué cualidades ha de tener un buen programa?

- 1- Corrección
- 2- Legibilidad
- 3- Eficiencia

Cómo construir programas grandes?

- Descomposición en partes (módulos)
- Especificación
- Usando abstracción

Abstracción --> consiste en olvidar los detalles e identificar cada parte con un concepto conocido.

Descomposición modular

Beneficios --> 1- Hace más comprensibles los programas

- 2- Facilita el diseño y la corrección
- 3- Facilita el análisis de la corrección, eficiencia...
- 4- Facilita la modificación posterior
- 5- Incrementa la reutilización de software
- 6- Facilita el trabajo en equipo

Tipos:

- Módulos con cohesión interna fuerte
 - Los módulos tienen significado por si mismos e interactúan con otros módulos de forma simple
- Módulos con acoplamiento débil
 - independientes: Los cambios en un módulo no afectan al resto de los módulos

Módulos basados en abstracciones:

- Abstracciones funcionales: uso de funciones (auxiliares)
- Especificaciones pre/post

Especificación vs Implementación:

Regla básica: Un cambio de implementación de una función que respete su especificación, no puede modificar la corrección del programa.

- Especificación: contrato entre implementador y el usuario
- Especificación: abstracción de la implementación

1.2- Diseño basado en objetos

Módulos basados en abstracciones:

- Abstracciones funcionales: describen e implementan nuevas operaciones
- **Abstracciones de datos: describen e implementan nuevos tipos de datos, incluyendo estructuras de datos**

Tipos Abstractos de Datos (TADs)

Un tipo se define dando:

- El nombre del tipo y una descripción de lo que es.
- Sus operaciones, incluida su descripción (qué hace, no cómo lo hace)
- Un tipo puede tener varias implementaciones. El tipo es su especificación, **no** su implementación.

TADs e independencia de los módulos

Fase de especificación: Se deciden las operaciones de los TADs y sus especificaciones

Fase de implementación: Se decide una representación y se implementan las operaciones

Clases y objetos

En los lenguajes orientados a objetos:

- Los módulos que definen TADs se llaman **clases**
- Los elementos (constantes y variables) del tipo que define una clase se llaman **objetos**
- Las operaciones del tipo definido por una clase se llaman **métodos**
- Los campos de una clase se llaman **atributos**
- Las clases tienen una parte **privada** y una **pública**
- Las clases pueden ser predefinidas o definidas por nosotros.
- Cada objeto es **propietario** de sus atributos y sus métodos
- Cada método tiene un **parámetro implícito**: su propietario sobre el que se aplica la operación.
- Podemos tener otras funciones que operen sobre el tipo, pero si no están en la clase, no son métodos

La clase Estudiante

```
class Estudiante{  
    /* Define el TAD estudiante caracterizado por un DNI y una  
     * nota (opcional)*/  
  
    private:  
        // Cómo representamos los objetos de la clase  
  
    public:  
        // Constructoras  
  
        Estudiante();  
        // Pre: Cierto  
        // Post: El resultado es un estudiante sin nota con DNI = 0  
        Estudiante (int dni);  
        // Pre: Cierto  
        // Post: El resultado es un estudiante sin nota con DNI = dni
```

Operaciones creadoras o constructoras

Son operaciones para construir objetos nuevos: y se llaman en la declaración de un objeto.

- Tienen el nombre de la clase
- Crean un objeto nuevo.
- Puede haber varias constructoras. Se distinguen por sus parámetros
- La **constructora por defecto** (sin parámetros) crea un objeto nuevo, dando valores por defecto a sus atributos.
- Se llaman al declarar un objeto:

```
Estudiante est1; Estudiante est2(12345678);
```

Métodos de una Clase

Las operaciones de una clase se dividen en

- Constructoras.
- Destructoras: destruyen los objetos (los eliminan de la memoria)
- Modificadoras: modifican el parámetro implícito
- Consultoras: suministran información contenida en el objeto.
- Entrada/Salida: Escriben información contenida en el objeto.

La clase Estudiante

```
// Modificadoras  
  
void poner_nota (double n);  
// Pre: El estudiante no tiene nota y n es una nota válida.  
// Post: El estudiante pasa a tener nota n  
  
void cambiar_nota (double n);  
// Pre: El estudiante tiene un nota que es válida.  
// Post: El estudiante pasa a tener nota n  
  
// Consultoras  
  
int consultar_DNI() const;  
// Pre: Cierto  
// Post: Retorna el DNI del estudiante  
  
bool tiene_nota() const;  
// Pre: Cierto  
// Post: Retorna si el estudiante tiene nota  
  
double consultar_nota() const;  
// Pre: El estudiante tiene nota  
// Post: Retorna la nota del estudiante  
  
// Operación de la clase  
  
static double nota_maxima() const;  
// Pre: Cierto  
// Post: Retorna la nota máxima que puede tener un estudiante
```

Los métodos de clase

Se declaran al definir el método como **static**.

- Pertenecen a la clase y no a los objetos
- No tienen parámetros implícitos.

Por ejemplo:

```
cin >> nota;
if (nota >= 0 && nota <= Estudiant::nota_maxima())
    e.cambiar_nota(nota);
else
    cout << "la nota introducida no es valida" << endl;
```

Si nos queremos referir al P.I. podemos usar **this**:

```
class C {
private:
...
public
...
void copia(C x) const {
    if (this != x){
        ...
    }
}
```

¡¡Ojo!! No es buen estilo:

```
bool mismo_color(punto p) const {
    return this->color == p.color;
}
```

Diseño Orientado a Objetos

1. Identificamos las clases que juegan un papel en el programa

El propio enunciado es una buena fuente de información (abstracciones de datos). También el esquema de implementación que tengamos en la cabeza.

2. Especificamos dichas clases
3. Implementamos el programa principal en términos de las operaciones y objetos definidos por las clases.
4. Implementamos las clases especificadas.
5. Para implementar las clases puede ser necesario definir, especificar e implementar nuevas clases

Implementación de una Clase

1. Elegimos una representación para los objetos
2. Describimos su invariante
3. Implementamos sus operaciones
4. Si hace falta, utilizamos funciones auxiliares (privadas)

Ficheros de una Clase

Es conveniente separar en ficheros diferentes la *especificación* de la implementación de una clase:

- .hh : Contiene la especificación de la clase, aunque también las cabeceras y atributos de la parte privada.
- .cc : Contiene la implementación de las operaciones de la clase

Genericidad

Una clase genérica es una clase que tiene un *tipo parámetro*. En C++, a las clases genéricas se les llama templates:

```
template <class T> class Lista {  
private:  
    vector <T> v;  
    int sl;  
...}
```

Después podemos usar esta declaración para crear listas de diferentes tipos:

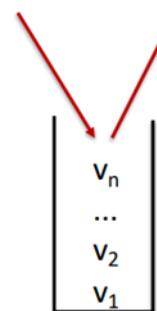
```
Lista <int> L1;  
Lista <string> L2;
```

Estructuras lineales

- Son estructuras de datos que contienen secuencias de valores
- Los accesos típicos que podemos tener son
 - Al primer elemento
 - Al último elemento
 - Al siguiente elemento
 - Al anterior elemento
- Las modificaciones típicas son inserciones o supresiones que pueden estar limitadas a los extremos
- Ejemplos típicos son pilas, colas, deque, listas, listas con prioridades, etc.
- En la STL de C++: stack, queue, deque, list, priority list, etc.
- Son *templates* (tienen un tipo como parámetro)
- Son contenedores (*containers*)

Pilas

- Solo se pueden acceder por un extremo.
- Tres operaciones básicas: *apilar*, *desapilar* y *cumbre*.
- El último apilado es al único que se puede acceder directamente y es el primero desapilado: *Last In – First Out*
- También se les llama *pushdown stores*



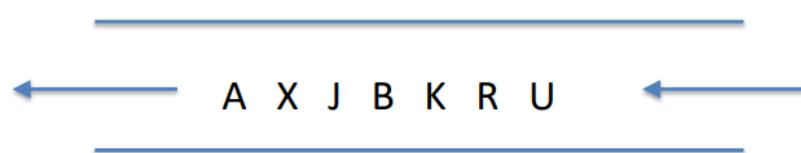
Estructuras lineales

- Operaciones básicas:
 - push
 - pop
 - top
 - empty

La clase queue

Tres operaciones básicas:

- Añadir un nuevo elemento (entrar, push)
- Eliminar el primer elemento que ha entrado (salir, pop)
- Ver quién es el primer elemento (primero, front)



Listas

Iteradores: declaración (instanciación)

1. Iteradores.
2. Especificación de la clase Lista.
3. Ejemplos de operaciones con listas
4. La operación *splice*.
5. Fusión de listas ordenadas

- Método `begin()`
- Método `end()`
- `list<Estudiant>::iterator it = l.begin();`
- `list<Estudiant>::iterator it2 = l.end();`
- Si una lista `l` está vacía, entonces `l.begin() = l.end()`

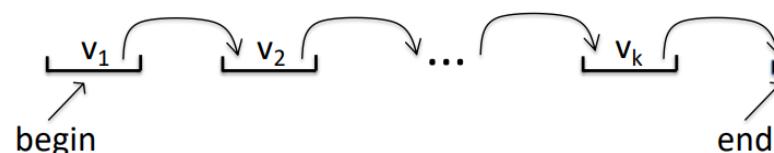
Listas

Las listas son estructuras lineales que permiten:

- Recorridos secuenciales de sus elementos.
- Insertar elementos en cualquier punto.
- Eliminar cualquier elemento
- Concatenar una lista a otra.
- Lo que nos permite hacer estas cosas son los *iteradores*

Contenedores e iteradores

- Un *iterador*, es un objeto que designa un elemento de un contenedor para desplazarnos por él.
- Un *contenedor* es una estructura de datos (un template) para almacenar objetos.
- Las listas son casos particulares de contenedores.



Esquema frecuente

Operaciones con iteradores

- `it1 = it2;`
- `it1 == it2, it1 != it2`
- `*it (si it != l.end())`
- `++it, --it (salvo si estamos en l.begin o en l.end())`
- **NO:** `it + 1, it1 + it2, it1 < it2, ...`

```
list<t> l;  
...  
list<t>::iterator it = l.begin();  
while (it != l.end() and not  
cond(*it)) {  
    ... acceder a *it ...  
    ++it; }
```

Iteradores constantes

Los iteradores constantes prohíben modificar el objeto referenciado por el iterador. Por ejemplo:

```
list<Estudiant>::const_iterator it1 = l.begin();
list<Estudiant>::iterator it2 = l.end();
```

Estaría prohibido: Suma de los elementos de una lista de enteros

```
*it1 = ...;
pero no:
it2 = it1;
*it2 = ...;
it1 = it2;
int suma(const list<int>& l) {
    int s = 0;
    for (list<int>::const_iterator it = l.begin();
          it != l.end(); ++it){
        // S es la suma de los elementos visitados de la lista
        s = s + *it;
    }
    return s;
}
```

Búsqueda en una lista de enteros

```
bool pertenece(const list<int>& l, int x) {
    list<int>::const_iterator it = l.begin();
    // it apunta a un elemento de l
    // x no está en las posiciones anteriores a it
    while ((it != l.end()) and (*it != x))
        ++it;
    return it != l.end();
}
```

Suma k a todos los elementos de una lista

```
void suma_k(list<int>& l, int k) {
    list<int>::iterator it = l.begin();
    // Se ha sumado k a todos los elementos anteriores a it
    while (it != l.end()) {
        *it += k;
        ++it;
    }
}
```

Inserción al por mayor

```
// Pre: true
/* Post: Se inserta en el PI toda la lista l delante del
   elemento apuntado por it y l queda vacía */
void splice(iterator it, list& l);
```

Concatenación de listas

```
// Pre: true
/* Post: Se inserta 12 al final de 11 y 12 queda vacía */
void concat(list& l1, list& l2){
    list<int>::iterator it = l1.end();
    l1.splice(it,l2);
}
```

Decir si una lista es capicua

```
void capicua(const list<int>& l) {
    list<int>::iterator it1 = l.begin();
    list<int>::iterator it2 = l.end();
    /* it1 e it2 apuntan a posiciones simétricas de la lista,
     * todas las parejas de elementos simétricos anteriores a it1
     * y posteriores a it2 son iguales */
    for (int i = 0; i < l.size()/2; ++i) {
        --it2;
        it = l.erase(it);
        if (*it1 != *it2) return false;
        ++ it1;
    }
    return true;
}
```

Inserción en una lista ordenada L1 de los elementos de otra lista ordenada L2

```
void inserc_ordenada(list<int>& L1, const list<int>& L2) {
    list<int>::iterator it1 = L1.begin();
    list<int>::iterator it2 = L2.begin();
    /* L1 y L2 están ordenadas. Todos los elementos de L1
     * anteriores a it1 son menores que el elemento apuntado por
     * it2. Y todos los elementos de L2 anteriores a it2 son
     * menores que el elemento apuntado por it1 */
    while (it1 != L1.end() and it2 != L2.end() ) {
        if (*it1 < *it2) ++it1;
        else {L1.insert(it1,*it2); ++it2;
    }
    while (it2 != L2.end() ) {
        L1.insert(it1,*it2);
        ++it2;
    }
}
```

Definiciones inductivas

La idea básica de una definición inductiva es que:

1. Para algunos casos básicos definimos la función directamente (los *más pequeños*)
2. Para el resto de los casos definimos la función en base a elementos *más pequeños*.
3. Tenemos *funciones de descomposición* que nos permiten obtener esos elementos más pequeños.

Factorial

$$n! = 1 * 2 * \dots * (n-1) * n$$

1. Caso base $n = 0$
2. El factorial de n lo podemos definir a partir del factorial de $n-1$
3. La función de descomposición es $n-1$

```
int factorial(int n){
    // Pre: n >= 0
    // Post: devuelve el factorial de n
    if (n == 0) return 1;
    else return n*factorial(n-1);
}
```

Suma de los elementos de una pila

Si $P = e_1 \ e_2 \ \dots \ e_n$

$\text{Suma}(P) = e_1 + e_2 + \dots + e_n$

1. Caso base: la pila está vacía

```
// Pre: true  
// Post: devuelve la suma de los valores de P  
  
int Suma(Stack <int> P) {  
    if (P.empty()) return 0;  
    else {  
        x = P.top();  
        P.pop();  
        return x+Suma(P);  
    }  
}
```

Terminación de una función recursiva

2. $\text{Suma}(P)$ se puede definir a partir de la suma del elemento que está en la cumbre de P y del resto de la pila
3. Las funciones de descomposición son `top` y `pop`

Para estar seguros de que una función recursiva termina:

- Hay que estar seguros de que las funciones de descomposición realmente nos dan elementos *más pequeños*,
- Los casos base son los más pequeños de todos.

La terminación se puede garantizar usando una función $|x|$ de medida o tamaño que cumple:

- $|x|$ nos devuelve un entero
- Si $|x| \leq 0$ entonces x es un caso base
- Si y es un parámetro de una llamada recursiva, entonces $|y| < |x|$.

Búsqueda en una pila

Si $P = e_1 \ e_2 \ \dots \ e_n$

1. Caso base: la pila está vacía

```
// Pre: true  
// Post: Nos dice si x está en P  
  
bool busq(Stack <int> P, int x) {  
    if (P.empty()) return false;  
    else {  
        if (P.top() == x) return true  
        else {  
            P.pop();  
            return busq(P, x);  
        }  
    }  
}
```

2. $\text{busq}(P, x)$ se puede definir a partir del elemento que está en la cumbre de P y del resto de la pila
3. Las funciones de descomposición son `top` y `pop`

Corrección parcial de un algoritmo recursivo

Hemos de demostrar:

- Si X es un caso inicial: directamente.
- En el caso general:
 - Hemos de comprobar que todo X' usado en las llamadas recursivas cumple Pre.
 - Comprobamos que los cálculos adicionales nos garantizan que el resultado de la función cumple Post.

```
// Exponenciación rápida  
  
// Pre: y >= 0  
// Post: Retorna xy  
  
int Potencia(int x, int y)  
    if (y == 0) return 1;  
    else if (y % 2 == 0)  
        return potencia(x*x, y/2);  
    else  
        return x*potencia(x, y-1);
```

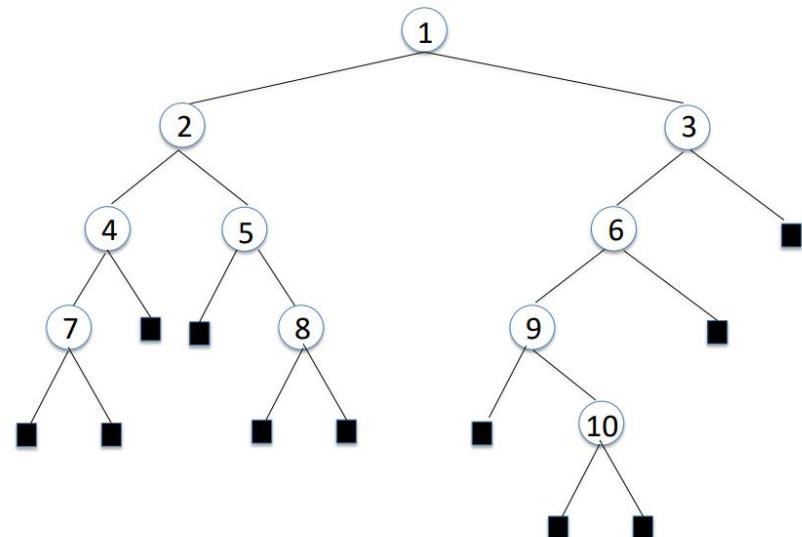
$$|x,y| = y$$

Especificación de la clase BinTree

```
template <class T> class BinTree {  
public:  
    // Constructoras  
    // Pre: true  
    // Post: crea un árbol vacío  
    BinTree();  
    // Pre: true  
    // Post: crea un árbol con x como raíz, y árboles vacíos  
    // como hijos  
    BinTree (const T& x);  
    // Pre: true  
    // Post: crea un árbol con x como raíz, left como hijo  
    // izquierdo y right como hijo derecho  
    BinTree (const T& x, const BinTree& left, const BinTree&  
right);
```

Árboles Binarios

- Un árbol binario es, o bien un árbol vacío, o bien es un nodo llamado raíz, que tiene dos sucesores (subárboles) que son árboles binarios
- Los dos sucesores de un nodo son su hijo izquierdo y su hijo derecho



```
// Consultoras // Pre: true  
// Post: Retorna true si el árbol y false en caso  
// contrario  
bool empty();  
  
// Pre: El parámetro implícito no está vacío  
// Post: retorna el hijo izqdo del parámetro implícito  
BinTree left();  
  
// Pre: El parámetro implícito no está vacío  
// Post: retorna el hijo dcho del parámetro implícito  
BinTree right();  
  
// Pre: El parámetro implícito no está vacío  
// Post: retorna la raíz del parámetro implícito  
T value();
```

Operaciones de BinTree

- BinTree no tiene modificadoras: la única manera de modificar un árbol binario es construir un arbol modificado y asignarlo al árbol original.
- Todos los métodos que hemos visto se ejecutan en tiempo constante.
- *No se hacen copias de los hijos.*

Tamaño de un árbol

```
/* Pre: true */  
/* Post: retorna el número de nodos del árbol t */  
int size(const BinTree <T>& t){  
    if (t.empty()) return 0;  
    else return 1 + size(t.left()) + size(t.right());  
}
```

Altura de un árbol

```
/* Pre: true */  
/* Post: retorna la altura del árbol t */  
int altura(const BinTree <T>& t){  
    if (t.empty()) return 0;  
    else return 1+max(altura(t.left()),altura(t.right()));  
}
```

Búsqueda en un árbol

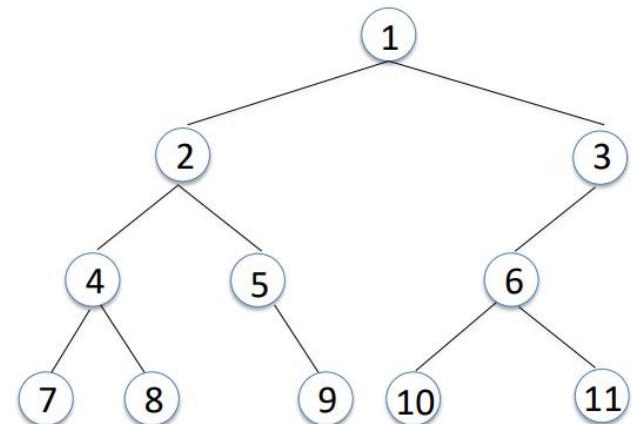
```
/* Pre: true */  
/* Post: nos dice si x está en t */  
bool busq(const BinTree <T>& t, int x){  
    if (t.empty()) return false;  
    else  
        return (t.value() == x) or busq(t.left(),x) or  
               busq(t.right(),x);  
}
```

Suma k a todos los valores de un árbol

```
/* Pre: true */  
/* Post: retorna un arbol t' con la misma forma que t,  
tal que el valor de cada nodo de t' es igual a k + el  
valor del nodo correspondiente de t */  
BinTree <int> sumak(const BinTree <int>& t, int k){  
    if (t.empty()) return t;  
    else return BinTree(t.value()+k,  
                        sumak(t.left(),k),  
                        sumak(t.right(),k));
```

Recorrido en preorden

1. Visitamos la raiz
2. Recorremos en preorden el hijo izquierdo
3. Recorremos en preorden el hijo derecho

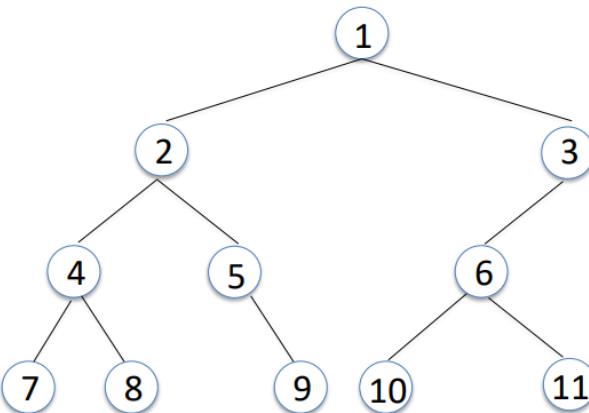


- Recorrido

1 2 4 7 8 5 9 3 6 10 11

Recorrido en postorden

1. Recorremos en postorden el hijo izquierdo
2. Recorremos en postorden el hijo derecho
3. Visitamos la raiz

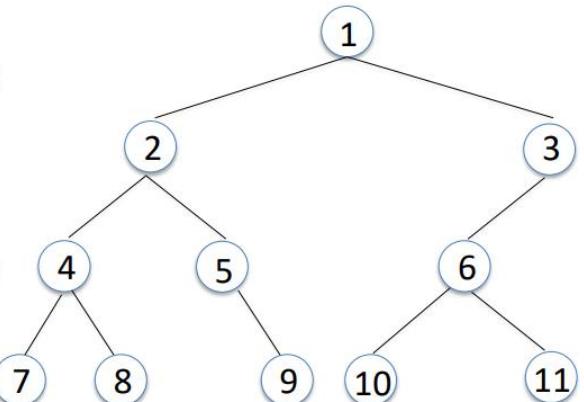


- Recorrido

7 8 4 9 5 2 10 11 6 3 1

Recorrido en inorder

1. Recorremos en inorder el hijo izquierdo
2. Visitamos la raiz
3. Recorremos en inorder el hijo derecho



- Recorrido

7 4 8 2 5 9 1 10 6 11 3

Recorrido en preorden

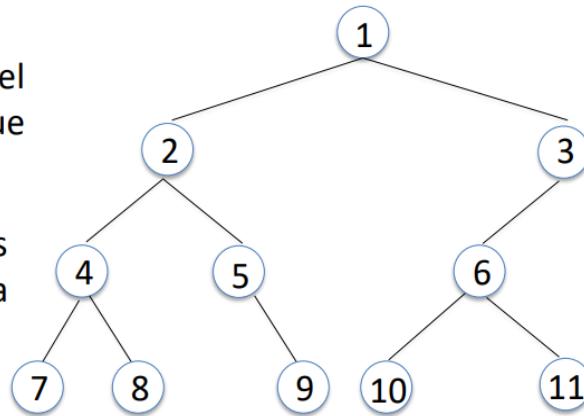
```
/* Pre: true */  
/* Post: El resultado es la lista en preorden de los  
elementos de t */  
list<T> preorden(const BinTree <T>& t,) {  
    list<T> L;  
    if (not t.empty()) {  
        L.insert(L.end(), t.value(),  
        L.splice(L.end(), preorden(t.left()),  
        L.splice(L.end(), preorden(t.right())));  
    }  
    return L;  
}
```

Recorrido en inorder

```
/* Pre: true */  
/* Post: El resultado es la lista en preorden de los  
elementos de t */  
list<T> preorden(const BinTree <T>& t,) {  
    list<T> L;  
    if (not t.empty()) {  
        L.splice(L.end(), preorden(t.left()),  
        L.insert(L.end(), t.value(),  
        L.splice(L.end(), preorden(t.right())));  
    }  
    return L;  
}
```

Recorrido por niveles

- Todos los nodos del nivel k son visitados antes que los del nivel $k+1$
- En cada nivel, los nodos se visitan de izquierda a derecha



- Recorrido

1 2 3 4 5 6 7 8 9 10 11

Recorrido por niveles

```
/* Pre: true */  
/* Post: El resultado es la lista de los elementos de t  
recorridos por niveles */  
list<int> niveles (const BinTree <int>& t,) {  
    list <int> L;  
    if (not t.empty()) {  
        queue <BinTree <int>> q; q.push(t);  
        while (not q.empty()) {  
            BinTree <int> aux = q.front(); q.pop();  
            L.insert(L.end(), aux.value());  
            if (not aux.left().empty()) q.push(aux.left());  
            if (not aux.right().empty()) q.push(aux.right());  
        }  
    }  
    return L;
```

Inmersión de una función en otra

Hacer una inmersión de una función f , quiere decir definir una función g , con más parámetros y que generaliza f .

Dos tipos de inmersiones:

- Inmersión con debilitamiento de la Post
- Inmersión con fortalecimiento de la Pre

Inmersión

Si directamente no se puede definir una función recursiva, se puede intentar añadir más parámetros

Si se repiten los cálculos, se pueden añadir más parámetros para recordarlos

Suma de un vector con inmersión

```
/* Pre: -1 <= n < v.size() */  
/* Post: devuelve la suma de los valores de  
v[0..n] */  
  
int i_Suma(const vector <int> &v, int n){  
    if (n < 0) return 0;  
    else return v[n]+i_Suma(v,n-1);  
}  
  
int Suma(const vector <int> &v){  
    return i_Suma(v,v.size()-1);  
}
```

Suma de un vector con otra inmersión

```
/* Pre: 0 <= n <= v.size() */
/* Post: devuelve la suma de los valores de
v[n..v.size()-1] */

int i_Suma(const vector <int> &v, int n){
    if (n == v.size()) return 0;
    else return v[n]+i_Suma(v,n+1);
}

int Suma(const vector <int> &v){
    return i_Suma(v,0);
}
```

Funciones de inmersión

Hay que

- Decidir qué inmersión se hará y qué parámetros añadir
- Especificar la función de inmersión (incluye la especificación de los nuevos parámetros y de los resultados)
- Definir la llamada inicial a la función de inmersión

Búsqueda en un vector ordenado

```
/* Pre: v.size()>0, v está ordenado crecientemente*/
/* Post: retorna una posición en que esté x, si x
está en v, si no retorna -1*/

int busq (const vector <int> &v, int x){
    return i_busq (v,x,0,v.size-1);
}
```

Búsqueda dicotómica

```
int i_busq(const vector <int> &v, int x, int
i, int j){
    int pos;
    if (j < i) pos = -1;
    else {
        mig = (i+j)/2;
        if (v[mig] = x) pos = mig;
        else {
            if (v[mig] < x)
                pos = i_busq(v,x, mig+1,j);
            else = i_busq(v,x, i,mig-1);
        }
    } return pos;
}
```

Mergesort

```
void mergesort (vector <int> &v, int i, int d){
    if (i<d){
        int m = (i+d)/2;
        mergesort(v,i,m);
        mergesort(v,m+1,d);
        fusiona(v,i,m,d);
    }
}
```

Inmersión por fortalecimiento de la Pre

- Cuando se debilita la postcondición, cada llamada recursiva hacen parte del trabajo.
- Cuando se fortalece la precondition, cada llamada recursiva recibe hecho parte del trabajo.

La primera suele ser más natural. La segunda tiene la ventaja de producir soluciones que son más fáciles de transformar en iterativas.

```

// Pre: true
/* Post: devuelve la suma de los valores de un
vector v */
int Suma(const vector <int> &v){
    return i_Suma(v,0,0);
}

/* Pre: 0 <= i < v.size(), s es la suma de
v[0..i-1] */
/* Post: devuelve la suma de los valores de v */

int i_Suma(const vector <int> &v, int i, int s){
    if (i == v.size()) return s;
    else return i_Suma(v,i+1,s+v[i]);
}

```

Recursividad lineal final (tail recursion)

- Un algoritmo recursivo es *lineal* si cada llamada genera una sola llamada recursiva
- Una función recursiva lineal es *final* si la última instrucción que se ejecuta es la llamada recursiva y el resultado obtenido es el resultado de esa llamada final

Recursividad lineal final (tail recursion)

- Las funciones recursivas lineales finales son fácilmente transformables en funciones iterativas.

Recursividad lineal final: el factorial

```

int factorial(int n){

    // Pre: n >= 0
    // Post: devuelve el factorial de n

    if (n == 0) return 1;
    else return n*factorial(n-1);
}

// Pre: n >= i >= 0, f = i!
// Post: devuelve el factorial de n

int i_factorial(int n, int i, int f){
    if (n == i) return f;
    else return i_factorial(n, i+1,f*(i+1));
}

int factorial(int n){
    return i_factorial(n,0,1);
}

```

Transformación a iterativo: factorial

```

int fact_iter(int n){
    int i = 0; int f = 1; //de la llamada inicial
    // Invariante: f = i! and i ≤ n
    while (i != n) {
        f = f*(i + 1); //de la llamada recursiva
        i = i+1;
    }
    return f;
}

```

Igualdad de pilas

```
// Pre: p1 = P1, p2 = P2
// Post: nos dice si P1 y P2 son iguales

bool iguales(Stack <int>& p1, Stack <int>& p2) {
    if (p1.empty() or p2.empty())
        return p1.empty() and p2.empty();
    else if (p1.top() != p2.top()) return false;
    else {
        p1.pop(); p2.pop();
        return iguales(p1,p2);
    }
}
```

Igualdad de pilas

```
// Pre: p1 = P1, p2 = P2
// Post: nos dice si P1 y P2 son iguales

bool ig_iter(Stack <int>& p1, Stack <int>& p2) {
    while (not p1.empty() and not p2.empty())
        if (p1.top() != p2.top()) return false;
        else {
            p1.pop();
            p2.pop();
        }
    return p1.empty() and p2.empty();
}
```

```
// Exponenciación (versión 2)

// Pre: x == A, y == B >= 0
// Post: El resultado p es AB

int p = 1;                                // Intercambio de dos variables enteras
while (y > 0) {                           // Pre: x = A, y = B.
    if (y % 2 == 0) {
        x = x * x;
        y = y / 2;
    } else {
        p = p * x;
        y = y - 1;
    }
}

// Post: x = B, y = A
```

$\boxed{\text{// } x = A+B, \text{ y } = B}$

$\boxed{\text{// } x = A+B, \text{ y } = A}$

Invariantes

Un invariante de un bucle es una asercción que siempre se cumple, independientemente del número de iteraciones

Típicamente, se coloca a la entrada del bucle

Describe la relación que existe entre las variables que intervienen en el bucle

Los invariantes se demuestran por inducción

Los invariantes son una buena documentación

$\boxed{\text{// Exponenciación}}$

$\boxed{\text{// Pre: x == A, y == B >= 0}}$

$\boxed{\text{// Inv: y>=0, A}^B \text{ = } x^y * \text{ p}}$

$\boxed{\text{// Post: El resultado p es A}^B}$

Verificación de bucles

Para razonar sobre un bucle hemos de:

1. Probar que la Pre del bucle implica el Invariante
2. Si se cumple Inv y se cumple C, después de ejecutar B, se vuelve a cumplir Invariante
3. Si se cumple Inv y no se cumple C, entonces se cumple la Post del bucle
4. El bucle termina

```
// Inv: y>=0, AB = xy * p
Suponemos que y = C, p = D

while (y > 0) {
    y = y - 1;
    p = p*x;
}

//Pero AB = xC * D = xC-1 * x * D = xy * p
```

Terminación de bucles

Para demostrar que un bucle termina hemos de definir una función F sobre las variables del bucle que cumpla (función de cota):

1. Si se cumple el invariante $F(\dots) \geq 0$
2. Si entramos en el bucle $F(\dots) > 0$
3. Si en un punto de la ejecución del bucle $F(\dots) = N$, después de ejecutar una iteración el $F(\dots) < N$.

La idea es que la función de cota nos diga cuántas iteraciones nos quedan como máximo.

Para comprobar que una función es correcta hay que:

1. Inventar un invariante I y una función de cota F.
2. Comprobar que después de la inicialización se cumple I.
3. Si se cumple I y la condición del bucle es cierta, después de ejecutar el cuerpo del bucle se vuelve a cumplir I.
4. I y la negación de la condición del bucle implican la post.
5. La función de cota decrece a cada iteración
6. Si entramos en el bucle, la función de cota es estrictamente positiva.

Número de elementos diferentes en una lista

```
// Pre: cert
/* Post: Si la lista está vacía retorna 0, en caso contrario, retorna el número de elementos diferentes que hay en L */
int ndif (List <int>& L) {
    int n = 0;
    List <int>::iterator it = L.begin();
    while (it != L.end()) {
        List <int>::iterator it1 = L.begin();
        while (it1 != it and *it1 != *it) ++it1;
        if (it1 == it) ++n;
        ++it;
    }
    return n;
}
```

Bucle interno

- **Función de cota.** $F = \text{número de elementos en el intervalo de } L \text{ entre } it1 \text{ e } it \text{ (excluido el apuntado por } it\text{)}$.
- F siempre es mayor o igual que 0.
- Cuando entramos en el bucle $F > 0$;
- A cada iteración, se reduce F , ya que incrementamos $it1$

- Invariante:

```
/* Inv: el valor de todos los valores que estan en
nodos anteriores a it1 son diferentes que *it, it1
apunta a un elemento de L en el intervalo
L.begin():it. */
```

```
/*Post del bucle interno: *it es la primera vez que
aparece si y solamente si it == it1 */
```

1. La primera vez, el invariante se cumple trivialmente.
2. Si se cumple Inv e **it != it1 and *it1 != *it**, claramente se vuelve a cumplir el invariante
3. Si no se cumple la condición del bucle, claramente se cumple la Post del bucle.

Bucle externo

- Función de cota. $F = \text{número de elementos en el intervalo de } L \text{ entre } it \text{ y } L.end()$.

- F siempre es positivo.
- Cuando entramos en el bucle $F > 0$;
- A cada iteración, se reduce F , ya que incrementamos it

- Invariante:

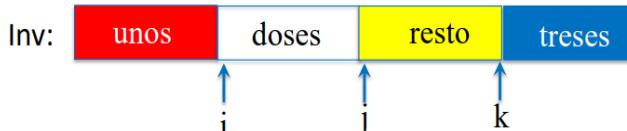
```
/* Inv: n = num de elementos diferentes en L hasta la
posición apuntada por it (excluida), it apunta a un
elemento de L */
//Post del bucle externo: n es el num de elementos
diferentes en L */
```

1. La primera vez, el invariante se cumple trivialmente.
2. Si se cumple Inv e **it != L.end()**, como después del bucle interno sabemos que el elemento apuntado por it es nuevo si y solo si **it1 == it**, si it apunta a un elemento nuevo se incremenra n en 1, y si no, se deja igual. con lo que al hacer $++it$, se vuelve a cumplir el invariante
3. Si no se cumple la condición del bucle y se cumple el invariante, claramente se cumple la Post del bucle.

Invariantes gráficos

```
void bandera_holandesa(vector <int> v){
    int i=0; int j=0; int k=v.size()-1;
    while(j <= k){
        if(v[j] == 1){
            swap(v[i],v[j]); ++j; ++i;
        }
        else if(v[j] == 2) ++j;
        else {swap(v[k],v[j]); --k;}
    }
}
```

$$F = k - j + 1$$



/* Pre: En la entrada tenemos una secuencia S de
caracteres acabada en un punto*/

```
int main() {
    char c;
    cin >> c;
    int cont = 0;
    // Inv: cont es el número de a's en la parte tratada
    //       de S y c contiene el primer carácter no
    //       tratado de S
    while (c != '.') {
        if (c == 'a') cont = cont + 1;
        cin >> c;
    }
    // cont es el número de a's en S
    cout << cont << endl;
}
/* Post: Escribe el número de a's que hay en S */
F(...) = longitud de S
```

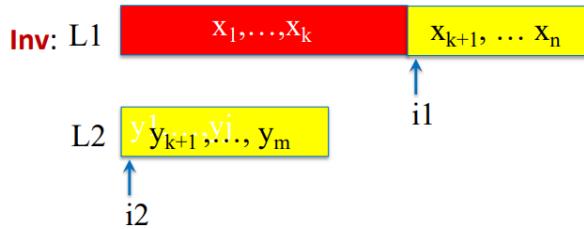
Diseño inductivo

Invertimos el proceso. A partir de una Pre y una Post:

1. Diseñamos un invariante adecuado
2. Definimos una inicialización para que se cumpla el invariante
3. Definimos una condición del bucle que, negada, garantice la postcondición del bucle
4. Diseñamos un cuerpo del bucle que mantenga el invariante.
5. Definimos una finalización para que se cumpla la Post

Inserción en una lista ordenada L1 de los elementos de otra lista ordenada L2

```
/* Pre: L1=[x1,...,xn], L2=[y1,...,ym] y las dos listas  
están ordenadas */  
/* Post: L1 contiene x1,...,xn,y1,...,ym y está ordenada, L2  
esta vacía */
```



L1 y L2 están ordenadas, todos los elementos de L2 son mayores que x_k y L1 contiene x₁,...,x_n, y₁,...,y_k

//condición del bucle:

```
(i1 != L1.end() and i2 != L2.end())
```

Es decir, a la salida del bucle se cumpliría:

```
/* L1 y L2 están ordenadas, todos los elementos de L2 son  
mayores que xk, L1 contiene x1,...,xn, y1,...,yk e (i1 == L1.end()  
o i2 == L2.end()) */
```

```
/* L1 y L2 están ordenadas, todos los elementos de L2 son  
mayores que xk, L1 contiene x1,...,xn, y1,...,yk e (i1 == L1.end()  
o i2 == L2.end()) */
```

```
L1.splice(i1,L2);
```

```
/* L1 está ordenada, L2 está vacía, todos los elementos de  
L2, mayores que xk y L1 contiene x1,...,xn, y1,...,ym e */
```

Es decir, la función sería:

```
void fusion(list<int> L1, list<int> L2){  
    list<int>::iterator i1 = L1.begin();  
    list<int>::iterator i2 = L2.begin();  
    while (i1 != L1.end() and i2 != L2.end() ) {  
        if (*i1 < *i2) ++i1;  
        else {  
            L1.insert(i1,*i2); i2 = L2.erase(i2);  
        }  
    }  
    L1.splice(i1,L2);  
}  
F(...) = dist. de i1 a L1.end() + dist. de i2 a L2.end()
```

Concepto de coste

Coste de un algoritmo:

- En tiempo: *número de operaciones*
- En memoria: *número de posiciones ocupadas*

```
// Pre: A es un vector no vacío  
// Post: devuelve i, tal que A[i] == x, si x está en A  
//        devuelve -1 si x no está en A  
int busq(int x, const vector<int>& A) {  
    for (int i = 0; i < A.size(); ++i)  
        if (A[i] == x) return i;  
    return -1;  
}
```

Concepto de coste (en tiempo)

Ejemplo: Búsqueda secuencial en un vector:

- Si el vector tiene 5 posiciones y el elemento buscado está en la segunda posición, el coste en tiempo es ...
- Si el vector tiene 5 posiciones y el elemento buscado está en la segunda posición, el coste en espacio es ...

Coste de un algoritmo

Coste de un algoritmo, suponiendo que el tamaño de los datos es N :

- Constante: c *Coste constante*
- Lineal: $c_1 * N + c_2$ *Coste del orden de N*
- Polinómico: $c_0 + c_1 * N + \dots + c_k * N^k$ *Coste del orden de N^k*
- Logarítmico: $c_1 * \log(c * N + c_2) + c_3$ *Coste del orden de $\log(N)$*
- Exponencial: $c^N + c'$ *Coste del orden de 2^N*

Concepto de coste (en tiempo)

Ejemplo: Búsqueda secuencial en un vector:

- Si el elemento buscado está en la primera posición: coste constante (Mejor caso)
- Si el elemento buscado está en la última posición, o no está en el vector: coste lineal (Peor caso)
- En término medio: coste lineal ($n/2$).

Concepto de eficiencia

Ejemplos: algoritmos que operan con vectores de tamaño n

- Búsqueda secuencial: n
- Búsqueda dicotómica: $\log_2 n$
- Ordenación por selección, inserción o burbuja: n^2
- Mergesort: $n * \log_2 n$
- Quicksort: ?

Eliminación de cálculos repetidos

Algoritmos iterativos:

- Añadir variables locales para *recordar* cálculos para la siguiente iteración
- No aparecen ni en la Pre ni en la Post
- Aparecen en el invariante

```
// Pre: v.size() > k >= 0
// Post: retorna true si hay un i, k <= i < v.size()
// tal que v[i] = v[i-k]+...+v[i-1]
```

```
bool suma_k_anteriores(const vector<int> &v, int k){
    int i = k;

    // Inv: no hay j < i tal que v[j] = v[j-k]+...+v[j-1]
    while (i < v.size()){
        if (v[i] == suma(v, i-k, i-1)) return true;
        ++i;
    }
    return false;
}
```

Coste total: $(n-k)*k$

```

// Pre: v.size() > k >= 0
// Post: retorna true si hay un i, k <= i < v.size()
//       tal que v[i] = v[i-k]+...+v[i-1]

bool suma_k_anteriores(const vector<int> &v, int k){
    int sum = 0; i = k;
    for (int j = 0; j < k; ++j) sum = sum+v[j];
    // Inv: no hay j < i tal que v[j] = v[j-k]+...+v[j-1],
    //       sum = v[i-k]+...+v[i-1],
    while (i < v.size()){
        if (v[i]==sum) return true;
        sum = sum-v[i-k]+v[i];
        ++i;
    }
    return false;
}

```

Coste total proporcional a n, independientemente de k.

Elementos bisagra

Un elemento $v[i]$ de un vector es un elemento bisagra si es igual a la diferencia entre la suma de los elementos posteriores y la suma de los anteriores:

$$v[i] = \text{suma}(v, i+1, v.size()-1) - \text{suma}(v, 0, i-1)$$

[1,3,**11**,6,5,4]

[**2**,1,1]

[1,2,**1**,0,4]

```

// Pre: true
// Post: retorna el número de elementos frontera que
//       hay en v

int bisagras(const vector<int> &v){
    int i = 0; int n = 0;
    // Inv: 0 <= i <= v.size(), n es el nº de elementos
    //       frontera de v[0..i-1]
    while (i < v.size()){
        if (v[i] == suma(v, i+1, v.size()-1)-suma(v, 0, i-1)) ++n;
        ++i;
    }
    return n;
}

```

Coste total: n^2

```

// Pre: true
// Post: retorna el número de elementos frontera que
//       hay en v

int bisagras(const vector<int> &v){
    int sumaant = 0; int sumapost = suma(v, 1, v.size()-1);
    int i = 0; int n = 0;
    // Inv: 0 <= i <= v.size(), n es el nº de elementos
    //       frontera de v[0..i-1], sumaant = suma(v, 0, i-1),
    //       sumapost = suma(v, i+1, v.size()-1),
    while (i < v.size()){
        if (v[i] == sumapost-sumaant) ++n;
        sumaant = sumaant+v[i];
        if (i < v.size()-1) sumapost = sumapost-v[i+1];
        ++i;
    }
    return n;
}

```

Coste proporcional a n