ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS LINEALES

Iteradores y listas enlazadas

Manuel Montenegro Montes
Departamento de Sistemas Informáticos y Computación
Facultad de Informática – Universidad Complutense de Madrid

Operaciones a implementar

Con listas circulares doblemente enlazadas.

Para iteradores

recordamos las operaciones que hemos visto en las primeras diapos de la semana.

- Obtener el elemento apuntado por el iterador (***elem***)
- Avanzar el iterador a la siguiente posición de la lista (***advance***)
- Igualdad entre dos iteradores (==)
  - Dos iteradores son iguales si recorren la misma lista y apuntan a la misma posición dentro de esta.

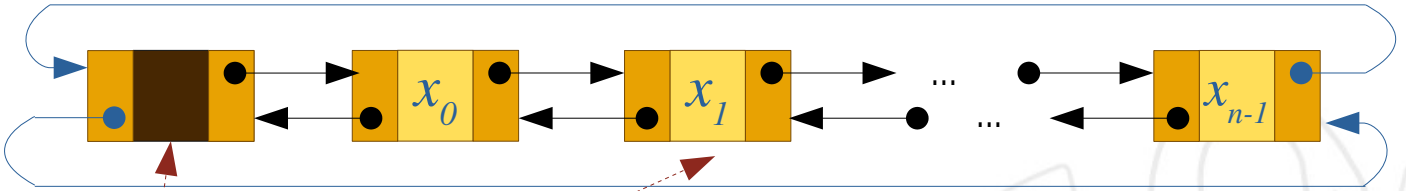
Tenemos que ver si el iterador actual es distinto de it.end() para saber si continuamos iterando por la lista o no.

Para listas

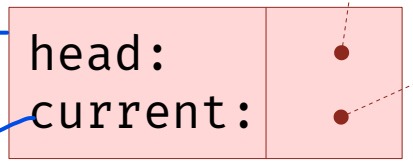
- Obtener un iterador al principio de la lista (***begin***)
- Obtener un iterador al final de la lista (***end***)

Iteradores en listas doblemente enlazadas

Circulares.



Iterador



- Un iterador contiene dos atributos:
  - El nodo del elemento apuntado por el iterador (`current`).
  - El nodo fantasma de la lista a la que el iterador pertenece (`head`).

Va a ser útil para comparar listas, para saber si iterador ha llegado al final de la lista.

La clase ListLinkedDouble<T> :: iterator

```
template <typename T>
class ListLinkedDouble {
public:
  …
  class iterator {
  public:
    iterator(Node *head, Node *current): head(head), current(current) { }
    …

  private:
    Node *head;
    Node *current;
  };
  …
};
```

Definimos la clase iterator DENTRO DE LA PROPIA CLASE.

No se si con lo de spl es necesario hacer esto.

HEAD Y CURRENT PUNTEROS A NODOS.

En realidad este CONSTRUCTOR DEBERÍA DE SER PRIVATE. Y para poder CREAR INSTANCIAS DE LA CLASE iterator debemos escribir FRIEND CLASS LISTLINKEDDOUBLE.

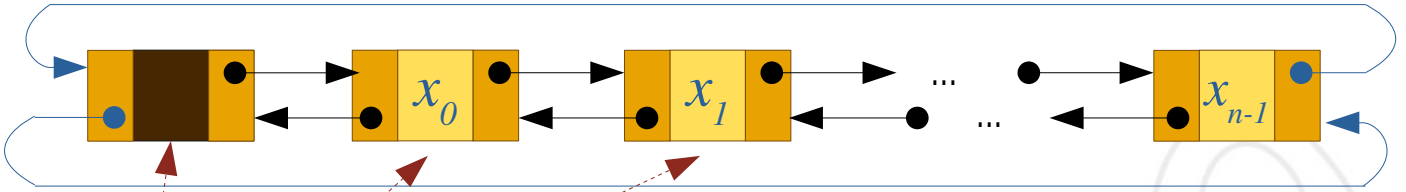
De forma que ListLinkedDouble puede acceder a los atributos y métodos privados de la clase iterator.

La clase ListLinkedDouble<T> :: iterator

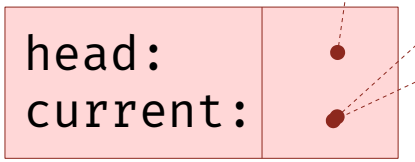
```
template <typename T>
class ListLinkedDouble {
public:
  …
  class iterator {
  public:
    …
    void advance();
    T & elem();
    bool operator==(const iterator &other) const;
    bool operator!=(const iterator &other) const;
    …

  };
  …
};
```

VEREMOS LA IMPLEMENTACIÓN DE ESTOS 4 MÉTODOS.

Implementación de advance( )



Iterador



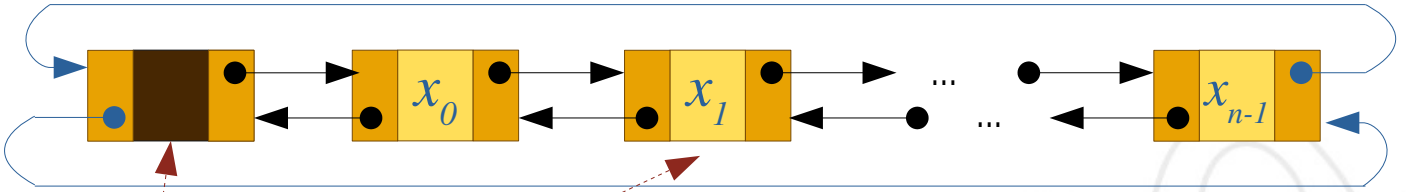

```
class iterator {
  public:
    void advance() {
      assert (current ≠ head);
      current = current→next;
    }
    …
};
```

avanza el current al siguiente.

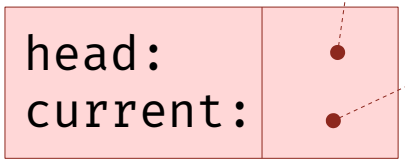
Cambia el puntero current al siguiente, current.next-

coste constante

Implementación de elem( )



Iterador




```
class iterator {
  public:
    T & elem() {
      assert (current ≠ head);
      return current→value;
    }
    …
};
```

Implementación de los operadores == y !=

```
class iterator {
  public:
    bool operator==(const iterator &other) const {
      return (head == other.head) &&
             (current == other.current);
    }

    bool operator!=(const iterator &other) const {
      return !(*this == other);
    }

};
```

tienen que tener el mismo nodo fantasma y además estar apuntando al mismo nodo de la misma lista. No al mismo nodo con el mismo valor si no que al mismo nodo.

se refieren al objeto this.

Creación de iteradores

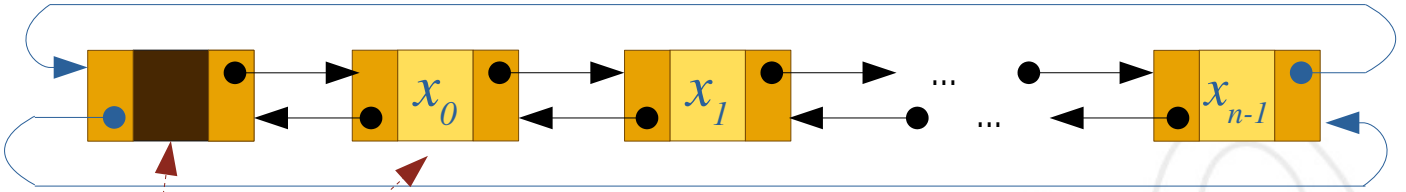
```
template <typename T>
class ListLinkedDouble {
public:
  class iterator { … }
   …
  iterator begin();
  iterator end();
   …
};
```

Devuelve un iterador al principio de la lista.

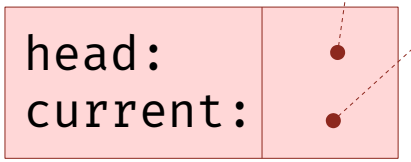
Nuevos métodos

Devuelve un iterador al final de la lista.

Implementación de begin( )



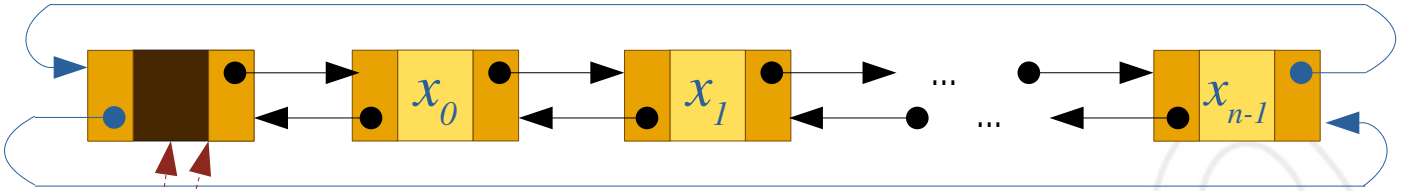
Iterador



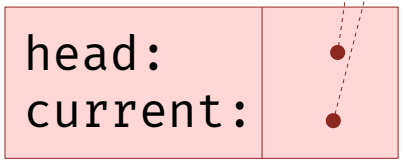

```
template <typename T>
class ListLinkedDouble {
   ...
  iterator begin() {
    return iterator(head, head→next);
  }
};
```

El siguiente al nodo fantasma.

Implementación de end( )



Iterador




```
template <typename T>
class ListLinkedDouble {
   ...
  iterator end() {
    return iterator(head, head);
  }
};
```

Iterador que apunta a la cabeza. Si el iterador apunta a la cabeza de la lista es señal de que hemos dejado de iterar

Ejemplo

- Dada una lista de nombres de personas, imprimir aquellas que empiecen por un carácter pasado como parámetro:

```
void imprime_empieza_por(ListLinkedDouble<std::string> &l, char c) {
  for (ListLinkedDouble<std::string>::iterator it = l.begin();
       it != l.end();
       it.advance()) {

    if (it.elem()[0] == c) {
      std::cout << it.elem() << std::endl;
    }

  }
}
```

Avanza mientras que NO sea end.

Si empieza por C

Si es igual imprime cadena.

Aquellas cadenas que empiecen con el char c deben ser impresas.

Ejemplo

- Modificar todos los elementos de una lista de enteros, multiplicándolos por uno pasado como parámetro.

```
void multiplicar_por(ListLinkedDouble<int> &l, int num) {
  for (ListLinkedDouble<int>::iterator it = l.begin();
       it != l.end();
       it.advance()) {
    it.elem() = it.elem() * num;
  }
}
```

Quedarnos con el cuerpo de este bucle.

como le pasamos una REFERENCIA, SI se MODIFICA la lista.

```
class iterator {
public:
  …
  void advance();
  T & elem();
  bool operator==(const iterator &other) const;
  bool operator!=(const iterator &other) const;
  …
};
```