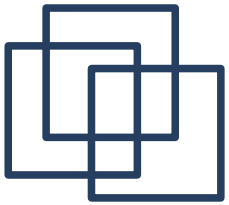
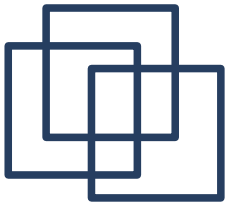


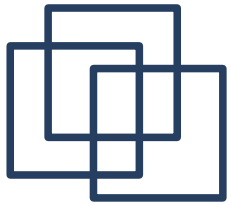
Listas Enlazadas

- **Historia:** contenedor que permite:
  - Desarrolladas por Allen Newell, Cliff Shaw y Herbert Simon en 1955-56 en RAND Corporation como estructura de datos primitiva para el Lenguaje de Proceso de Inofrmación.



Listas

- **Descripción:** contenedor que permite
  - Almacenar una secuencia de elementos permitiendo el recorrido hacia delante como hacia atrás.
  - La insercion y borrado se puede realizar en un orden constante
  - La inserción y borrado de elementos en la lista no invalida los iteradores.
  - **No** permite el acceso aleatorio (mediante valores enteros) a los elementos
  - El número de elementos puede variar dinámicamente



List: Representación

Listas doblemente enlazadas

- Cada nodo apunta a su predecesor y su sucesor
- Se suele utilizar nodos “nulos” para faciliar el mantenimiento de la lista, por ejemplo en la cabecera de la lista

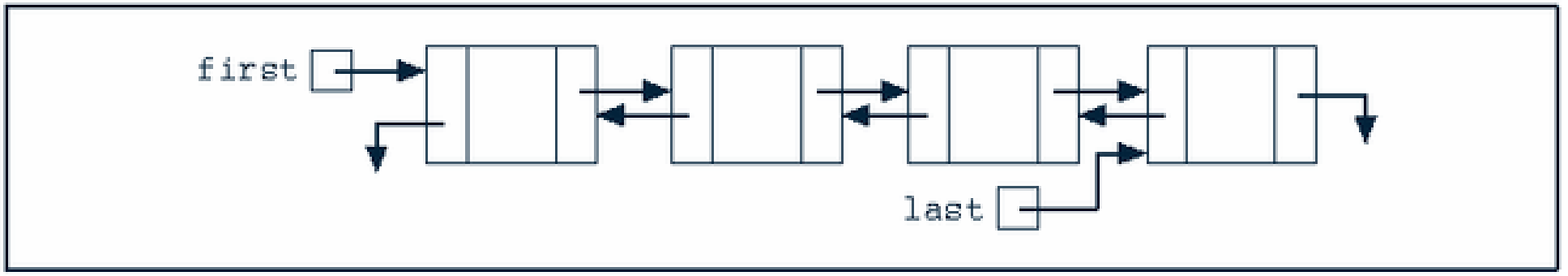
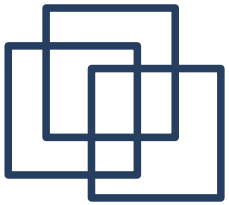
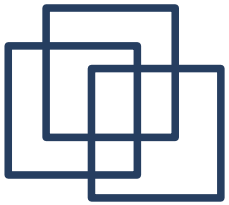


Figure 5-42 Doubly linked list



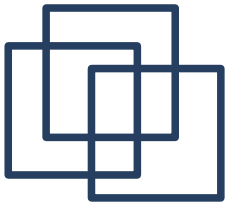
List: representación

```
template<typename T> class list {
 private:
    struct nodo_lista {
      nodo_lista * anterior;
      T elemento;
      nodo_lista * siguiente;
    };
   typedef nodo_lista * Posicion;
    Posicion cab;
    size_type num_elem;
```



List: Función de Abstracción

```
/**
  list<T> modela una lista de elementos de
tipo T: L=<a1, a2, ,...,an>

  Dado el objeto del tipo rep r, r = {cab,
num_elem}, la lista que representa

     l = < r.cab->siguiente->elemento,

           r.cab->siguiente->siguiente-
>elemento,

           ...

           r.cab->siguiente-> (r.num_elem)
->siguiente->elemento >

**/
```



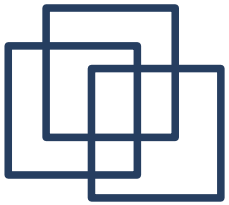
List: Inv. Representación

- 0 <= r.num_elem
- Si (r.cab->anterior == r.cab->siguiente)

  entonces r = < >
- Es una representación de lista circular:
  - Dada una lista l, se verifica que l.end() == r.cab
  - r.cab->anterior == r.cab->siguiente-> (r.num_elem) ->siguiente->elemento
- Para cualquier posición p de l:

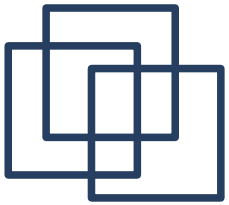
  p->siguiente->anterior == p

  p->anterior->siguiente == p



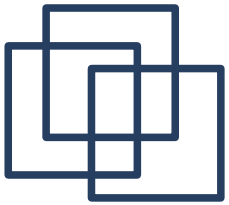
list.h

```
template <typename T> class list {
public:
  class iterator;
  class const_iterator;
  typedef unsigned int size_type;

 list();
 list(const list<T> & l);
 list(iterator inicio, iterator final);
 bool empty() const;
 void clear ();
 size_type size() const;
```



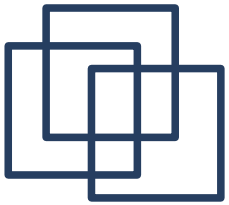
list.h

```
template <typename T> class list {
public:
  void push_back(const T &dato);
  void push_front(const T &dato);
  void pop_back();
  void pop_front();
  T & back();
  T & front();
  const T & back() const;
  const T & front() const;
  list<T> & operator=(const list<T> & l);
  bool operator==(const list<T> & l) const;
  void swap (list<T> & l);
```



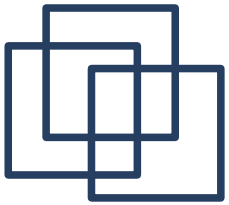
list.h

```
template <typename T> class list {
public:
   .......
   iterator insert(iterator p, const T &
  elemento);
   iterator erase(iterator p);
   iterator begin();
   const_iterator begin() const;
   iterator end();
   const_iterator end() const;
   void splice( iterator pos, list<T> & L,
   iterator f, iterator l);
   ~list();
  ..........
```



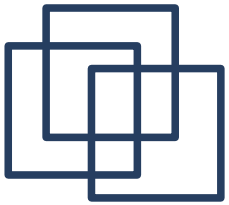
list.h

```
template <typename T> class list {
public:
  class iterator {
      public:
    iterator();
    iterator(const iterator & i);
    T & operator*();
    iterator & operator++();
    iterator & operator--();
    bool operator==(const iterator &i) const;
    bool operator!=(const iterator &i) const;
  private:
    iterator(nodo_lista * p);
    nodo_lista * eliterador;
     };
```



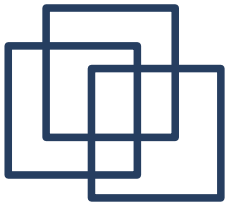
list.hxx

```
template <typename T>
list<T>::list()
{
  cab = new nodo_lista;
  assert(cab != 0);
  cab->siguiente = cab;
  cab->anterior = cab;
  num_elem = 0;
}
```



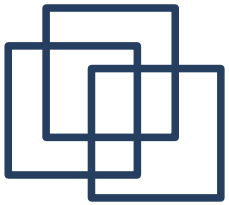
list.hxx

```
template <typename T>
bool  list<T>::empty() const
{
  return num_elem == 0;
}


template <typename T>
typename list<T>::size_type list<T>::size() const
{
  return num_elem;
}
}
```



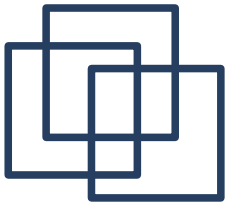
list.hxx

```
template<typename T> list<T>::list(const list<T>
  & l)
{  num_elem = l.num_elem;
   // crear nodo cabecera
  cab = new nodo_lista;   assert(cab != 0);
  cab->siguiente = cab;   cab->anterior = cab;

  Posicion p = l.cab->siguiente;
  while (p != l.cab)    {
      Posicion q = new nodo_lista;
      assert(q != 0);
      q->elemento = p->elemento;
      q->anterior = cab->anterior;
      q->siguiente = cab;
      cab->anterior->siguiente = q;
      cab->anterior = q;
      p = p->siguiente;
    }
}
```



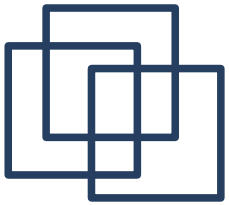
list.hxx

```
template <typename T>
typename list<T>::iterator
list<T>::insert(list<T>::iterator pos, const T &
  elemento)
{  Posicion q = new nodo_lista;
  assert(q != 0);
  q->anterior = pos.eliterador->anterior;
  q->elemento = elemento;
  q->siguiente = pos.eliterador;
  pos.eliterador->anterior = q;
  q->anterior->siguiente = q;
  num_elem++;

  return iterator(q);
}
```



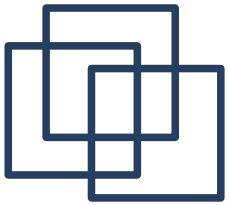
list.hxx

```
template <typename T>
typename list<T>::iterator
list<T>::erase(list<T>::iterator pos)
{  if (pos != end())
    {  Posicion q = pos.eliterador;
       pos.eliterador->anterior->siguiente =
       pos.eliterador->siguiente;
       pos.eliterador->siguiente->anterior =
       pos.eliterador->anterior;
       pos.eliterador = q->siguiente;
       num_elem--;
       delete q;
    }
  return pos;
}
```



list.hxx

```
template<typename T>
void
list<T>::push_front(const T & elemento)
{
  insert(begin(), elemento);
}

template<typename T>
void
list<T>::push_back(const T & elemento)
{
  insert(end(), elemento);
}
```

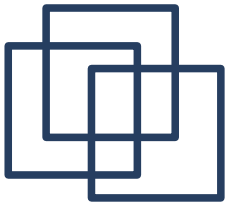


list.hxx

```
template <typename T> void  list<T>::clear()
{
  if (!empty())
    {  Posicion p = cab->siguiente;
      while (p != cab)
        {    Posicion q = p;
             p = p->siguiente;
             delete q;
        }
    }
  cab->siguiente = cab;
  cab->anterior = cab;
  num_elem = 0;
}
```

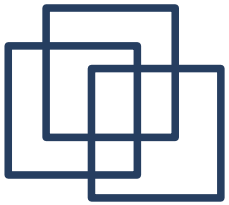
list.hxx

```
template <typename T>
void list<T>::splice( list<T>::iterator pos,
  list<T> & L, list<T>::iterator f,
  list<T>::iterator l)
{  Posicion ptr_l = l.eliterador;
   Posicion ptr_f_ant = f.eliterador->anterior;
// Enganchamos en this
pos.eliterador->anterior->siguiente =
  f.eliterador;
l.eliterador->anterior->siguiente =
  pos.eliterador;
f.eliterador->anterior = pos.eliterador-
  >anterior;
pos.eliterador->anterior = l.eliterador-
  >anterior;
// Quitamos de L
ptr_f_ant->siguiente = l.eliterador;
l.eliterador->anterior = ptr_f_ant;
}
```



¿Y el iterador?

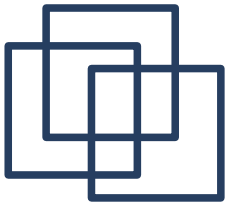
- Tipos del TDA list
  - size_type: Entero sin signo
  - iterator: Iterator sobre list
  - const_iterator: El iterator constante de una list.
  - reverse_iterator: Iterator que recorre en orden inverso una list
  - const_reverse_iterator: Iterator constante que recorre en orden inverso una list



list e iterator (1)

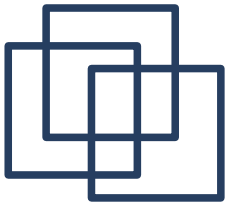
```
template<typename T> class list
   { public:      .........
      class iterator {
        public:
                iterator();
                 .......
        private:
            iterator(nodo_lista * p);

          nodo_lista * eliterador;
          friend class list;
          friend class const_iterator;
                .....
      };
```

```
      . . . . .
      iterator begin( ) const;
      iterator end() const;
      ....
private: // de la clase list
......
};
```



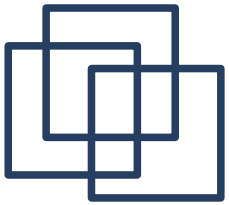
list e iterator (2)

```
template<typename T> class list {
  class iterator {
    public:
       iterator();
       iterator( const list<T>::iterator & it);
       T & operator*();
       iterator & operator++();
       iterator & operator=(const list<T>::iterator & it);
       bool operator==(const list<T>::iterator & it) const;
       bool operator!=(const list<T>::iterator & it) const;
   private:
    .....
```



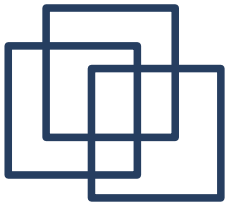
list e iterator (2)

```
template <typename T>
inline typename list<T>::iterator
list<T>::begin()
{
  return iterator(cab->siguiente);
}


template <typename T>
inline typename list<T>::iterator
list<T>::end()
{
  return iterator(cab);
}
```



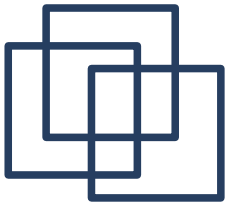
list e iterator (3)

```
template <typename T> list<T>::iterator::iterator()
{   eliterador=0; };

template <typename T> list<T>::iterator::iterator(const
  list<T>::iterator & i)
{ eliterador = i.eliterador; }

template <typename T>
  list<T>::iterator::iterator(list<T>::nodo_lista * p)
{ eliterador = p; }


template <typename T> T & list<T>::iterator::operator*()
{  return eliterador->elemento;}
```



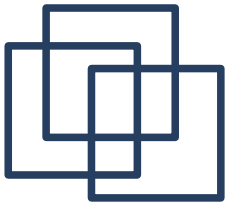
list e iterator (3)

```
template <typename T> typename list<T>::iterator &
list<T>::iterator::operator++()
{   eliterador = eliterador->siguiente;
    return *this; }

template <typename T>
bool  list<T>::iterator::operator!=(const
  list<T>::iterator & i) const
{   return eliterador != i.eliterador; }

template <typename T> bool
list<T>::iterator::operator==(const list<T>::iterator
  & i) const
{
  return eliterador == i.eliterador;
}
```



Ejercicios

- Implementar const_iterator
- Implementar reverse_iterator
- Implementar const_reverse_iterator
- Implementar las operaciones que faltan de la lista



Slist vs list

- **Slist: Listas simplemente enlazadas**
  - Class nodo_lista {

    T data;

    nodo_lista * siguiente;

    }
- **Lista doblemente enlazada**
  - Class nodo_lista {

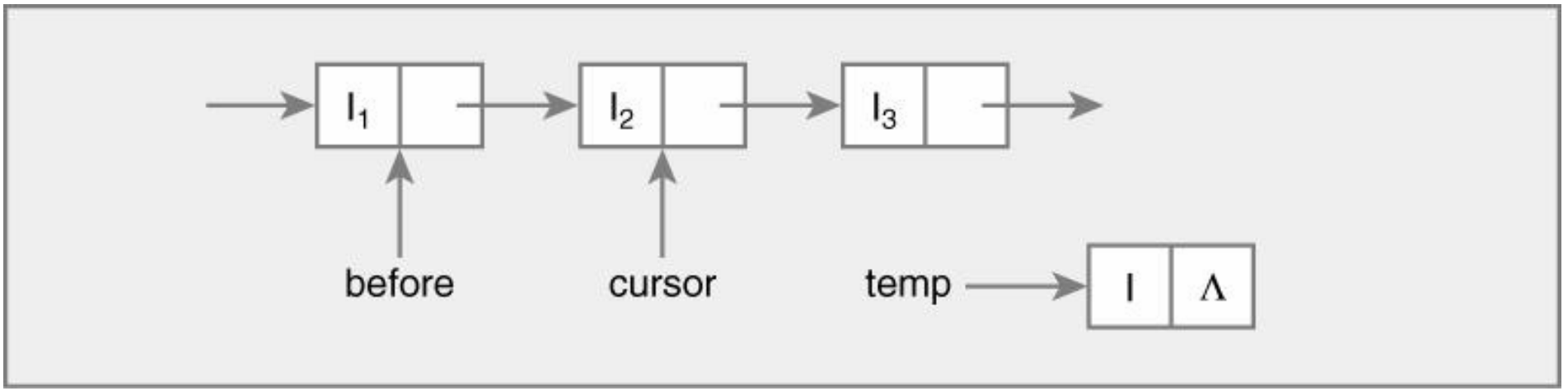
    nodo_lista * anterior;

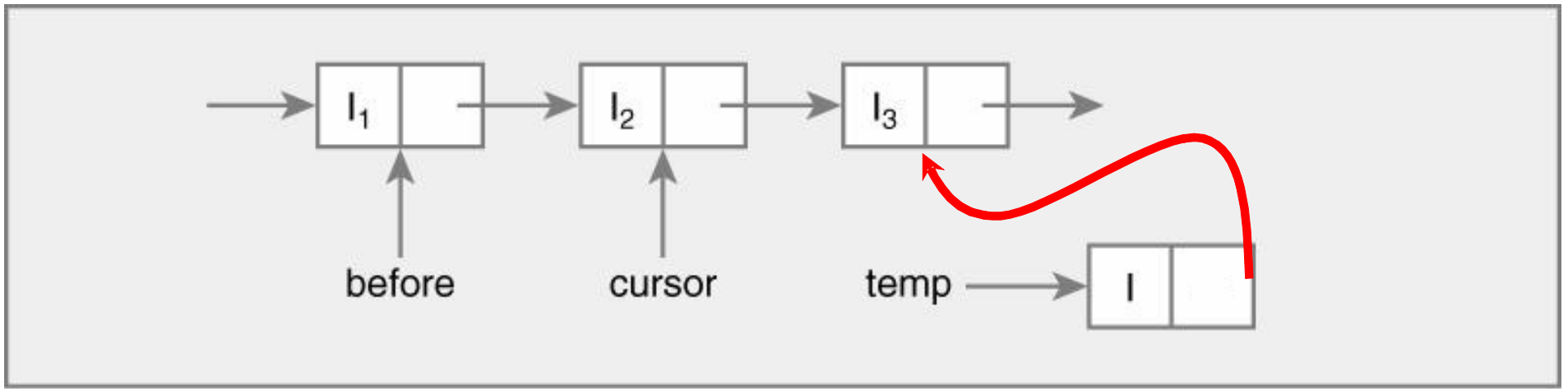
    T data;

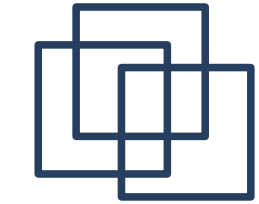
    nodo_lista * siguiente;

    }

Listas simplemente enlazadas







Listas simplemente enlazadas