4.26 Контейнеры list и forward_list, их внутреннее устройство, примерная реализация основных методов (конструкторы, деструкторы, операторы присваивания, insert, erase, push/pop_back, push/pop_front). Правила инвалидации итераторов в list и forward_list.

std::list представляет собой контейнер, который поддерживает быструю вставку и удаление элементов из любой позиции в контейнере. Быстрый произвольный доступ не поддерживается. Он реализован в виде двусвязного списка. В отличие от **std::forward_list** этот контейнер обеспечивает возможность двунаправленного итерирования, являясь при этом менее эффективным в отношении используемой памяти.

forward_list - односвязный список,в котором отсутствует методы *back*(), *pop_back*()

Об устройстве:

Связный список aka **list**, хранит внутренний тип *Node* для хранения "вершинок" списка. Ноды хранят элемент листа и указатели на предыдущую и следующую вершинку (логично, что в **std::forward_list** хранится указатель только на следующую вершинку).

В полях листа хранится указатель на начало списка, размер листа и аллокатор.

Хранить список можно так: Делаем фейковую вершинку – **head**. Ссылкой next она указывает на первую ноду list'а, а ссылкой prev – на последнюю вершинку. Итератор указывает на *Node*.

```
template<typename T, typename Alloc = std::allocator<T>>
class List {
private:
    struct Node {
        T val;
        Node* next = nullptr;
        Node* prev = nullptr;
        explicit Node(const T& val) : val(val) {}
    };

    size_t sz = 0;
    Alloc t_allocator;
    typename Alloc::template rebind<Node>::other allocator;
    Node* head;

public:

    ////////////// ITERATORS ////////////
    template <bool IsConst>
    class common_iterator {
    public:
        std::conditional_t<IsConst, const Node*, Node*> ptr_node;
        /*.....*/
    };

    using t_node_alloc = std::allocator_traits<Alloc>;
    using node_alloc = std::allocator_traits
                            <typename Alloc::template rebind<Node>::other>;
    using iterator = common_iterator<false>;
};
```

Реализация основных методов

```
////////////// LIST METHODS ////////////
explicit List(const Alloc& alloc = Alloc()) : sz(0), t_allocator(alloc) {
    head = node_alloc::allocate(allocator, 1);
    head->next = head;
    head->prev = head;
}
explicit List(size_t n, /*const T& val,*/ const Alloc& alloc = Alloc()) : List(
    alloc) {
    if (n < 0) throw std::bad_alloc();
    size_t i = 0;
    try {
        for (; i < n; ++i) {
            Node *new_node = node_alloc::allocate(allocator, 1);
            /*node_alloc::construct(allocator, new_node, val);*/
            node_alloc::construct(allocator, new_node);
            link_nodes(head->next, new_node);
        }
    } catch (...) {
        for (size_t j = 0; j < i; ++j) {
            Node* copy_node = head->prev;
            del_node(head->prev);
            node_alloc::destroy(allocator, copy_node);
            node_alloc::deallocate(allocator, copy_node, 1);
        }
        node_alloc::deallocate(allocator, head, 1);
        throw;
    }
    sz = n;
}

List(const List&& l) : sz(l.sz),
t_allocator(t_node_alloc::select_on_container_copy_construction(l.t_allocator)),
allocator(node_alloc::select_on_container_copy_construction(l.allocator))
{
    Node *new_head = node_alloc::allocate(allocator, 1);
    new_head->next = new_head;
    new_head->prev = new_head;

    Node* l_begin = l.head->next;
    size_t i = 0;
    try {
        for (; i < l.sz; ++i) {
            Node* new_node = node_alloc::allocate(allocator, 1);
            node_alloc::construct(allocator, new_node, l_begin->val);
            link_nodes(new_head->prev, new_node);
            l_begin = l_begin->next;
        }
    } catch (...) {
        for (size_t j = 0; j < i; ++j) {
            Node* copy_node = new_head->prev;
            del_node(new_head->prev);
            node_alloc::destroy(allocator, copy_node);
            node_alloc::deallocate(allocator, copy_node, 1);
        }
        node_alloc::deallocate(allocator, new_head, 1);
        throw;
    }
    sz = l.sz;
```

```
    head = node_alloc::allocate(allocator, 1);
    head = new_head;
}

~List() {
    while (sz != 0)
        pop_back();
    node_alloc::deallocate(allocator, head, 1);
}

List& operator=(const List& l) {
    while (sz != 0)
        pop_back();
    if (node_alloc::propagate_on_container_copy_assignment::value) {
        node_alloc::deallocate(allocator, head, 1);
        allocator = l.allocator;
        t_allocator = l.t_allocator;
        head = node_alloc::allocate(allocator, 1);
        head->next = head;
        head->prev = head;
    }
    Node* l_begin = l.head->next;
    for (size_t i = 0; i < l.sz; ++i) {
        push_back(l_begin->val);
        l_begin = l_begin->next;
    }
    return *this;
}
```

Реализация вставки и удаления

```
void push_back(const T &val) { insert(end(), val); }
void push_front(const T &val) { nsert(begin(), val); }
void pop_back() { erase(--end()); }
void pop_front() { erase(begin()); }

template<bool IsConst>
iterator insert(const common_iterator<IsConst> &it, const T &val) {
    Node* new_node = node_alloc::allocate(allocator, 1);
    try {
        node_alloc::construct(allocator, new_node, val);
        link_nodes((it.ptr_node)->prev, new_node);
    } catch (...) {
        Node* copy_node = new_node;
        del_node(new_node);
        node_alloc::destroy(allocator, copy_node);
        node_alloc::deallocate(allocator, copy_node, 1);
        throw;
    }
    ++sz;
    return List::iterator(new_node);
}
template<bool IsConst>
iterator erase(const common_iterator<IsConst> &it) {
    Node* copy_node = const_cast<Node*>(it.ptr_node);
    iterator copy_it(copy_node);
    del_node(it.ptr_node);
    node_alloc::destroy(allocator, it.ptr_node);
    node_alloc::deallocate(allocator, copy_node, 1);
    --sz;
    return ++copy_it;
}
```

Среди методов list - *sort* (так как stl-евская std::sort работает на RA-итераторах, в листе реализована сортировка слиянием), *reverse*, *merge*, *splice* (двух списком целиком или часть одного списка вклеить в другой список)

Инвалидация итераторов

list: Никакой из методов `insert, emplace_front, emplace_back, emplace, push_front, push_back` **не инвалидирует** итераторы и ссылки.

forward_list: Ни одна из перегрузок метода `insert_after` **не инвалидирует** итераторы и ссылки.