

Desarrollo de Iteradores

16 de mayo de 2018

diseño

Especificación → Diseño

¿Por qué especificamos?

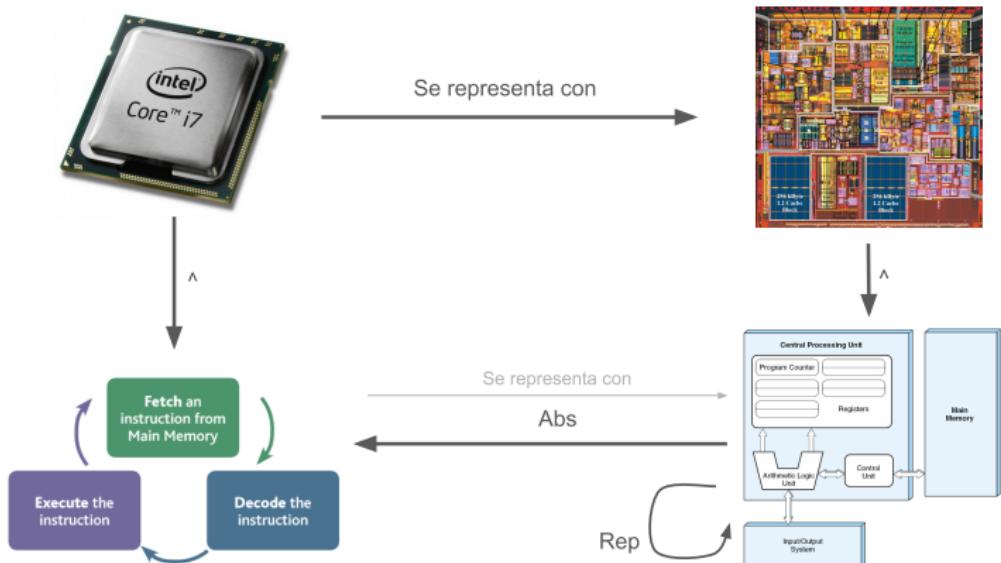
Para decir que significa cruzar el camino.

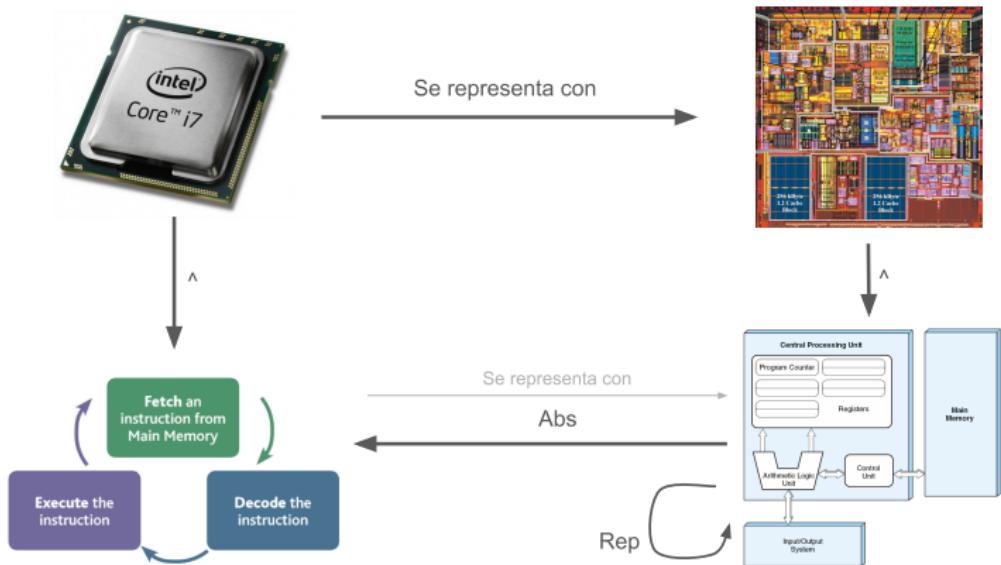
¿Por qué diseñamos?

Para saber qué camino vamos a tomar y cuánto tiempo nos va a tomar cruzar el camino.

¿Por qué armamos un módulo de diseño?

Para que otros puedan decir “para llegar al otro lado” y sigan con otra cosa.









¿Qué es un módulo de diseño?

Módulo

Requerimientos

Parámetros formales

Género

Se explica con

Interfaz

[...]

Estructura de representación

[...]

Algoritmos

[...]

4. Módulo Lista Enlazada(α)

El módulo Lista Enlazada provee una secuencia que permite la inserción, del primer y último elemento. En cambio, el acceso a un elemento aleatorio este módulo implementa lo que se conoce como una lista doblemente enlazad

En cuanto al recorrido de los elementos, se provee un iterador bidireccionales. De esta forma, se pueden aplicar filtros recorriendo una única vez la lista tanto apuntando al inicio, en cuyo caso el siguiente del iterador es el primer fin, en cuyo caso el anterior del iterador es el último elemento de la lista. En cada de la lista en forma eficiente.

Para describir la complejidad de las operaciones, vamos a llamar $copy(a)$ (a $copy$ es una función de α en \mathbb{N}).¹

Interfaz

parámetros formales

géneros α
función COPIAR(in $a : \alpha$) $\rightarrow res : \alpha$

Pre $\equiv \{\text{true}\}$

Post $\equiv \{res =_{\text{obs}} a\}$

Complejidad: $\Theta(copy(a))$

Descripción: función de copia de α 's

se explica con: SECUENCIA(α), ITERADOR BIDIRECCIONAL(α).

géneros: lista(α), itLista(α).

Apunte de módulos básicos

¿Qué es un módulo de diseño?

Módulo

Requerimientos

Parámetros formales

Género

Se explica con

Interfaz

[...]

Estructura de representación

[...]

Algoritmos

[...]

Operaciones básicas de lista

VACÍA() $\rightarrow res : \text{lista}(\alpha)$

Pre $\equiv \{\text{true}\}$

Post $\equiv \{res =_{\text{obs}} <>\}$

Complejidad: $\Theta(1)$

Descripción: genera una lista vacía.

AGREGARADELANTE(**in/out** $l : \text{lista}(\alpha)$, **in** $a : \alpha$) $\rightarrow res : \text{itLista}(\alpha)$

Pre $\equiv \{l =_{\text{obs}} l_0\}$

Post $\equiv \{l =_{\text{obs}} a \bullet l_0 \wedge res = \text{CrearItBi}(<>, l) \wedge \text{alias}(\text{SecuSuby}(res) = l)\}$

Complejidad: $\Theta(\text{copy}(a))$

Descripción: agrega el elemento a como primer elemento de la lista. Retorna
Siguiente devuelva a .

Aliasing: el elemento a agrega por copia. El iterador se invalida si y sólo si
iterador sin utilizar la función ELIMINARSIGUIENTE.

AGREGARATRAS(**in/out** $l : \text{lista}(\alpha)$, **in** $a : \alpha$) $\rightarrow res : \text{itLista}(\alpha)$

Pre $\equiv \{l =_{\text{obs}} l_0\}$

Post $\equiv \{l =_{\text{obs}} l_0 \circ a \wedge res = \text{CrearItBi}(l_0, a) \wedge \text{alias}(\text{SecuSuby}(res) = l)\}$

Complejidad: $\Theta(\text{copy}(a))$

Descripción: agrega el elemento a como último elemento de la lista. Retorna
Siguiente devuelva a .

¿Qué es un módulo de diseño?

Módulo

Requerimientos

Parámetros formales

Género

Se explica con

Interfaz

[...]

Estructura de representación

[...]

Algoritmos

[...]

Representación

Representación de la lista

El objetivo de este módulo es implementar una lista doblemente enlazada con punteros. Para simplificar un poco el manejo de la estructura, vamos a reemplazarla por una lista circular. La lista circular apunta al primero y el anterior del primero apunta al último. La estructura de representación y su función de abstracción son las siguientes.

lista(α) se representa con lst

donde lst es tupla(primer: puntero(nodo), longitud: nat)

donde nodo es tupla(dato: α , anterior: puntero(nodo), siguiente: puntero(nodo))

Rep : lst \rightarrow bool

Rep(l) \equiv true \iff ($l.\text{primer} = \text{NULL}$) $=$ ($l.\text{longitud} = 0$) \wedge_l ($l.\text{longitud} \neq 0 \Rightarrow_l$

Nodo(l , $l.\text{longitud}$) $= l.\text{primer}$ \wedge

($\forall i: \text{nat}$) ($\text{Nodo}(l, i) \rightarrow \text{siguiente} = \text{Nodo}(l, i + 1) \rightarrow \text{anterior}$) \wedge

($\forall i: \text{nat}$) ($1 \leq i < l.\text{longitud} \Rightarrow \text{Nodo}(l, i) \neq l.\text{primer}$)

Nodo : lst $l \times \text{nat} \rightarrow$ puntero(nodo)

Nodo(l, i) \equiv if $i = 0$ then $l.\text{primer}$ else Nodo(FinLst(l), $i - 1$) fi

FinLst : lst \rightarrow lst

FinLst(l) \equiv Lst($l.\text{primer} \rightarrow \text{siguiente}$, $l.\text{longitud} - \min\{l.\text{longitud}, 1\}$)

Lst : puntero(nodo) \times nat \rightarrow lst

Lst(p, n) \equiv $\langle p, n \rangle$

Abs : lst $l \rightarrow \text{secu}(\alpha)$

Abs(l) \equiv if $l.\text{longitud} = 0$ then $\langle \rangle$ else $l.\text{primer} \rightarrow \text{dato} \bullet \text{Abs}(\text{FinLst}(l))$ fi

Diseñando Iteradores

Iteradores

Aparecen durante el diseño:

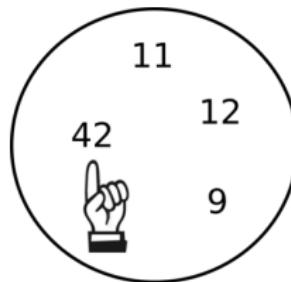
- ▶ Eficiencia (recorrido y modificación)
- ▶ Interfaz común de contenedores

Su funcionamiento depende de la estructura interna del contenedor...

→ Módulo interno (Diseño) o Member class (C++)

Repasemos

- ▶ colección + dedo



- ▶ inicialización (`iterator Coleccion::begin()`)
- ▶ avanzar (`iterator Coleccion::iterator::operator++()`)
- ▶ obtener elemento (`T Coleccion::iterator::operator*()`)
- ▶ saber si terminé
(`T Coleccion::iterator::operator==(const iterator& o)`
+ `iterator Coleccion::end()`)

Ejemplo: Vector

$\text{vector}(\alpha)$ se representa con estr

donde estr es tupla(

valores: arreglo(α)

tam: nat

capacidad: nat

)

Rep : estr \rightarrow bool

Rep(e) \equiv true \iff

$\text{tam} \leq \text{capacidad}$ \wedge

para todo $0 \leq i < \text{tam} \Rightarrow \text{definido?}(i, \text{valores})$ \wedge

para todo $\text{tam} \leq i < \text{capacidad} \Rightarrow \neg \text{definido}(i, \text{valores})$

Abs : estr e \rightarrow secu(α)

$\{\text{Rep}(e)\}$

Abs(e) =_{obs} s: secu(α) |

$\text{long}(s) = \text{tam}$ \wedge

para todo $0 \leq i < \text{tam} \Rightarrow \text{iesimo}(i, s) = \text{valores}[i]$

Ejemplo: Lista

lista(α) se representa con estr

donde estr es tupla($prim: \text{puntero}(\text{nodo})$)

donde nodo es tupla($valor: \alpha, sig: \text{puntero}(\text{nodo})$)

Rep : estr \longrightarrow bool

Rep(e) \equiv true \iff siguiendo los punteros a siguiente desde prim no veo ningún nodo dos veces

Abs : estr e \longrightarrow secu(α) $\{\text{Rep}(e)\}$

Abs(e) =_{obs} s: secu(α) |

Si prim es NULL, la secuencia es vacía. Sino el valor del nodo apuntado por prim es prim(s) y la lista conformada por sig del nodo apuntado por prim conforma una lista que es fin(s).

Ejemplo: ABB

$\text{conj}(\alpha)$ se representa con estr

donde estr es tupla(*raiz*: puntero(nodo))

donde nodo es tupla(*valor*: α , *izq*: puntero(nodo), *der*: puntero(nodo))

Rep : estr \longrightarrow bool

Rep(*e*) \equiv true \iff

Raiz es NULL o

Todos los elementos en el subarbol izq son menores que raiz.valor,
todos los elementos en el subarbol der son mayores que raiz.valor,
siguiendo los punteros no tengo un ciclo y el rep se cumple para
ambos subárboles

Abs : estr *e* \longrightarrow conj(*alpha*) $\{\text{Rep}(e)\}$

Abs(*e*) =_{obs} c: conj(*alpha*) |

Si raiz es NULL, el conjunto está vacío. Sino el conjunto posee los elementos
en algún nodo alcanzable desde raiz y ningún otro.

Ejemplo: ABB

TREE-SUCCESSOR(x)

- 1 **if** $right[x] \neq \text{NIL}$
- 2 **then return** TREE-MINIMUM($right[x]$)
- 3 $y \leftarrow p[x]$
- 4 **while** $y \neq \text{NIL}$ and $x = right[y]$
- 5 **do** $x \leftarrow y$
- 6 $y \leftarrow p[y]$
- 7 **return** y

Ejemplo: ABB

iterador **se representa con** iter

donde iter es tupla(*actual*: puntero(nodo)
padres: pila(puntero(nodo)))

Rep : iter → bool

Rep(*e*) ≡ true ⇔

Para todo puntero(nodo) en *padres*, su anterior en la pila
es hijo de este y *actual* es hijo del *tope*(*padres*).

```
ConstructorIterador(Nodo* inicio)
```

```
padres ← vacio()
```

```
minimo_y_apilar(inicio)
```

```
ConstructorIterador()
```

```
padres ← vacio()
```

```
actual ← NULL
```

Ejemplo: ABB

```
minimo_y_apilar(Nodo* inicio)
if inicio = NULL then
    actual ← NULL
    return
end if
Nodo* n ← inicio
while n.izq ≠ NULL do
    padres.apilar(n)
    n ← n.izq
end while
actual ← n
```

Ejemplo: ABB

```
iterator iterator::operator++()
if actual->der ≠ NULL then
    padres.apilar(actual)
    minimo_y_apilar(actual->der)
else
    while ¬padres.vacio() ∧ padres.tope().der = actual do
        actual ← padres.tope()
        padres.desapilar()
    end while
    if padres.vacio() then
        actual ← NULL
    end if
end if
```

En C++

Member classes

```
template<typename T>
class Vector {
public:
    typedef T value_type;

    class iterator; //Forward declaration
    class const_iterator;

    /* [...] */

    iterator begin();
    iterator end();

    const_iterator begin() const;
    const_iterator end() const;

private:
    T* valores;
    int tam;
    int capacidad;
};
```

En el iterador

```
template<typename T>
class Vector {
public:
/* [...] */
class iterator {
public:
    typedef T value_type;

    iterator(const iterator&);
    iterator& operator=(const iterator&);
    bool operator==(const iterator &) const;
    bool operator!=(const iterator &) const;

    iterator& operator++();
    value_type& operator*() const;

    friend class Lista;

private:
    iterator(T*, int pos);

    T* _valores;
    int _pos;
};

private:
/* [...] */
```

```
template<typename T>
Vector<T>::iterator::iterator(T* valores, int pos) :
    _valores(valores), _pos(pos) {}

template<typename T>
Vector<T>::iterator::iterator(const Vector<T>::iterator otro) :
    _valores(otro._valores), _pos(otro._pos) {}

template<typename T>
Vector<T>::iterator Vector<T>::begin() {
    return iterator(this->valores, 0);
}

template<typename T>
Vector<T>::iterator Vector<T>::end() {
    return iterator(this->valores, this->tam);
}
```

```
template<typename T>
Vector<T>::iterator& Vector<T>::iterator::operator++() {
    _pos++;
}

template<typename T>
T& Vector<T>::iterator::operator*() {
    _valores[_pos];
}

template<typename T>
bool Vector<T>::iterator::operator==(const Vector<T>::iterator & otro) const{
    return _pos == otro._pos;
}
```

Aclaración iteradores

En general, se asegura que el iterador tiene sentido mientras no se modifique la estructura que itera. De modificarse, los iteradores pueden invalidarse.