Desarrollo de Iteradores

Algoritmos y Estructuras de Datos II

Iteradores

Aparecen durante el diseño:

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

Iteradores

Aparecen durante el diseño:

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

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

Iteradores

Aparecen durante el diseño:

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

Su funcionamiento depende de la estructura interna del contenedor...
$\rightarrow$ Módulo interno (Diseño) o Member class (C++)

¿Por qué los necesitamos?

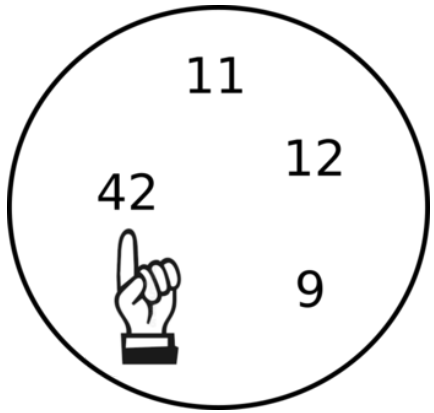
- Dan una interfaz común para recorrer la estructura ocultando los detalles de la estructura que iteran.
- Podemos usar iteradores como "punteros seguros" a la estructura interna sin exponerla.
- No destruye la estructura que recorre, por lo que evita hacer una copia innecesaria de la estructura antes de recorrerla.
- Estandard de C++ (si vamos a modularizar contenedores vamos a hacerlo bien...).
- Operaciones eficientes (insertar en un árbol en O(1) amortizado [1])

[1] `https://en.cppreference.com/w/cpp/container/set/insert`

> 3-4) Amortized constant if the insertion happens in the position just *before* the hint, logarithmic in the size of the container otherwise. (since C++11)

Iteradores - Repaso

- 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 interator& o)`
  `+ iterator Coleccion::end()`)

El iterador tiene entidad propia. Recordemos como se usan:

```
vector<int> vec = {1, 3, 4, 8};
vector<int>::iterator it = vec.begin();
++it;
++it;
cout << *it << endl; // 4
vector<int>::iterator it2 = it;
++it2;
cout << *it2 << endl; // 8
cout << it2 == vec.end() << endl; // False
++it2;
cout << it2 == vec.end() << endl; // True
cout << *it << endl; // 4
```

Se crea desde vector pero luego se opera de forma independiente. No obstante, el iterador debe tener alguna referencia al vector.

Ejemplo: Vector

`vector`(α) **se representa con** `estr`

donde `estr` es `tupla`(
valores: `arreglo`(α)
tam: `nat`
capacidad: `nat`
)

`iter` **se representa con** `estr_iter`

donde `estr_iter` es `tupla`(???
)

`vector` `::begin`	`vector::iter` `::op++`	`vector::iter` `::op*`	`vector::end`

Ejemplo: Vector

`vector`(α) **se representa con** `estr`

donde `estr` es `tupla`(
valores: `arreglo`(α)
tam: `nat`
capacidad: `nat`
)

Rep : estr $\longrightarrow$ bool
Rep(e) $\equiv$ true $\Longleftrightarrow$
tam $\leq$ capacidad $\wedge$
para todo i : *nat*, $0 \leq i <$ *tam* $\Rightarrow$ definido?(i, valores) $\wedge$
para todo i : *nat*, *tam* $\leq i <$ *capacidad* $\Rightarrow \neg$ definido(i, valores)

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

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

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

```cpp
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;
}
```

Más ejemplos

Ejemplo: Lista

`lista`(α) **se representa con** `estr`

donde `estr` es `tupla`(*prim*: `puntero(nodo)`)
donde `nodo` es `tupla`(*valor*: α, *sig*: `puntero(nodo)`)

Rep : estr $\longrightarrow$ bool
Rep(e) $\equiv$ true $\Longleftrightarrow$ siguiendo los punteros a siguiente desde prim no veo ningún nodo dos veces

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

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

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

`conj`(α) **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 $\Longleftrightarrow$
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*) $\{$Rep(e)$\}$

Abs(e) $=_{\text{obs}}$ 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] ≠ NIL
2     then return TREE-MINIMUM(right[x])
3  y ← p[x]
4  while y ≠ NIL and x = right[y]
5        do x ← y
6           y ← p[y]
7  return y
```

Necesitamos $p[x]$, pero nuestra estructura no tiene puntero al padre...

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 $\Longleftrightarrow$
Para todo puntero(nodo) en *padres*, su siguiente en la pila su hijo y *actual* es hijo del *tope(padres)*.

```
iterador(Nodo* inicio)
    padres ← vacio()
    minimo_y_apilar(inicio)

iterador()
    padres ← vacio()
    actual ← NULL
```

Ejemplo: ABB

```
iterador::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

```
TREE-SUCCESSOR(x)
1  if right[x] ≠ NIL
2     then return TREE-MINIMUM(right[x])
3  y ← p[x]
4  while y ≠ NIL and x = right[y]
5        do x ← y
6           y ← p[y]
7  return y
```

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 ← padre.tope()
          padres.desapilar()
      else
          actual ← NULL
      end if
  end if
```

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.