Selected files

2 printable files

```
franklist.h
franklist.hpp
```

franklist.h

```cpp
#ifndef FRANKLIST_H
#define FRANKLIST_H

#include <iostream>

namespace vhuk {

template <typename T>
class FrankList;

template <typename T>
std::ostream& operator<<(std::ostream& out, const FrankList<T>& rhv);

template <typename T>
class FrankList {
public:
    using value_type = T;
    using reference = value_type&;
    using const_reference = const value_type&;
    using size_type = std::size_t;
    using pointer = value_type*;
    using const_pointer = const value_type*;
private:
    struct Node
    {
        T val;
        Node* next;
        Node* prev;
        Node* asc;
        Node* desc;
        Node();
        Node(T val);
    };
private:
    class base_iterator
    {
        friend FrankList<value_type>;
    public:
        ~base_iterator();
        bool operator==(const base_iterator& rhv) const; //O(1)
        bool operator!=(const base_iterator& rhv) const; //O(1)
    protected:
        explicit base_iterator(Node* ptr); //O(1)
    protected:
        Node* ptr = nullptr;
    };
public:
    class const_iterator : public base_iterator
    {
        friend FrankList<value_type>;
    public:
        const_iterator(const base_iterator& rhv); //O(1)
        const_iterator(base_iterator&& rhv); //O(1)

        const const_iterator& operator=(const base_iterator& rhv); //O(1)
        const const_iterator& operator=(base_iterator&& rhv); //O(1)
        const_reference operator*() const; //O(1)
        const_pointer operator->() const; //O(1)

        const const_iterator& operator++(); //O(1)
        const const_iterator operator++(int); //O(1)
        const const_iterator& operator--(); //O(1)
        const const_iterator operator--(int); //O(1)

    protected:
        explicit const_iterator(Node* ptr); //O(1)
    };

public:
    class iterator : public const_iterator
    {
        friend FrankList<value_type>;
    public:
        iterator(const base_iterator& rhv); //O(1)
        iterator(base_iterator&& rhv); //O(1)

        reference operator*(); //O(1)
        pointer operator->(); //O(1)

        const iterator& operator=(const base_iterator& rhv); //O(1)
        const iterator& operator=(base_iterator&& rhv); //O(1)
    protected:
        explicit iterator(Node* ptr); //O(1)
    };

public:
    class const_reverse_iterator : public base_iterator
    {
        friend FrankList<value_type>;
    public:
        const_reverse_iterator(const base_iterator& rhv); //O(1)
        const_reverse_iterator(base_iterator&& rhv); //O(1)

        const const_reverse_iterator& operator=(const base_iterator& rhv); //O(1)
        const const_reverse_iterator& operator=(base_iterator&& rhv); //O(1)
        const_reference operator*() const; //O(1)
        const_pointer operator->() const; //O(1)

        const const_reverse_iterator& operator++(); //O(1)
        const const_reverse_iterator operator++(int); //O(1)
        const const_reverse_iterator& operator--(); //O(1)
        const const_reverse_iterator operator--(int); //O(1)

    protected:
        explicit const_reverse_iterator(Node* ptr); //O(1)
    };
```

```cpp
public:
    class reverse_iterator : public const_reverse_iterator
    {
        friend FrankList<value_type>;
    public:
        reverse_iterator(const base_iterator& rhv); //O(1)
        reverse_iterator(base_iterator&& rhv); //O(1)

        reference operator*(); //O(1)
        pointer operator->(); //O(1)

        const reverse_iterator& operator=(const base_iterator& rhv); //O(1)
        const reverse_iterator& operator=(base_iterator&& rhv); //O(1)

    protected:
        explicit reverse_iterator(Node* ptr); //O(1)
    };
public:
    class const_asc_iterator :public base_iterator
    {
        friend FrankList<value_type>;
    public:
        const_asc_iterator(const base_iterator& rhv); //O(1)
        const_asc_iterator(base_iterator&& rhv); //O(1)

        const const_asc_iterator& operator=(const base_iterator& rhv); //O(1)
        const const_asc_iterator& operator=(base_iterator&& rhv); //O(1)
        const_reference operator*() const; //O(1)
        const_pointer operator->() const; //O(1)

        const const_asc_iterator& operator++(); //O(1)
        const const_asc_iterator operator++(int); //O(1)
        const const_asc_iterator& operator--(); //O(1)
        const const_asc_iterator operator--(int); //O(1)

    protected:
        explicit const_asc_iterator(Node* ptr); //O(1)
    };
public:
    class asc_iterator : public const_asc_iterator
    {
        friend FrankList<value_type>;
    public:
        asc_iterator(const base_iterator& rhv); //O(1)
        asc_iterator(base_iterator&& rhv); //O(1)

        reference operator*(); //O(1)
        pointer operator->(); //O(1)

        const asc_iterator& operator=(const base_iterator& rhv); //O(1)
        const asc_iterator& operator=(base_iterator&& rhv); //O(1)

    protected:
        explicit asc_iterator(Node* ptr); //O(1)
    };
public:
    class const_desc_iterator : public base_iterator
    {
        friend FrankList<value_type>;
    public:
        const_desc_iterator(const base_iterator& rhv); //O(1)
        const_desc_iterator(base_iterator&& rhv); //O(1)

        const const_desc_iterator& operator=(const base_iterator& rhv); //O(1)
        const const_desc_iterator& operator=(base_iterator&& rhv); //O(1)
        const_reference operator*() const; //O(1)
        const_pointer operator->() const; //O(1)

        const const_desc_iterator& operator++(); //O(1)
        const const_desc_iterator operator++(int); //O(1)
        const const_desc_iterator& operator--(); //O(1)
        const const_desc_iterator operator--(int); //O(1)

    protected:
        explicit const_desc_iterator(Node* ptr); //O(1)
    };
public:
    class desc_iterator : public const_desc_iterator
    {
        friend FrankList<value_type>;
    public:
        desc_iterator(const base_iterator& rhv); //O(1)
        desc_iterator(base_iterator&& rhv); //O(1)

        reference operator*(); //O(1)
        pointer operator->(); //O(1)

        const desc_iterator& operator=(const base_iterator& rhv); //O(1)
        const desc_iterator& operator=(base_iterator&& rhv); //O(1)

    protected:
        explicit desc_iterator(Node* ptr); //O(1)
    };
public:
    class const_multi_iterator : public base_iterator
    {
        friend FrankList<value_type>;
    public:
        const_multi_iterator(const base_iterator& rhv); //O(1)
        const_multi_iterator(base_iterator&& rhv); //O(1)

        const const_multi_iterator& operator=(const base_iterator& rhv); //O(1)
        const const_multi_iterator& operator=(base_iterator&& rhv); //O(1)
        const_reference operator*() const; //O(1)
        const_pointer operator->() const; //O(1)

        const const_multi_iterator& operator++(); //O(1)
        const const_multi_iterator operator++(int); //O(1)
        const const_multi_iterator& operator--(); //O(1)
        const const_multi_iterator operator--(int); //O(1)

        void chmod(); //O(1)
    protected:
        explicit const_multi_iterator(Node* ptr); //O(1)
        bool mode = true;
    };
public:
    class multi_iterator : public const_multi_iterator
    {
        friend FrankList<value_type>;
    public:
        multi_iterator(const base_iterator& rhv); //O(1)
        multi_iterator(base_iterator&& rhv); //O(1)

        reference operator*(); //O(1)
        pointer operator->(); //O(1)

        const multi_iterator& operator=(const base_iterator& rhv); //O(1)
        const multi_iterator& operator=(base_iterator&& rhv); //O(1)

    protected:
        explicit multi_iterator(Node* ptr); //O(1)
    };
public:
    class const_multi_reverse_iterator : public base_iterator
    {
        friend FrankList<value_type>;
    public:
        const_multi_reverse_iterator(const base_iterator& rhv); //O(1)
        const_multi_reverse_iterator(base_iterator&& rhv); //O(1)

        const const_multi_reverse_iterator& operator=(const base_iterator& rhv); //O(1)
        const const_multi_reverse_iterator& operator=(base_iterator&& rhv); //O(1)
        const_reference operator*() const; //O(1)
        const_pointer operator->() const; //O(1)

        const const_multi_reverse_iterator& operator++(); //O(1)
        const const_multi_reverse_iterator operator++(int); //O(1)
        const const_multi_reverse_iterator& operator--(); //O(1)
        const const_multi_reverse_iterator operator--(int); //O(1)


        void chmod(); //O(1)
    protected:
        explicit const_multi_reverse_iterator(Node* ptr); //O(1)
        bool mode = true;
    };
public:
    class multi_reverse_iterator : public const_multi_reverse_iterator
    {
        friend FrankList<value_type>;
    public:
        multi_reverse_iterator(const base_iterator& rhv); //O(1)
        multi_reverse_iterator(base_iterator&& rhv); //O(1)

        reference operator*(); //O(1)
        pointer operator->(); //O(1)

        const multi_reverse_iterator& operator=(const base_iterator& rhv); //O(1)
        const multi_reverse_iterator& operator=(base_iterator&& rhv); //O(1)

    protected:
        explicit multi_reverse_iterator(Node* ptr); //O(1)
    };

public:
    FrankList(); //O(1)
    FrankList(size_type size); //O(n)
    FrankList(size_type size, const_reference init); //O(n)
    FrankList(const FrankList<value_type>& rhv); //O(n)
    FrankList(FrankList<value_type>&& rhv); //O(1)
    FrankList(std::initializer_list<value_type> init); //O(n)
    template <typename input_iterator>
    FrankList(input_iterator f, input_iterator l); //O(n)
    ~FrankList();

public:
    void swap(FrankList<value_type>& rhv); //O(1)

    size_type size() const; //O(n)

    bool empty() const; //O(1)
    void resize(size_type s, const_reference init = value_type()); //O(n)
    void clear() noexcept; //O(n)

    void push_front(const_reference elem); //~O(1)
    void pop_front(); //O(1)
    void push_back(const_reference elem); //~O(1)
    void pop_back(); //O(1)

    const_reference front() const; //O(1)
    reference front(); //O(1)
    const_reference back() const; //O(1)
    reference back(); //O(1)
    const_reference min() const; //O(1)
    reference min(); //O(1)
    const_reference max() const; //O(1)
    reference max(); //O(1)

    const FrankList<value_type>& operator=(const FrankList<value_type>& rhv); //O(n)
    const FrankList<value_type>& operator=(FrankList<value_type>&& rhv); //O(n)
    const FrankList<value_type>& operator=(std::initializer_list<value_type> init); //O(n)

    bool operator==(const FrankList<value_type>& rhv) const; //O(n)
    bool operator!=(const FrankList<value_type>& rhv) const; //O(n)
    bool operator<(const FrankList<value_type>& rhv) const; //O(n)
    bool operator<=(const FrankList<value_type>& rhv) const; //O(n)
    bool operator>(const FrankList<value_type>& rhv) const; //O(n)
    bool operator>=(const FrankList<value_type>& rhv) const; //O(n)

public:
    const_iterator cbegin() const; //O(1)
    const_iterator cend() const; //O(1)
    const_reverse_iterator crbegin() const; //O(1)
    const_reverse_iterator crend() const; //O(1)
    const_asc_iterator cabegin() const; //O(1)
    const_asc_iterator caend() const; //O(1)
    const_desc_iterator cdbegin() const; //O(1)
    const_desc_iterator cdend() const; //O(1)
    const_multi_iterator cmbegin() const; //O(1)
    const_multi_iterator cmend() const; //O(1)
    const_multi_iterator cmabegin() const; //O(1)
    const_multi_iterator cmaend() const; //O(1)
    const_multi_reverse_iterator cmrbegin() const; //O(1)
    const_multi_reverse_iterator cmrend() const; //O(1)
    const_multi_reverse_iterator cmrdbegin() const; //O(1)
    const_multi_reverse_iterator cmrdend() const; //O(1)

    iterator begin(); //O(1)
    iterator end(); //O(1)
    reverse_iterator rbegin(); //O(1)
    reverse_iterator rend(); //O(1)
    asc_iterator abegin(); //O(1)
    asc_iterator aend(); //O(1)
    desc_iterator dbegin(); //O(1)
    desc_iterator dend(); //O(1)
    multi_iterator mbegin(); //O(1)
    multi_iterator mend(); //O(1)
    multi_iterator mabegin(); //O(1)
    multi_iterator maend(); //O(1)
    multi_reverse_iterator mrbegin(); //O(1)
    multi_reverse_iterator mrend(); //O(1)
    multi_reverse_iterator mrdbegin(); //O(1)
    multi_reverse_iterator mrdend(); //O(1)

public:
    template <typename iter>
    typename std::enable_if<std::is_base_of<const_iterator, iter>::value ||
                              std::is_base_of<const_asc_iterator, iter>::value ||
                              std::is_base_of<const_multi_iterator, iter>::value,
             iter>::type
    insert(iter pos, const_reference val) { //O(1)
        return insert_def(pos, val);
    }

    template <typename iter>
    typename std::enable_if<std::is_base_of<const_reverse_iterator, iter>::value ||
                              std::is_base_of<const_desc_iterator, iter>::value ||
                              std::is_base_of<const_multi_reverse_iterator, iter>::value,
             iter>::type
    insert(iter pos, const_reference val) { //O(1)
        return insert_rev(pos, val);
    }

    template <typename iter>
    iter insert(iter pos, size_type size, const_reference val); //O(n)
    template <typename iter>
    iter insert(iter pos, std::initializer_list<value_type> init); //O(n)
    template <typename iter, typename input_iterator>
    iter insert(iter pos, input_iterator f, input_iterator l); //O(n)

    template <typename iter>
    iter erase(iter pos); //O(1)
    template <typename iter>
    iter erase(iter f, iter l); //O(n)

    size_type remove(const_reference val); //O(n)
    template <typename unary_predicate>
    size_type remove_if(unary_predicate func); //O(n)

    void reverse(); //O(n)
    void sort(bool reversed = false); //O(n)

    iterator find(const_reference elem); //O(n)
    iterator rfind(const_reference elem); //O(n)

    template <typename unary_predicate>
    void traverse(unary_predicate func, bool sorted = false, bool reversed = false); //O(n)

    void print(bool sorted = false, bool reversed = false); //O(n)

protected:
    void put_in_sorted_order(Node* ptr); //O(n)
    void organize_left(Node* ptr); //O(1)
    void organize_right(Node* ptr); //O(1)
private:
    template <typename iter>
    iter insert_def(iter pos, const_reference val); //O(1)

    template <typename iter>
    iter insert_rev(iter pos, const_reference val); //O(1)

private:
    Node* head;
    Node* tail;
    Node* ahead;
    Node* atail;
};



}

#include "franklist.hpp"


#endif // _FRANKLIST_H__

```

franklist.hpp

```cpp
#ifndef FRANKLIST_HPP
#define FRANKLIST_HPP
#include "franklist.h"


namespace vhuk {


template <typename T>
std::ostream& operator<<(std::ostream& out, const FrankList<T>& rhv) {
    for (auto it = rhv.cbegin(); it != rhv.cend(); ++it) {
        out << *it << " ";
    }
    return out;
}

template <typename T>
FrankList<T>::Node::Node() : val(), next(nullptr), prev(nullptr), asc(nullptr), 

template <typename T>
FrankList<T>::Node::Node(T val) : val(val), next(nullptr), prev(nullptr), asc(nu

template <typename T>
FrankList<T>::FrankList() : head(nullptr), tail(nullptr), ahead(nullptr), atail(

template <typename T>
FrankList<T>::FrankList(std::size_t size) : FrankList() {
    for (std::size_t i = 0; i < size; ++i) {
        push_back(T());
    }
}

template <typename T>
FrankList<T>::FrankList(std::size_t size, const_reference init) : FrankList() {
    for (std::size_t i = 0; i < size; ++i) {
        push_back(init);
    }
}

template <typename T>
FrankList<T>::FrankList(const FrankList<T>& rhv) : FrankList() {
    for (auto it = rhv.begin(); it != rhv.end(); ++it) {
        push_back(*it);
    }
}

template <typename T>
FrankList<T>::FrankList(FrankList<T>&& rhv) : head(rhv.head), tail(rhv.tail), ah
{
   rhv.head = nullptr;
   rhv.tail = nullptr;
   rhv.ahead = nullptr;
   rhv.atail = nullptr;
}

template <typename T>
FrankList<T>::FrankList(std::initializer_list<value_type> init) : FrankList() {
   for (const auto& i : init){
      push_back(i);
    }
}
template <typename T>
template <typename input_iterator>
FrankList<T>::FrankList(input_iterator f, input_iterator l) : FrankList() {
   for (input_iterator it = f; it != l; ++it){
      push_back(*it);
   }
}

template <typename T>
FrankList<T>::~FrankList() {
   clear();
}

template <typename T>
void FrankList<T>::swap(FrankList<T>& rhv) {
   std::swap(head, rhv.head);
   std::swap(tail, rhv.tail);
   std::swap(ahead, rhv.ahead);
   std::swap (atail, rhv.atail);
}

template <typename T>
typename FrankList<T>::size_type FrankList<T>::size() const {
   size_type size = 0;
   Node* i = head;
   while (i != nullptr){
      ++size;
      i = i -> next;
   }

   return size;
}

template <typename T>
bool FrankList<T>::empty() const {
   return head == nullptr;
}

template <typename T>
void FrankList<T>::resize(size_type s, const_reference init){
   size_type size = this -> size();
   if (size > s){
      for (size_type i = s; i < size; ++i){
         pop_back();
      }
   } else if (size < s) {
      for (size_type i = size; i < s; ++i) {
         push_back(init);
      }
   }
}

template <typename T>
void FrankList<T>::clear () noexcept {
   Node* i = head;
   while (i != nullptr) {
      Node* next = i->next;
      delete i;
      i = next;
   }

   head = nullptr;
   tail = nullptr;
   ahead = nullptr;
   atail = nullptr;
}

template <typename T>
void FrankList<T>::push_front(const_reference elem) {
   Node* ptr = new Node(elem);
   if (head != nullptr) {
      head -> prev = ptr;
   } else {
      tail = ptr;
   }

   ptr -> next = head;
   head = ptr;
   put_in_sorted_order(ptr);
}

template <typename T>
void FrankList<T>::pop_front() {
   if (head == nullptr) {
      return;
   }

   if (head -> next != nullptr) {
      head -> next -> prev = nullptr;
      if (head -> asc && head -> desc){
         head -> desc -> asc = head -> asc;
         head -> asc -> desc = head -> desc;
      }
      Node* ptr = head -> next;
      delete head;
      head = ptr;
   } else {
      delete head;
      head = nullptr;
      tail = nullptr;
   }
}


template <typename T>
void FrankList<T>::push_back(const_reference elem) {
   Node* ptr = new Node (elem);
   if (head == nullptr) {
      head = ptr;
      tail = ptr;
      return;
    } else {
      tail -> next = ptr;
      ptr -> prev = tail;
      tail = ptr;
   }

      put_in_sorted_order(ptr);
}

template <typename T>
void FrankList<T>::pop_back() {
   if (tail == nullptr) {
      return;
   }

   if (tail -> prev != nullptr){
      tail -> prev -> next = nullptr;
      if (tail -> desc && tail -> asc){
         tail -> asc -> desc = tail -> desc;
         tail -> desc -> asc = tail -> asc;
       }
      Node* ptr = tail;
      tail = tail -> prev;
      delete ptr;
   } else {
      delete tail;
      tail = nullptr;
      head = nullptr;
   }
}

template <typename T>
typename FrankList<T>::const_reference FrankList<T>::front() const {
   if (head == nullptr){
      throw std::out_of_range("Error");
   }

   return head -> val;
}

template <typename T>
typename FrankList<T>::reference FrankList<T>::front() {
   if (head == nullptr){
      throw std::out_of_range("Error");
   }

   return head -> val;
}

template <typename T>
typename FrankList<T>::const_reference FrankList<T>::back() const {
   if (tail == nullptr){
      throw std::out_of_range("Error");
   }

   return tail -> val;
}

template <typename T>
typename FrankList<T>::reference FrankList<T>::back() {
   if (tail == nullptr){
      throw std::out_of_range("Error");
   }

   return tail -> val;
}

template <typename T>
typename FrankList<T>::const_reference FrankList<T>::min() const {
   if (ahead == nullptr){
      throw std::out_of_range("Error");
   }

   return ahead -> val;
}

```

```cpp
template <typename T>
typename FrankList<T>::reference FrankList<T>::min() {
   if (ahead == nullptr){
      throw std::out_of_range("Error");
   }

   return ahead -> val;
}

template <typename T>
typename FrankList<T>::const_reference FrankList<T>::max() const {
   if (atail == nullptr){
      throw std::out_of_range("Error");
   }

   return atail -> val;
}

template <typename T>
typename FrankList<T>::reference FrankList<T>::max() {
   if (atail == nullptr){
      throw std::out_of_range("Error");
   }

   return atail -> val;
}

template <typename T>
const FrankList<T>& FrankList<T>::operator=(const FrankList<T>& rhv) {
    if (this != &rhv) {
        clear();
        for (auto it = rhv.cbegin(); it != rhv.cend(); ++it) {
            push_back(*it);
        }
    }
    return *this;
}

template <typename T>
const FrankList<T>& FrankList<T>::operator=(FrankList<T>&& rhv) {
   if (this != &rhv){
      clear();
      head = rhv.head;
      tail = rhv.tail;
      ahead = rhv.ahead;
      atail = rhv.atail;
      rhv.head = nullptr;
      rhv.tail = nullptr;
      rhv.ahead = nullptr;
      rhv.atail = nullptr;
   }

   return *this;
}

template <typename T>
const FrankList<T>& FrankList<T>::operator=(std::initializer_list<value_type> in
   clear();
   for(const auto& i : init){
      push_back(i);
   }

   return *this;
}


template <typename T>
bool FrankList<T>::operator==(const FrankList<value_type>& rhv) const {
   if (size() != rhv.size()){
      return false;
   }

   auto i = cbegin();
   auto j = rhv.cbegin();

   while (i != cend() && j != rhv.cend()){
      if(*i != *j){
         return false;
      }

      ++i;
      ++j;
   }

   return (i == cend() && j == rhv.cend());
}


template <typename T>
bool FrankList<T>::operator!=(const FrankList<value_type>& rhv) const {
   return !(*this == rhv);
}

template <typename T>
bool FrankList<T>::operator<(const FrankList<value_type>& rhv) const {
   size_type size1 = size();
   size_type size2 = rhv.size();
   if (size1 < size2){
      return true;
   } else if (size1 > size2){
      return false;
   }

   auto i = cbegin();
   auto j = rhv.cbegin();

   while (i != cend() && j != rhv.cend()){
      if (*i < *j){
         return true;
      } else if (*i > *j){
         return false;
      }

      ++i;
      ++j;
   }

   return !(i == cend() && j == rhv.cend());
}

template <typename T>
bool FrankList<T>::operator<=(const FrankList<value_type>& rhv) const {
   return (*this == rhv || *this < rhv);
}

template <typename T>
bool FrankList<T>::operator>(const FrankList<value_type>& rhv) const {
   return !(*this <= rhv);
}

template <typename T>
bool FrankList<T>::operator>=(const FrankList<value_type>& rhv) const {
   return (*this > rhv || *this == rhv);
}

template <typename T>
typename FrankList<T>::const_iterator FrankList<T>::cbegin() const {
   return const_iterator(head);
}

template <typename T>
typename FrankList<T>::const_iterator FrankList<T>::cend() const {
   return const_iterator(nullptr);
}

template <typename T>
typename FrankList<T>::const_reverse_iterator FrankList<T>::crbegin() const {
   return const_reverse_iterator(tail);
}

template <typename T>
typename FrankList<T>::const_reverse_iterator FrankList<T>::crend() const {
   return const_reverse_iterator(nullptr);
}

template <typename T>
typename FrankList<T>::const_asc_iterator FrankList<T>::cabegin() const {
   return const_asc_iterator(ahead);
}

template <typename T>
typename FrankList<T>::const_asc_iterator FrankList<T>::caend() const {
   return const_asc_iterator(nullptr);
}

template <typename T>
typename FrankList<T>::const_desc_iterator FrankList<T>::cdbegin() const {
   return const_desc_iterator(atail);
}

template <typename T>
typename FrankList<T>::const_desc_iterator FrankList<T>::cdend() const {
   return const_desc_iterator(nullptr);
}

template <typename T>
typename FrankList<T>::const_multi_iterator FrankList<T>::cmbegin() const {
   return const_multi_iterator(head);
}

template <typename T>
typename FrankList<T>::const_multi_iterator FrankList<T>::cmend() const {
   return const_multi_iterator(nullptr);
}

template <typename T>
typename FrankList<T>::const_multi_iterator FrankList<T>::cmabegin() const {
   return const_multi_iterator(ahead);
}

template <typename T>
typename FrankList<T>::const_multi_iterator FrankList<T>::cmaend() const {
   return const_multi_iterator(nullptr);
}

template <typename T>
typename FrankList<T>::const_multi_reverse_iterator FrankList<T>::cmrbegin() con
   return const_multi_reverse_iterator(tail);
}

template <typename T>
typename FrankList<T>::const_multi_reverse_iterator FrankList<T>::cmrend() const
   return const_multi_reverse_iterator(nullptr);
}

template <typename T>
typename FrankList<T>::const_multi_reverse_iterator FrankList<T>::cmrdbegin() co
   return const_multi_reverse_iterator(atail);
}

template <typename T>
typename FrankList<T>::const_multi_reverse_iterator FrankList<T>::cmrdend() cons
   return const_multi_reverse_iterator(nullptr);
}

template <typename T>
typename FrankList<T>::iterator FrankList<T>::begin() {
   return iterator(head);
}

template <typename T>
typename FrankList<T>::iterator FrankList<T>::end() {
   return iterator(nullptr);
}

template <typename T>
typename FrankList<T>::reverse_iterator FrankList<T>::rbegin() {
   return reverse_iterator(tail);
}

template <typename T>
typename FrankList<T>::reverse_iterator FrankList<T>::rend() {
   return reverse_iterator(nullptr);
}

template <typename T>
```

```cpp
typename FrankList<T>::asc_iterator FrankList<T>::abegin() {
   return asc_iterator(ahead);
}

template <typename T>
typename FrankList<T>::asc_iterator FrankList<T>::aend() {
   return asc_iterator(nullptr);
}

template <typename T>
typename FrankList<T>::desc_iterator FrankList<T>::dbegin() {
   return desc_iterator(atail);
}

template <typename T>
typename FrankList<T>::desc_iterator FrankList<T>::dend() {
   return desc_iterator(nullptr);
}

template <typename T>
typename FrankList<T>::multi_iterator FrankList<T>::mbegin() {
   return multi_iterator(head);
}

template <typename T>
typename FrankList<T>::multi_iterator FrankList<T>::mend() {
   return multi_iterator(nullptr);
}

template <typename T>
typename FrankList<T>::multi_iterator FrankList<T>::mabegin() {
   return multi_iterator(ahead);
}

template <typename T>
typename FrankList<T>::multi_iterator FrankList<T>::maend() {
   return multi_iterator(nullptr);
}

template <typename T>
typename FrankList<T>::multi_reverse_iterator FrankList<T>::mrbegin() {
   return multi_reverse_iterator(tail);
}

template <typename T>
typename FrankList<T>::multi_reverse_iterator FrankList<T>::mrend() {
   return multi_reverse_iterator(nullptr);
}

template <typename T>
typename FrankList<T>::multi_reverse_iterator FrankList<T>::mrdbegin() {
   return multi_reverse_iterator(atail);
}

template <typename T>
typename FrankList<T>::multi_reverse_iterator FrankList<T>::mrdend() {
   return multi_reverse_iterator(nullptr);
}


template <typename T>
template <typename iter>
iter FrankList<T>::insert(iter pos, size_type size, const_reference val){
   if (empty() && pos != begin()){
      throw std::invalid_argument("Error");
   }

   if (pos == begin()){
      for(size_type s = 0; s < size; ++s){
         push_front(val);
      }
   } else if (pos == end()){
      for(size_type s = 0; s < size; ++s){
         push_back(val);
      }
   } else {
      for(size_type s = 0; s < size; ++s){
         pos = insert_def(pos, val);
      }
   }

   return pos;
}

template <typename T>
template <typename iter>
iter FrankList<T>::insert(iter pos, std::initializer_list<value_type> init){
   if (empty() && pos != begin()){
      throw std::invalid_argument("Error");
   }

   if (pos == begin()){
      for(const auto& i : init){
         push_front(i);
      }
   } else if (pos == end()){
      for(const auto& i : init){
         push_back(i);
      }
   } else {
      for(const auto& i : init){
         pos = insert_def(pos, i);
      }
   }

   return pos;
}

template <typename T>
template <typename iter, typename input_iterator>
iter FrankList<T>::insert(iter pos, input_iterator f, input_iterator l){
   if (empty() && pos != begin()){
      throw std::invalid_argument("Error");
   }

   if (f == l){
      return pos;
   }
```

```cpp

    if (pos == begin()){
      for(input_iterator it = f; it != l; ++it){
         push_front(*it);
      }
   } else if (pos == end()){
      for(input_iterator it = f; it != l; ++it){
         push_back(*it);
      }
   } else {
      for(input_iterator it = f; it != l; ++it){
         pos = insert_def(pos, *it);
      }
   }

   return pos;
}

template <typename T>
template <typename iter>
iter FrankList<T>::erase(iter pos){
   if (empty()){
      throw std::invalid_argument("Error");
   }

    Node* ptr = pos.ptr;
    iter next(pos);
    ++next;

    if (pos == begin()) {
        pop_front();
        return next;
    } else if (pos == iter(tail)) {
        pop_back();
        return next;
    }

    if (ptr->prev && ptr->next) {
        ptr->prev->next = ptr->next;
        ptr->next->prev = ptr->prev;
    }

    if (ptr->desc && ptr->asc) {
        ptr->desc->asc = ptr->asc;
        ptr->asc->desc = ptr->desc;
    }

    delete ptr;
    return next;
}


template <typename T>
template <typename iter>
iter FrankList<T>::erase(iter f, iter l){
      iter it = f;
      while (it != l){
         it = erase(it);
      }

      return it;
}

template <typename T>
typename FrankList<T>::size_type FrankList<T>::remove(const_reference val){
   size_type count = 0;
   iterator it = begin();
   while (it != end()){
      if (it != end() && *it == val){
         std::cout << *it << std::endl;
         it = erase(it);
         ++count;
         if (it == end()){
            break;
         }
      } else {
         ++it;
      }
   }

   return count;
}

template <typename T>
template <typename unary_predicate>
typename FrankList<T>::size_type FrankList<T>::remove_if(unary_predicate func){
   size_type count = 0;
   iterator it = begin();
   while (it != end()){
      if(func(*it)){
         it = erase(it);
         ++count;
      } else {
         ++it;
      }
   }

   return count;
}

template <typename T>
void FrankList<T>::reverse() {
    std::swap(head, tail);
    Node* it = head;
    while (it != nullptr) {
        std::swap(it->next, it->prev);
        it = it->next;
    }
}

template <typename T>
void FrankList<T>::sort(bool reversed) {
    if (!ahead) {
        return;
    }
   if (!reversed){
   Node* tmp = ahead;
   while (tmp != nullptr){
      tmp -> next = tmp -> asc;
      tmp -> prev = tmp -> desc;
      tmp = tmp -> next;
   }

   head = ahead;
   tail = atail;
   } else {
   Node* tmp = atail;
   while (tmp != nullptr){
      tmp -> next = tmp -> desc;
      tmp -> prev = tmp -> asc;
      tmp = tmp -> next;
   }

   head = atail;
   tail = ahead;
 }
}

template <typename T>
typename FrankList<T>::iterator FrankList<T>::find(const_reference elem){
   iterator it = begin();
   while (it != end()){
      if (*it != elem){
         ++it;
      } else {
         break;
      }
   }


   organize_left(it.ptr);
   return it;
}

template <typename T>
typename FrankList<T>::iterator FrankList<T>::rfind(const_reference elem){
   const_reverse_iterator it = crbegin();
   while (it != crend()){
      if(*it != elem){
         ++it;
      } else {
         break;
      }
   }

   organize_right(it.ptr);
   return it;
}


template <typename T>
template <typename unary_predicate>
void FrankList<T>::traverse(unary_predicate func, bool sorted, bool reversed){
    if (!head) {
        return;
    }
    if (!sorted && !reversed) {
        for (auto it = begin(); it != end(); ++it) {
            func(*it);
        }
    }
    else if (!sorted) {
        for (auto it = rbegin(); it != rend(); ++it) {
            func(*it);
        }
    }
    else if (!reversed) {
        for (auto it = abegin(); it != aend(); ++it) {
            func(*it);
        }
    }
    else {
        for (auto it = dbegin(); it != dend(); ++it) {
            func(*it);
        }
    }
}

template <typename T>
void FrankList<T>::print(bool sorted, bool reversed) {
   if(!head){
      std::cout << std::endl;
   }

   if (!sorted && !reversed) {
        for (auto it = begin(); it != end(); ++it) {
            std::cout << *it << ' ';
        }
    }
    else if (!sorted) {
        for (auto it = rbegin(); it != rend(); ++it) {
            std::cout << *it << ' ';
        }
    }
    else if (!reversed) {
        for (auto it = abegin(); it != aend(); ++it) {
            std::cout << *it << ' ';
        }
    }
    else {
        for (auto it = dbegin(); it != dend(); ++it) {
            std::cout << *it << ' ';
        }
    }
}


template <typename T>
void FrankList<T>::put_in_sorted_order(Node* ptr) {
   if (!ahead){
      ahead = ptr;
      atail = ptr;
      return;
   }

   Node* tmp = ahead;
   while (tmp -> asc && tmp -> val <= ptr -> val){
      tmp = tmp -> asc;
   }

   if (tmp && tmp->val < ptr->val) {
        ptr->asc = tmp->asc;
        if (tmp->asc) {
            tmp->asc->desc = ptr;
        } else {
            atail = ptr;
        }
        tmp->asc = ptr;
        ptr->desc = tmp;
    } else {
        ptr->desc = tmp->desc;
        if (tmp->desc) {
            tmp->desc->asc = ptr;
        } else {
            ahead = ptr;
        }
        ptr->asc = tmp;
        tmp->desc = ptr;
    }
}


template <typename T>
void FrankList<T>::organize_left(Node* ptr) {
   if (ptr == nullptr || ptr == head){
      return;
   }


   Node* node = ptr -> prev;
   node -> prev -> next = ptr;
   ptr -> prev = node -> prev;
   node -> prev = ptr;
   node -> next = ptr -> next;
   node -> next -> prev = node;
   ptr -> next = node;

   if (node == head){
      head = ptr;
   }

   if (ptr == tail){
      tail = node;
   }
}


template <typename T>
void FrankList<T>::organize_right(Node* ptr) {
   if (ptr == nullptr || ptr == tail){
      return;
   }


   Node* nodeP = ptr -> prev;
   Node* nodeN = ptr -> next;

   nodeP -> next = nodeN;
   nodeN -> prev = nodeP;
   ptr -> next = nodeN -> next;
   ptr -> prev = nodeN;
   nodeN -> next -> prev = ptr;
   nodeN -> next = ptr;

   if (ptr == head){
      head = nodeN;
   }

   if (nodeN == tail){
      tail = ptr;
   }
}

template <typename T>
template <typename iter>
iter FrankList<T>::insert_def(iter pos, const_reference val) {
    Node* ptr = new Node(val);
    Node* node = pos.ptr;

    if (node == nullptr) {
        if (!tail) {
            head = tail = ptr;
        } else {
            tail->next = ptr;
            ptr->prev = tail;
            tail = ptr;
        }
    } else {
        ptr->next = node;
        ptr->prev = node->prev;
        if (node->prev) {
            node->prev->next = ptr;
        } else {
            head = ptr;
        }
        node->prev = ptr;
    }
    put_in_sorted_order(ptr);
    return iter(ptr);
}

template <typename T>
template <typename iter>
iter FrankList<T>::insert_rev(iter pos, const_reference val) {
    Node* ptr = new Node(val);
    Node* node = pos.ptr;

    if (node == nullptr) {
        if (!tail) {
            head = tail = ptr;
        } else {
            tail->next = ptr;
            ptr->prev = tail;
            tail = ptr;
        }
    } else {
        ptr->prev = node;
        ptr->next = node->next;
        if (node->next) {
            node->next->prev = ptr;
        } else {
            tail = ptr;
        }
        node->next = ptr;
    }
    put_in_sorted_order(ptr);
    return iter(ptr);
}



////////////////////////////////////////////////////////////////////////////////

template <typename T>
FrankList<T>::base_iterator::~base_iterator() {
   ptr = nullptr;
}

template <typename T>
bool FrankList<T>::base_iterator::operator==(const base_iterator& rhv) const {
   return this -> ptr == rhv.ptr;
}

template <typename T>
bool FrankList<T>::base_iterator::operator!=(const base_iterator& rhv) const {
   return !(this->ptr == rhv.ptr);
}

template <typename T>
FrankList<T>::base_iterator::base_iterator(Node* ptr) : ptr(ptr) {}

template <typename T>
FrankList<T>::const_iterator::const_iterator(const base_iterator& rhv) : base_it

template <typename T>
FrankList<T>::const_iterator::const_iterator(base_iterator&& rhv) : base_iterato
   rhv.ptr = nullptr;
}

template <typename T>
const typename FrankList<T>::const_iterator& FrankList<T>::const_iterator::opera
   this -> ptr = rhv.ptr;
   return *this;
}

template <typename T>
const typename FrankList<T>::const_iterator& FrankList<T>::const_iterator::opera
   if (this == &rhv){
      return *this;
   }

   this -> ptr = rhv.ptr;
   rhv.ptr = nullptr;
   return *this;
}

template <typename T>
typename FrankList<T>::const_reference FrankList<T>::const_iterator::operator*()
   return this -> ptr -> val;
}

template <typename T>
typename FrankList<T>::const_pointer FrankList<T>::const_iterator::operator-> ()
   return this -> ptr;
}

template <typename T>
const typename FrankList<T>::const_iterator& FrankList<T>::const_iterator::opera
    this -> ptr = this -> ptr -> next;
    return *this;
}

template <typename T>
const typename FrankList<T>::const_iterator FrankList<T>::const_iterator::operat
   const_iterator tmp = *this;
   ++(*this);
   return tmp;
}

template <typename T>
const typename FrankList<T>::const_iterator& FrankList<T>::const_iterator::opera
    this -> ptr = this -> ptr -> prev;
    return *this;
}

template <typename T>
const typename FrankList<T>::const_iterator FrankList<T>::const_iterator::operat
   const_iterator tmp = *this;
   --(*this);
   return tmp;
}

template <typename T>
FrankList<T>::const_iterator::const_iterator(Node* ptr) : base_iterator(ptr) {}

template <typename T>
FrankList<T>::iterator::iterator(const base_iterator& rhv) : const_iterator(rhv.

template <typename T>
FrankList<T>::iterator::iterator(base_iterator&& rhv) : const_iterator(rhv.ptr)
   rhv.ptr = nullptr;
}

template <typename T>
typename FrankList<T>::reference FrankList<T>::iterator::operator*() {
   return this -> ptr -> val;
}

template <typename T>
typename FrankList<T>::pointer FrankList<T>::iterator::operator-> () {
   return this -> ptr;
}
```

```cpp

template <typename T>
const typename FrankList<T>::iterator& FrankList<T>::iterator::operator=(const b
   this -> ptr = rhv.ptr;
   return *this;
}

template <typename T>
const typename FrankList<T>::iterator& FrankList<T>::iterator::operator=(base_it
   if (this == &rhv){
      return *this;
   }

   this -> ptr = rhv.ptr;
   rhv.ptr = nullptr;
   return *this;
}

template <typename T>
FrankList<T>::iterator::iterator(Node* ptr) : const_iterator(ptr) {}

template <typename T>
FrankList<T>::const_reverse_iterator::const_reverse_iterator(const base_iterator

template <typename T>
FrankList<T>::const_reverse_iterator::const_reverse_iterator(base_iterator&& rhv
   rhv.ptr = nullptr;
}

template <typename T>
const typename FrankList<T>::const_reverse_iterator& FrankList<T>::const_reverse
base_iterator& rhv){
   this -> ptr = rhv.ptr;
   return *this;
}

template <typename T>
const typename FrankList<T>::const_reverse_iterator& FrankList<T>::const_reverse
(base_iterator&& rhv) {
   if (this = &rhv){
      return *this;
   }

   this -> ptr = rhv.ptr;
   rhv.ptr = nullptr;
   return *this;
}

template <typename T>
typename FrankList<T>::const_reference FrankList<T>::const_reverse_iterator::ope
    return this->ptr->val;
}

template <typename T>
typename FrankList<T>::const_pointer FrankList<T>::const_reverse_iterator::opera
    return this->ptr;
}

template <typename T>
const typename FrankList<T>::const_reverse_iterator& FrankList<T>::const_reverse
    this->ptr = this->ptr->prev;
    return *this;
}


template <typename T>
const typename FrankList<T>::const_reverse_iterator FrankList<T>::const_reverse_
    const_reverse_iterator tmp = *this;
    ++(*this);
    return tmp;
}

template <typename T>
const typename FrankList<T>::const_reverse_iterator& FrankList<T>::const_reverse
    this->ptr = this->ptr->next;
    return *this;
}

template <typename T>
const typename FrankList<T>::const_reverse_iterator FrankList<T>::const_reverse_
    const_reverse_iterator tmp = *this;
    --(*this);
    return tmp;
}

template <typename T>
FrankList<T>::const_reverse_iterator::const_reverse_iterator(Node* ptr) : base_i

template <typename T>
FrankList<T>::reverse_iterator::reverse_iterator(const base_iterator& rhv) : con

template <typename T>
FrankList<T>::reverse_iterator::reverse_iterator(base_iterator&& rhv) : const_re
   rhv.ptr = nullptr;
}

template <typename T>
typename FrankList<T>::reference FrankList<T>::reverse_iterator::operator*() {
    return this->ptr->val;
}

template <typename T>
typename FrankList<T>::pointer FrankList<T>::reverse_iterator::operator->() {
    return this->ptr;
}


template <typename T>
const typename FrankList<T>::reverse_iterator& FrankList<T>::reverse_iterator::o
{
   this -> ptr = rhv.ptr;
   return *this;
}

template <typename T>
const typename FrankList<T>::reverse_iterator& FrankList<T>::reverse_iterator::o
   if (this = &rhv){
      return *this;
   }

   this -> ptr = rhv.ptr;
   rhv.ptr = nullptr;
   return *this;
}

template <typename T>
FrankList<T>::reverse_iterator::reverse_iterator(Node* ptr) : const_reverse_iter

template <typename T>
FrankList<T>::const_asc_iterator::const_asc_iterator(const base_iterator& rhv) :

template <typename T>
FrankList<T>::const_asc_iterator::const_asc_iterator(base_iterator&& rhv) : base
   rhv.ptr = nullptr;
}

template <typename T>
const typename FrankList<T>::const_asc_iterator& FrankList<T>::const_asc_iterato
rhv){
   this -> ptr = rhv.ptr;
   return *this;
}

template <typename T>
const typename FrankList<T>::const_asc_iterator& FrankList<T>::const_asc_iterato
{
   if (this = &rhv){
      return *this;
   }

   this -> ptr = rhv.ptr;
   rhv.ptr = nullptr;
   return *this;
}

template <typename T>
typename FrankList<T>::const_reference FrankList<T>::const_asc_iterator::operato
    return this->ptr->val;
}

template <typename T>
typename FrankList<T>::const_pointer FrankList<T>::const_asc_iterator::operator-
    return this->ptr;
}

template <typename T>
const typename FrankList<T>::const_asc_iterator& FrankList<T>::const_asc_iterato
   this -> ptr = this -> ptr -> asc;
   return *this;
}

template <typename T>
const typename FrankList<T>::const_asc_iterator FrankList<T>::const_asc_iterator
   const_asc_iterator tmp = *this;
   ++(*this);
   return tmp;
}

template <typename T>
const typename FrankList<T>::const_asc_iterator& FrankList<T>::const_asc_iterato
   this -> ptr = this -> ptr -> desc;
   return *this;
}

template <typename T>
const typename FrankList<T>::const_asc_iterator FrankList<T>::const_asc_iterator
   const_asc_iterator tmp = *this;
   --(*this);
   return tmp;
}

template <typename T>
FrankList<T>::const_asc_iterator::const_asc_iterator(Node* ptr) : base_iterator(

template <typename T>
FrankList<T>::asc_iterator::asc_iterator(const base_iterator& rhv) : const_asc_i
  // this -> ptr = rhv.ptr;
}

template <typename T>
FrankList<T>::asc_iterator::asc_iterator(base_iterator&& rhv) : const_asc_iterat
   rhv.ptr = nullptr;
}

template <typename T>
typename FrankList<T>::reference FrankList<T>::asc_iterator::operator*() {
    return this->ptr->val;
}

template <typename T>
typename FrankList<T>::pointer FrankList<T>::asc_iterator::operator->() {
    return this->ptr;
}

template <typename T>
const typename FrankList<T>::asc_iterator& FrankList<T>::asc_iterator::operator=
   this -> ptr = rhv.ptr;
   return *this;
}

template <typename T>
const typename FrankList<T>::asc_iterator& FrankList<T>::asc_iterator::operator=
   if (this = &rhv){
      return *this;
   }

   this -> ptr = rhv.ptr;
   rhv.ptr = nullptr;
   return *this;
}

template <typename T>
FrankList<T>::asc_iterator::asc_iterator(Node* ptr) : const_asc_iterator (ptr)

template <typename T>
FrankList<T>::const_desc_iterator::const_desc_iterator(const base_iterator& rhv)
```

```cpp

template <typename T>
FrankList<T>::const_desc_iterator::const_desc_iterator(base_iterator&& rhv) : ba
   rhv.ptr = nullptr;
}

template <typename T>
const typename FrankList<T>::const_desc_iterator& FrankList<T>::const_desc_itera
base_iterator& rhv){
   this -> ptr = rhv.ptr;
   return *this;
}

template <typename T>
const typename FrankList<T>::const_desc_iterator& FrankList<T>::const_desc_itera
rhv) {
   this -> ptr = rhv.ptr;
   rhv.ptr = nullptr;
   return *this;
}

template <typename T>
typename FrankList<T>::const_reference FrankList<T>::const_desc_iterator::operat
   return this -> ptr -> val;
}

template <typename T>
typename FrankList<T>::const_pointer FrankList<T>::const_desc_iterator::operator
   return this -> ptr;
}

template <typename T>
const typename FrankList<T>::const_desc_iterator& FrankList<T>::const_desc_itera
   this -> ptr = this -> ptr -> desc;
   return *this;
}

template <typename T>
const typename FrankList<T>::const_desc_iterator FrankList<T>::const_desc_iterat
   const_desc_iterator tmp = *this;
   ++(*this);
   return tmp;
}

template <typename T>
const typename FrankList<T>::const_desc_iterator& FrankList<T>::const_desc_itera
   this -> ptr = this -> ptr -> asc;
   return *this;
}

template <typename T>
const typename FrankList<T>::const_desc_iterator FrankList<T>::const_desc_iterat
   const_desc_iterator tmp = *this;
   --(*this);
   return tmp;
}

template <typename T>
FrankList<T>::const_desc_iterator::const_desc_iterator(Node* ptr) : base_iterato

template <typename T>
FrankList<T>::desc_iterator::desc_iterator(const base_iterator& rhv) : const_des

template <typename T>
FrankList<T>::desc_iterator::desc_iterator(base_iterator&& rhv) : const_desc_ite
   rhv.ptr = nullptr;
}

template <typename T>
typename FrankList<T>::reference FrankList<T>::desc_iterator::operator*() {
   return this -> ptr -> val;
}

template <typename T>
typename FrankList<T>::pointer FrankList<T>::desc_iterator::operator->() {
   return this -> ptr;
}

template <typename T>
const typename FrankList<T>::desc_iterator& FrankList<T>::desc_iterator::operato
   this -> ptr = rhv.ptr;
   return *this;
}

template <typename T>
const typename FrankList<T>::desc_iterator& FrankList<T>::desc_iterator::operato
   this -> ptr = rhv.ptr;
   rhv.ptr = nullptr;
   return *this;
}

template <typename T>
FrankList<T>::desc_iterator::desc_iterator(Node* ptr) : const_desc_iterator(ptr)

template <typename T>
FrankList<T>::const_multi_iterator::const_multi_iterator(const base_iterator& rh

template <typename T>
FrankList<T>::const_multi_iterator::const_multi_iterator(base_iterator&& rhv) :
   rhv.ptr = nullptr;
}

template <typename T>
const typename FrankList<T>::const_multi_iterator& FrankList<T>::const_multi_ite
base_iterator& rhv){
   this -> ptr = rhv.ptr;
   return *this;
}

template <typename T>
const typename FrankList<T>::const_multi_iterator& FrankList<T>::const_multi_ite
rhv) {
   this -> ptr = rhv.ptr;
   rhv.ptr = nullptr;
   return *this;
}

template <typename T>
```

```cpp
typename FrankList<T>::const_reference FrankList<T>::const_multi_iterator::opera
   return this -> ptr -> val;
}

template <typename T>
typename FrankList<T>::const_pointer FrankList<T>::const_multi_iterator::operato
   return this -> ptr;
}

template <typename T>
const typename FrankList<T>::const_multi_iterator& FrankList<T>::const_multi_ite
   if (mode){
      this -> ptr = this -> ptr -> next;
   } else {
      this -> ptr = this -> ptr -> asc;
   }

   return *this;
}

template <typename T>
const typename FrankList<T>::const_multi_iterator FrankList<T>::const_multi_iter
   const_multi_iterator tmp(*this);
   if (mode){
      this -> ptr = this -> ptr -> next;
   } else {
      this -> ptr = this -> ptr -> asc;
   }

   return tmp;
}

template <typename T>
const typename FrankList<T>::const_multi_iterator& FrankList<T>::const_multi_ite
   if (mode){
      this -> ptr = this -> ptr -> prev;
   } else {
      this -> ptr = this -> ptr -> desc;
   }

   return *this;
}

template <typename T>
const typename FrankList<T>::const_multi_iterator FrankList<T>::const_multi_iter
   const_multi_iterator tmp(*this);
   if (mode){
      this -> ptr = this -> ptr -> prev;
   } else {
      this -> ptr = this -> ptr -> desc;
   }

   return tmp;
}

template <typename T>
void FrankList<T>::const_multi_iterator::chmod() {
   mode = !mode;
}

template <typename T>
FrankList<T>::const_multi_iterator::const_multi_iterator(Node* ptr) : base_itera

template <typename T>
FrankList<T>::multi_iterator::multi_iterator(const base_iterator& rhv) : const_m

template <typename T>
FrankList<T>::multi_iterator::multi_iterator(base_iterator&& rhv) : const_multi_
   rhv.ptr = nullptr;
}

template <typename T>
typename FrankList<T>::reference FrankList<T>::multi_iterator::operator*() {
   return this -> ptr -> val;
}

template <typename T>
typename FrankList<T>::pointer FrankList<T>::multi_iterator::operator->() {
   return this -> ptr;
}

template <typename T>
const typename FrankList<T>::multi_iterator& FrankList<T>::multi_iterator::opera
   this -> ptr = rhv.ptr;
   return *this;
}

template <typename T>
const typename FrankList<T>::multi_iterator& FrankList<T>::multi_iterator::opera
   this -> ptr = rhv.ptr;
   rhv.ptr = nullptr;
   return *this;
}

template <typename T>
FrankList<T>::multi_iterator::multi_iterator(Node* ptr) : const_multi_iterator(p

template <typename T>
FrankList<T>::const_multi_reverse_iterator::const_multi_reverse_iterator(const b
base_iterator(rhv.ptr) {}

template <typename T>
FrankList<T>::const_multi_reverse_iterator::const_multi_reverse_iterator(base_it
base_iterator(rhv.ptr) {
   rhv.ptr = nullptr;
}

template <typename T>
const typename FrankList<T>::const_multi_reverse_iterator& FrankList<T>::const_m
(const base_iterator& rhv){
   this -> ptr = rhv.ptr;
   return *this;
}

template <typename T>
const typename FrankList<T>::const_multi_reverse_iterator& FrankList<T>::const_m
(base_iterator&& rhv) {
   this -> ptr = rhv.ptr;
   rhv.ptr = nullptr;
```

```cpp
    return *this;
}

template <typename T>
typename FrankList<T>::const_reference FrankList<T>::const_multi_reverse_iterato
    return this -> ptr -> val;
}

template <typename T>
typename FrankList<T>::const_pointer FrankList<T>::const_multi_reverse_iterator:
    return this -> ptr;
}

template <typename T>
const typename FrankList<T>::const_multi_reverse_iterator& FrankList<T>::const_m
(){
    if (mode){
        this -> ptr = this -> ptr -> prev;
    } else {
        this -> ptr = this -> ptr -> desc;
    }

    return *this;
}

template <typename T>
const typename FrankList<T>::const_multi_reverse_iterator FrankList<T>::const_mu
(int){
    const_multi_reverse_iterator tmp(*this);
    ++(*this);
    return tmp;
}

template <typename T>
const typename FrankList<T>::const_multi_reverse_iterator& FrankList<T>::const_m
(){
    if (mode){
        this -> ptr = this -> ptr -> next;
    } else {
        this -> ptr = this -> ptr -> asc;
    }

    return *this;
}

template <typename T>
const typename FrankList<T>::const_multi_reverse_iterator FrankList<T>::const_mu
(int){
    const_multi_reverse_iterator tmp(*this);
    --(*this);
    return tmp;
}

template <typename T>
void FrankList<T>::const_multi_reverse_iterator::chmod() {
     mode = !mode;
}

template <typename T>
FrankList<T>::const_multi_reverse_iterator::const_multi_reverse_iterator(Node* p
```

```cpp

template <typename T>
FrankList<T>::multi_reverse_iterator::multi_reverse_iterator(const base_iterator
const_multi_reverse_iterator(rhv.ptr) {}

template <typename T>
FrankList<T>::multi_reverse_iterator::multi_reverse_iterator(base_iterator&& rhv
const_multi_reverse_iterator(rhv.ptr) {
    rhv.ptr = nullptr;
}

template <typename T>
typename FrankList<T>::reference FrankList<T>::multi_reverse_iterator::operator*
    return this -> ptr -> val;
}

template <typename T>
typename FrankList<T>::pointer FrankList<T>::multi_reverse_iterator::operator->(
    return this -> ptr;
}

template <typename T>
const typename FrankList<T>::multi_reverse_iterator& FrankList<T>::multi_reverse
base_iterator& rhv){
    this -> ptr = rhv.ptr;
    return *this;
}

template <typename T>
const typename FrankList<T>::multi_reverse_iterator& FrankList<T>::multi_reverse
(base_iterator&& rhv) {
    this -> ptr = rhv.ptr;
    rhv.ptr = nullptr;
    return *this;
}

template <typename T>
FrankList<T>::multi_reverse_iterator::multi_reverse_iterator(Node* ptr) : const_

}



#endif
```