Clase de Iteradores en C++

Algoritmos y estructuras de datos II

12 de octubre de 2016

Iteradores

- ¿Qué es un iterador?
  - Es una estructura que permite recorrer otra de manera eficiente
- ¿Para qué sirven?
  - En general mantienen una interfaz común y ocultan los detalles de la estructura que iteran, evitando tener que hacer distintos algoritmos para recorrer distintos contenedores (algoritmo genéricos)
  - No destruye la estructura que recorre, por lo que evita hacer una copia innecesaria de la estructura antes de recorrerla
  - Podemos usar iteradores como "punteros seguros" a la estructura interna sin exponerla.

Veamos un ejemplo de una implementación...

Interfaz REDUCIDA de Lista

```
template <typename T>
class Lista {
    public:
        Lista(); /// Vacia()
        Lista(const Lista& otra); /// Copiar()
        ~Lista(); /// Destruye la lista

        /// Opreaciones básicas
        bool EsVacia() const;
        Nat Longitud() const;
        void Fin(); /// Elimina el primer elemento
        void Comienzo(); /// Elimina el último elemento

        const T& Primero() const; /// O(1)
        const T& Ultimo() const; /// O(1)
        const T& operator[](Nat i) const; /// Operador "iésimo" O(n)

        /// Inserción de elementos
        Iterador AgregarAdelante(const T& elem);
        Iterador AgregarAtras(const T& elem);

    private:
```

Estructura interna de Lista

```
template <typename T>
class Lista {
    public:
        ...

    private:
        struct Nodo {
            Nodo(const T& d) :
                dato(d), anterior(NULL), siguiente(NULL) {};

            T dato;
            Nodo* anterior;
            Nodo* siguiente;
        };

        Nodo* primero;
        Nat longitud;
};
```

Hasta acá no hay nada nuevo!

¿Dónde definimos el iterador?

```
template <typename T>
class Lista {
    public:
        /*********************************
         * Iterador de Lista, modificable *
         *********************************/
        class Iterador {
            public:
                ...

            private:
                ...
        };

        ... /// resto de la parte pública del tipo Lista

};
```

Interfaz

```
class Iterador {
    public:
        Iterador() : lista(NULL), nodo_siguiente(NULL) {}
        Iterador(const typename Lista<T>::Iterador& otro) :
            lista(otro.lista),
            nodo_siguiente(otro.nodo_siguiente) {}

        bool HayAnterior() const;
        bool HaySiguiente() const;

        T& Anterior() const;
        T& Siguiente() const;

        void Avanzar();
        void Retroceder();

        ...
```

Interfaz

```
    ...

    void EliminarAnterior();
    void EliminarSiguiente();

    void AgregarComoAnterior(const T& elem);
    void AgregarComoSiguiente(const T& elem);

    bool operator==(const typename Lista<T>::Iterador& otro) const;

private:
    ...
};
```

Interfaz

Hay algo raro en la interfaz?

Interfaz

Hay algo raro en la interfaz? El constructor no recibe una lista

Interfaz

Hay algo raro en la interfaz? El constructor no recibe una lista

Por qué? Para qué sirve entonces el constructor? Es necesario?

Interfaz

Hay algo raro en la interfaz? El constructor no recibe una lista

Por qué? Para qué sirve entonces el constructor? Es necesario?

Dónde definimos el constructor de Iterador?

Estructura interna del Iterador

```
class Iterador {
    public:
        ...

    private:
        /// El constructor es privado, necesitamos el friend.
        Iterador(Lista<T>* _lista, typename Lista<T>::Nodo* _proximo)
            : lista(_lista), nodo_siguiente(_proximo) {};

        friend typename Lista<T>::Iterador Lista<T>::CrearIt(); /// lo qué?
        friend typename Lista<T>::Iterador Lista<T>::CrearItUlt(); /// lo qué?

        Lista<T>* lista;
        typename Lista<T>::Nodo* nodo_siguiente;
};
```

Revisando en detalle

```
friend typename Lista<T>::Iterador Lista<T>::CrearIt(); /// lo que?
```

[1] http://www.cplusplus.com/doc/tutorial/inheritance/

Revisando en detalle

```
friend typename Lista<T>::Iterador Lista<T>::CrearIt(); /// lo que?
```

- friend: permite acceder a los miembros privados y protegidos de una clase desde otra clase o método[1]

[1] http://www.cplusplus.com/doc/tutorial/inheritance/

Revisando en detalle

```
friend typename Lista<T>::Iterador Lista<T>::CrearIt(); /// lo que?
```

- friend: permite acceder a los miembros privados y protegidos de una clase desde otra clase o método[1]
- typename: le indica al compilador que lo que sigue es una clase y NO una variable estática o de clase

[1] http://www.cplusplus.com/doc/tutorial/inheritance/

Veamos el código del iterador (1)

```
template <typename T>
typename Lista<T>::Iterador Lista<T>::CrearIt()
{
    return Iterador(this, primero);
}

template <typename T>
typename Lista<T>::Iterador Lista<T>::CrearItUlt()
{
    return Iterador(this, NULL);
}

template <typename T>
bool Lista<T>::Iterador::HaySiguiente() const
{
    return nodo_siguiente != NULL;
}
```

Veamos el código del iterador (2)

```
template <typename T>
bool Lista<T>::Iterador::HayAnterior() const
{
    return nodo_siguiente != lista->primero;
}

template <typename T>
T& Lista<T>::Iterador::Siguiente() const
{
    assert(HaySiguiente());
    return nodo_siguiente->dato;
}

template <typename T>
T& Lista<T>::Iterador::Anterior() const
{
    assert(HayAnterior());
    return SiguienteReal()->anterior->dato;
}
```

Veamos el código del iterador (3)

```
template <typename T>
void Lista<T>::Iterador::Avanzar()
{
    assert(HaySiguiente());
    nodo_siguiente = nodo_siguiente->siguiente;
    if(nodo_siguiente == lista->primero) nodo_siguiente = NULL;
}

template <typename T>
void Lista<T>::Iterador::Retroceder()
{
    assert(HayAnterior());
    nodo_siguiente = SiguienteReal()->anterior;
}
```

Veamos el código del iterador (4)

```
template <typename T>
void Lista<T>::Iterador::AgregarComoAnterior(const T& dato)
{
    Nodo* sig = SiguienteReal();
    Nodo* nuevo = new Nodo(dato);
    //asignamos anterior y siguiente de acuerdo a si el nodo es el primero
    //o no de la lista circular
    nuevo->anterior = sig == NULL ? nuevo : sig->anterior;
    nuevo->siguiente = sig == NULL ? nuevo : sig;
    //reencadenamos los otros nodos (notar que no hay problema cuando nuevo
    //es el primer nodo creado de la lista)
    nuevo->anterior->siguiente = nuevo;
    nuevo->siguiente->anterior = nuevo;
    //cambiamos el primero en el caso que nodo_siguiente == primero
    if(nodo_siguiente == lista->primero)
        lista->primero = nuevo;

    lista->longitud++;
}
```

Veamos el código del iterador (5)

```
template <typename T>
void Lista<T>::Iterador::AgregarComoSiguiente(const T& dato)
{
    AgregarComoAnterior(dato);
    Retroceder();
}

template <typename T>
void Lista<T>::Iterador::EliminarAnterior()
{
    assert(HayAnterior());
    Retroceder();
    EliminarSiguiente();
}
```

Veamos el código del iterador (6)

```
template <typename T>
void Lista<T>::Iterador::EliminarSiguiente()
{
    assert(HaySiguiente());

    Nodo* tmp = nodo_siguiente;
    //reencadenamos los nodos
    tmp->siguiente->anterior = tmp->anterior;
    tmp->anterior->siguiente = tmp->siguiente;
    //borramos el unico nodo que habia?
    nodo_siguiente = tmp->siguiente == tmp ? NULL : tmp->siguiente;
    //borramos el último?
    nodo_siguiente = tmp->siguiente == lista->primero ? NULL : tmp->siguiente;

    if(tmp == lista->primero) //borramos el primero?
        lista->primero = nodo_siguiente;

    delete tmp;
    lista->longitud--;
}
```

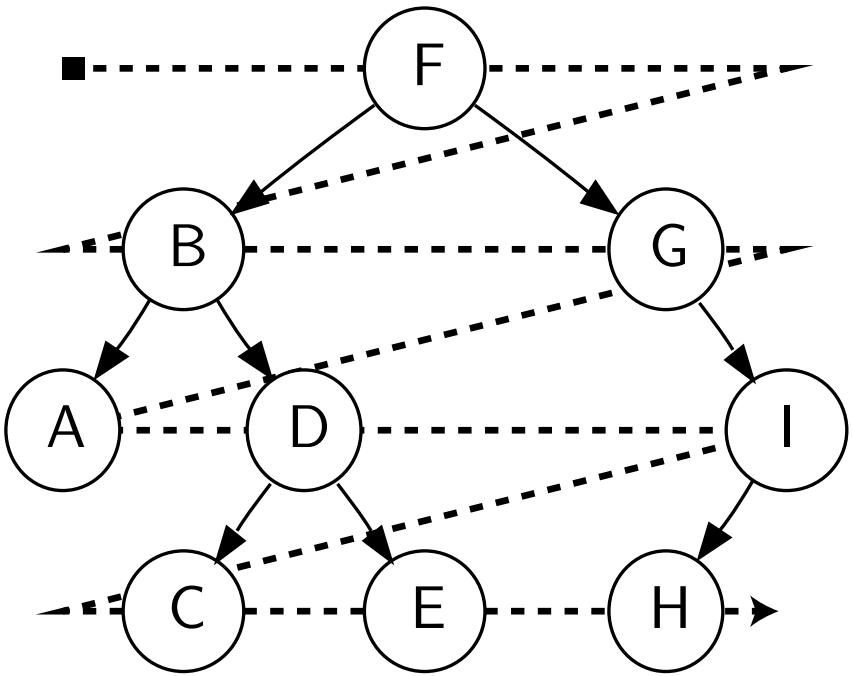
Veamos el código del iterador (7)

```
template<class T>
bool Lista<T>::Iterador::operator==(const typename Lista<T>::Iterador& otro) co
  return lista == otro.lista && nodo_siguiente == otro.nodo_siguiente;
}

template <typename T>
typename Lista<T>::Nodo* Lista<T>::Iterador::SiguienteReal() const {
    return nodo_siguiente == NULL ? lista->primero : nodo_siguiente;
}
```

Representación de árbol binario

```
template <typename T>
class ArbolBinario {
    public:
        ...

    private:

        struct Nodo {
            Nodo(const T& d) :
                dato(d), izq(NULL), der(NULL) {};

            T dato;
            Nodo* izq;
            Nodo* der;
        };

        Nodo* raiz;
};
```

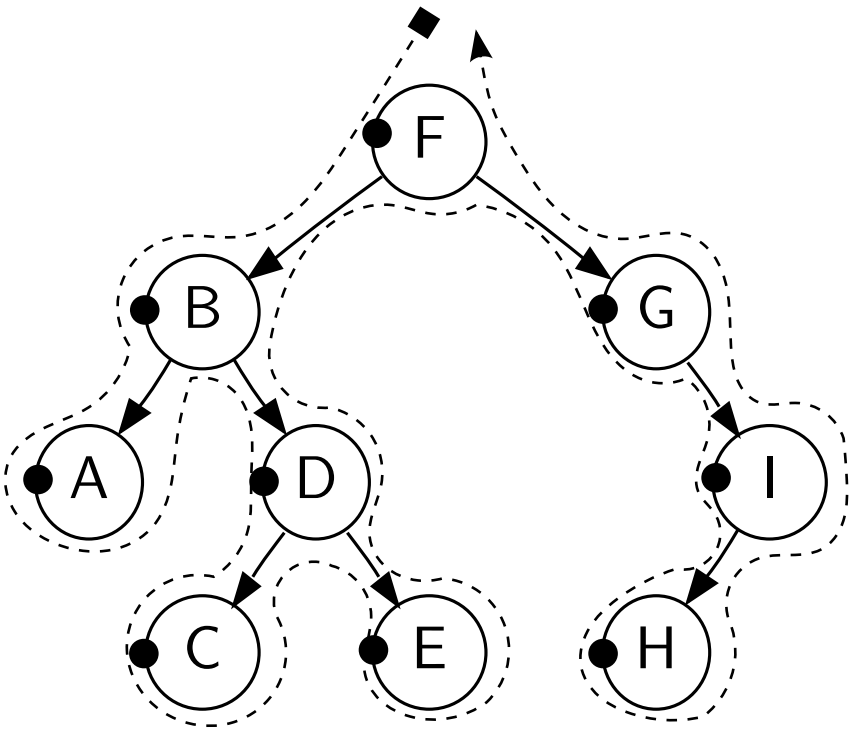
BFS en árbol binario



orden: F, B, G, A, D, I, C, E, H.

```
void recorrerEnAncho(Nodo* nodo)
{
  if ( nodo == NULL )
    return;

  Cola<Nodo*> q;
  q.encolar( nodo );

  while ( not cola.esVacia() )
  {
    proximo_nodo = cola.desencolar();
    f( proximo_nodo );

    if ( proximo.izq != NULL )
      q.encolar( proximo.izq );

    if ( proximo.der != NULL )
      q.encolar( proximo.der );
  }
}
```

DFS en árbol binario



orden: F, B, A, D, C, E, G, I, H.

```
void recorrerEnAltura(Nodo* nodo)
{
  if ( nodo == NULL )
    return;

  Pila<Nodo*> s;
  s.apilar( nodo );

  while ( not pila.esVacia() )
  {
    proximo_nodo = pila.desapilar();
    f( proximo_nodo );

    if ( proximo.der != NULL )
      q.apilar( proximo.der );

    if ( proximo.izq != NULL )
      q.apilar( proximo.izq );
  }
}
```

Ejercicios más complejos

1) Iterador de Diccionario

Tomar el código del módulo Diccionario subido a la página y dotar al mismo de un iterador[a].

[a] Pista: El iterador puede "devolver" tuplas del estilo <Clave, Significado>

2) Iterador de árbol binario

Tomar el código del Taller de ABB de la clase anterior y dotar al mismo de un iterador[a].

[a] Pista: Van a tener que usar alguna estructura auxiliar como una pila o cola