

Desarrollo de Iteradores

16 de mayo de 2018

Especificación → Diseño

¿Por qué especificamos?

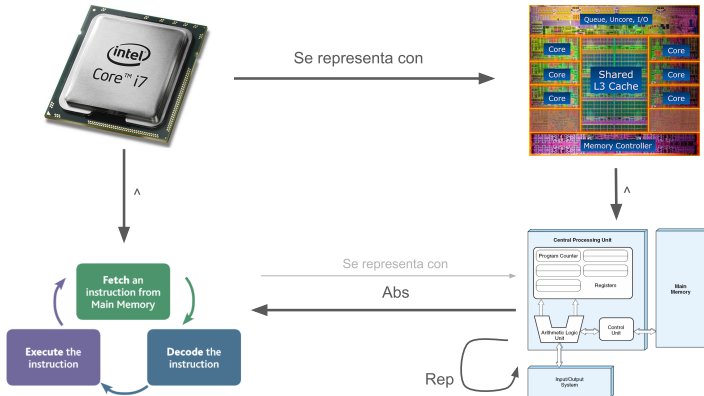
Para decir que significa cruzar el camino.

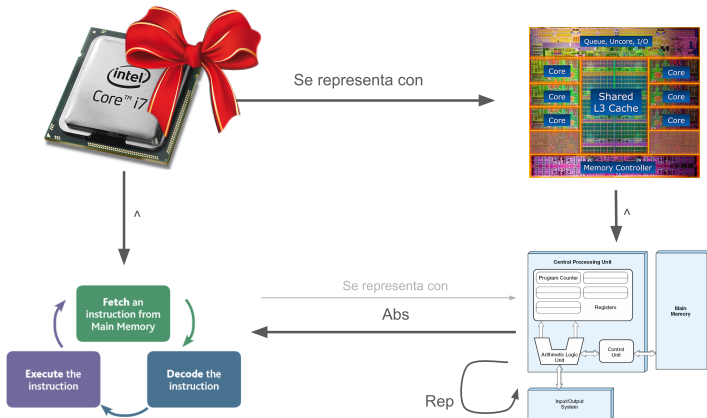
¿Por qué diseñamos?

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

¿Por qué armamos un módulo de diseño?

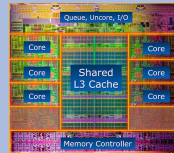
Para que otros puedan decir “para llegar al otro lado” y sigan con otra cosa.





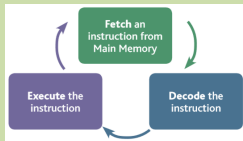


Se representa con



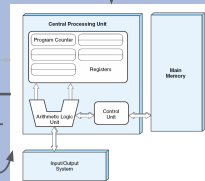
Más abstracción

Menos abstracción



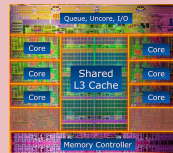
Se representa con

Abs

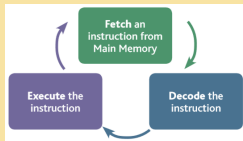




Se representa con



Mundo Implementación

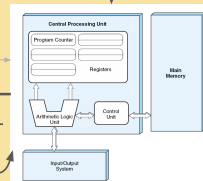


Mundo Tads

Se representa con

Abs

Rep



¿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 enlazada.

En cuanto al recorrido de los elementos, se provee un iterador bidireccional. 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 elemento, en cuyo caso el anterior del iterador es el último elemento de la lista. En caso de la lista en forma eficiente.

Para describir la complejidad de las operaciones, vamos a llamar $copy(a)$ a la copia de a en forma eficiente. ¹

Interfaz

parámetros formales

géneros α

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

$Pre \equiv \{true\}$

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

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() $\rightarrow res : lista(\alpha)$

Pre $\equiv \{true\}$

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

Complejidad: $\Theta(1)$

Descripción: genera una lista vacía.

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

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

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

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

Descripción: agrega el elemento a como primer elemento de la lista. Retorna el iterador de la lista. Si la lista es vacía, devuelve el iterador vacío.

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

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

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

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

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

Descripción: agrega el elemento a como último elemento de la lista. Retorna el iterador de la lista. Si la lista es vacía, devuelve el iterador vacío.

¿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. El último 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**(*primero*: 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*.primero = NULL) = (*l*.longitud = 0) \wedge (*l*.longitud \neq 0 \Rightarrow
 Nodo(*l*, *l*.longitud) = *l*.primero \wedge
 ($\forall i$: nat)(Nodo(*l*, *i*) \rightarrow siguiente = Nodo(*l*, *i* + 1) \rightarrow anterior) \wedge
 ($\forall i$: nat)(1 $\leq i < l$.longitud \Rightarrow Nodo(*l*, *i*) $\neq l$.primero)

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

Nodo(*l*, *i*) \equiv if *i* = 0 then *l*.primero else **Nodo**(FinLst(*l*), *i* - 1) fi

FinLst : lst \rightarrow lst

FinLst(*l*) \equiv Lst(*l*.primero \rightarrow siguiente, *l*.longitud - mín{*l*.longitud, 1})

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

Lst(*p*, *n*) \equiv (*p*, *n*)

Abs : lst \rightarrow secu(α)

Abs(*l*) \equiv if *l*.longitud = 0 then $\langle \rangle$ else *l*.primero \rightarrow dato • **Abs**(FinLst(*l*)) fi

Diseñando Iteradores

Iteradores

Aparecen durante el diseño:

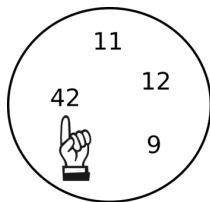
- ▶ Eficiencia (recorrido y modificación)
- ▶ Interfaz común de contenedores

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 $\text{tupla}(\begin{array}{l} \text{valores: arreglo}(\alpha) \\ \text{tam: nat} \\ \text{capacidad: nat} \end{array})$

$\text{Rep} : \text{estr} \longrightarrow \text{bool}$

$\text{Rep}(e) \equiv \text{true} \iff$
 $\text{tam} \leq \text{capacidad} \wedge$
 $\text{para todo } i : \text{nat}, 0 \leq i < \text{tam} \Rightarrow \text{definido?}(i, \text{valores}) \wedge$
 $\text{para todo } i : \text{nat}, \text{tam} \leq i < \text{capacidad} \Rightarrow \neg \text{definido}(i, \text{valores})$

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

$\text{Abs}(e) =_{\text{obs}} s : \text{secu}(\alpha) \mid$
 $\text{long}(s) = \text{tam} \wedge$
 $\text{para todo } 0 \leq i < \text{tam} \Rightarrow \text{iesimo}(i, s) = \text{valores}[i]$

Ejemplo: Lista

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

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

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

$\text{Rep} : \text{estr} \longrightarrow \text{bool}$

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

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

$\text{Abs}(e) =_{\text{obs}} s : \text{secu}(\alpha) \mid$

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

Ejemplo: ABB

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

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

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

$\text{Rep} : \text{estr} \longrightarrow \text{bool}$

$\text{Rep}(e) \equiv \text{true} \iff$

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

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

$\text{Abs}(e) =_{\text{obs}} c : \text{conj}(\alpha) \mid$

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 \longrightarrow bool

Rep(*e*) \equiv true \iff

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 \leftarrow vacio()

minimo_y_apilar(inicio)

ConstructorIterador()

padres \leftarrow vacio()

actual \leftarrow NULL

Ejemplo: ABB

```
minimo_y_apilar(Nodo* inicio)
  if inicio = NULL then
    actual  $\leftarrow$  NULL
    return
  end if
  Nodo* n  $\leftarrow$  inicio
  while n.izq  $\neq$  NULL do
    padres.apilar(n)
    n  $\leftarrow$  n.izq
  end while
  actual  $\leftarrow$  n
```

Ejemplo: ABB

```
iterator iterator::operator++()  
    if actual→der ≠ NULL then  
        padres.apilar(actual)  
        minimo_y_apilar(actual→der)  
    else  
        while  $\neg \text{padres.vacio}() \wedge \text{padres.tope().der} = \text{actual}$  do  
            actual ← padres.tope()  
            padres.desapilar()  
        end while  
        if  $\neg \text{padres.vacio}()$  then  
            actual ← padre.tope()  
            padres.desapilar()  
        else  
            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 vector;

    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.