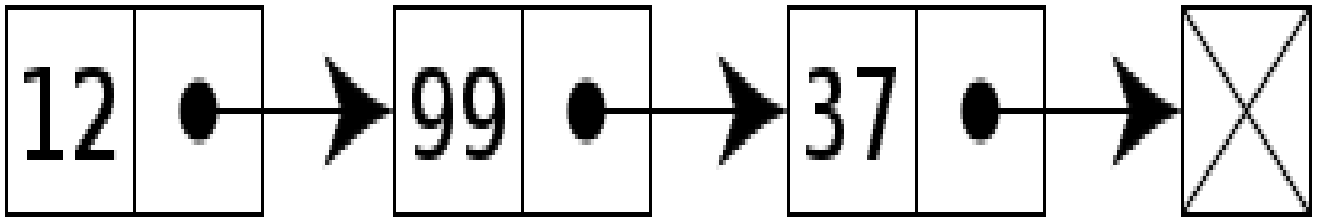
Listas

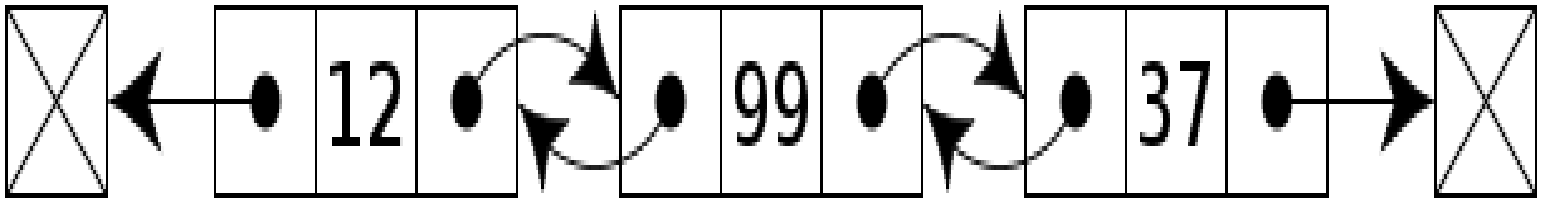
- Una lista ligada es una estructura de datos ligados que consiste en un grupo de nodos que juntos representan una secuencia.
- Es un tipo de dato abstracto que se suele usar como base para otros tipos como stacks, queues, associative arrays, y S-expressions.

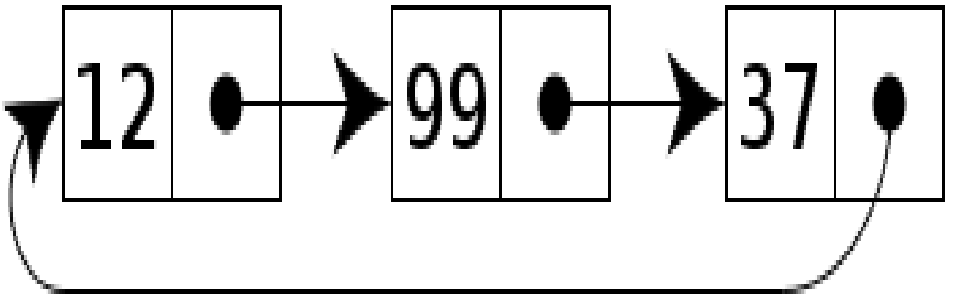
Listas

Ventajas	Desventajas
Memoria dinámica	Desperdicio de memoria
Facilidad para insertar y borrar un nodo	Acceso secuencial
Estructuras de datos lineales son fáciles de implementar con listas	Largo tiempo para accesar cada nodo
Se pueden expandir en tiempo real	Las listas simples son dificiles de recorrer en reversa, y las dobles gastan mucha memoria.

Listas simples y doblemente ligadas







```
namespace std {
    #include <initializer_list>
    template <class T, class Allocator = allocator<T> > class list;

    template <class T, class Allocator>
      bool operator==(const list<T,Allocator>& x, const list<T,Allocator>& y);
    template <class T, class Allocator>
      bool operator< (const list<T,Allocator>& x, const list<T,Allocator>& y);
    template <class T, class Allocator>
      bool operator!=(const list<T,Allocator>& x, const list<T,Allocator>& y);
    template <class T, class Allocator>
      bool operator> (const list<T,Allocator>& x, const list<T,Allocator>& y);
    template <class T, class Allocator>
      bool operator>=(const list<T,Allocator>& x, const list<T,Allocator>& y);
    template <class T, class Allocator>
      bool operator<=(const list<T,Allocator>& x, const list<T,Allocator>& y);

    template <class T, class Allocator>
      void swap(list<T,Allocator>& x, list<T,Allocator>& y);

}
```

Asignador - Allocator

- Los asignadores son clases que definen los modelos de memoria a ser usados por algunas partes de la libreria estandar, es especifico, por los contenedores STL.

- El template por default del asignador “allocator” (minusculas) es le que todos los contenedores estandar usaran si su ultimo parametro (opcional) no es especificado.

Variable T

- Debe cumplir con los requisitos de **copyAssignable** y **copyConstructible**
  - (Hasta C++11)
- Los requerimientos dependen de las operaciones realizadas por los contenedores, generalmente con si es un tipo completo y es borrable es suficiente.

Miembro Tipo	Definiciòn
value_type	T
allocator_type	Asignador
size_type	Tipo entero sin signo (normalmente std::size_t)
difference_type	Tipo entero con signo
reference	Allocator::reference (until C++11) value_type& (since C++11)
const_reference	Allocator::const_reference (until C++11) const value_type& (since C++11)
pointer	Allocator::pointer (until C++11) std::allocator_traits<Allocator>::pointer (since C++11)
const_pointer	Allocator::const_pointer (until C++11) std::allocator_traits<Allocator>::const_pointer (since C++11)
iterator	Iterador Bidireccional
const_iterator	Iterador Bidireccional constante
reverse_iterator	std::reverse_iterator<iterator>
const_reverse_iterator	std::reverse_iterator<const_iterator>

```
template <class T, class Allocator = allocator<T> >
class list {
public:
    // types:
    typedef value_type&                                         reference;
    typedef const value_type&                                   const_reference;
    typedef /*implementation-defined*/                          iterator;
    typedef /*implementation-defined*/                          const_iterator;
    typedef /*implementation-defined*/                          size_type;
    typedef /*implementation-defined*/                          difference_type;
    typedef T                                                    value_type;
    typedef Allocator                                           allocator_type;
    typedef typename allocator_traits<Allocator>::pointer       pointer;
    typedef typename allocator_traits<Allocator>::const_pointer const_pointer;
    typedef std::reverse_iterator<iterator>                     reverse_iterator;
    typedef std::reverse_iterator<const_iterator>               const_reverse_iterator;
```

```
//     construct/copy/destroy:
 explicit list(const Allocator& = Allocator());
 explicit list(size_type n);
list(size_type n, const T& value,const Allocator& = Allocator());
 template <class InputIterator>
        list(InputIterator first, InputIterator last,const Allocator& =
Allocator());
    list(const list<T,Allocator>& x);
list(list&&);
 list(const list&, const Allocator&);
 list(list&&, const Allocator&);
list(initializer_list<T>, const Allocator& = Allocator());
~list();
 list<T,Allocator>& operator=(const list<T,Allocator>& x);
 list<T,Allocator>& operator=(list<T,Allocator>&& x);
 list& operator=(initializer_list<T>);
```

```
template <class InputIterator>
        void assign(InputIterator first, InputIterator last);
    void assign(size_type n, const T& t);
    void assign(initializer_list<T>);
    allocator_type get_allocator() const noexcept;

    // iterators:
    iterator                begin() noexcept;
    const_iterator          begin() const noexcept;
    iterator                end() noexcept;
    const_iterator          end() const noexcept;

    reverse_iterator        rbegin() noexcept;
    const_reverse_iterator  rbegin() const noexcept;
    reverse_iterator        rend() noexcept;
    const_reverse_iterator  rend() const noexcept;

    const_iterator          cbegin() noexcept;
    const_iterator          cend() noexcept;
    const_reverse_iterator  crbegin() const noexcept;
    const_reverse_iterator  crend() const noexcept;
```

```
// capacity:
size_type size()              const noexcept;
size_type max_size()          const noexcept;
void                          resize(size_type sz);
void                          resize(size_type sz, const T& c);
bool                          empty() const noexcept;
// element access:
reference                     front();
const_reference front()       const;
reference                     back();
const_reference back()        const;

// modifiers:
template <class... Args> void emplace_front(Args&&... args);
void pop_front();
template <class... Args> void emplace_back(Args&&... args);
void push_front(const T& x);
void push_front(T&& x);
void push_back(const T& x);
void push_back(T&& x);
void pop_back();
```

```
template <class... Args> iterator emplace(const_iterator position, Args&&...
args);

    iterator insert(const_iterator position, const T& x);
    iterator insert(const_iterator position, T&& x);
    iterator insert(const_iterator position, size_type n, const T& x);
    template <class InputIterator>
        iterator insert (const_iterator position, InputIterator first,
                         InputIterator last);
    iterator insert(const_iterator position, initializer_list<T>);


    iterator erase(const_iterator position);
    iterator erase(const_iterator first, const_iterator last);
    void     swap(list<T,Allocator>&);
    void     clear() noexcept;
```

```
// list operations:
   void splice(const_iterator position, list<T,Allocator>& x);

   void splice(const_iterator position, list<T,Allocator>&& x);

   void splice(const_iterator position, list<T,Allocator>& x,
               const_iterator i);

   void splice(const_iterator position, list<T,Allocator>&& x,
               const_iterator i);

   void splice(const_iterator position, list<T,Allocator>& x,
               const_iterator first, const_iterator last);

   void splice(const_iterator position, list<T,Allocator>&& x,
               const_iterator first, const_iterator last);
```

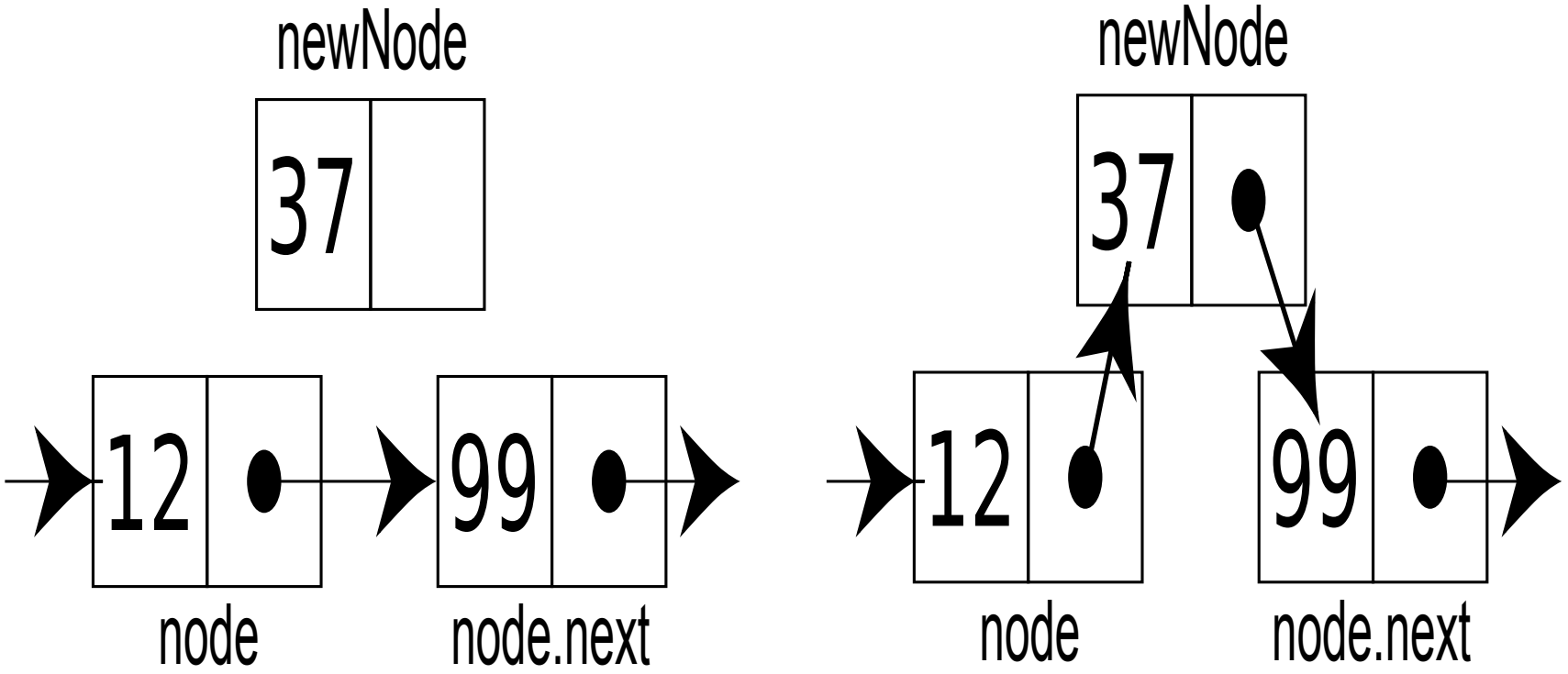
```
 void remove(const T& value);
    template <class Predicate> void remove_if(Predicate pred);

    void unique();
    template <class BinaryPredicate> void unique(BinaryPredicate
binary_pred);

    void merge(list<T,Allocator>& x);
    void merge(list<T,Allocator>&& x);
    template <class Compare> void merge(list<T,Allocator>& x, Compare
comp);
    template <class Compare> void merge(list<T,Allocator>&& x, Compare
comp);

    void sort();
    template <class Compare> void sort(Compare comp);

    void reverse() noexcept;
};
```


newNode
37
12
99
node
node.next
newNode
37
12
99
node
node.next