Задание 1.3

Постройте шаблон класса двусвязного списка путём наследования от класса IteratedLinkedList. Реализуйте функции добавления элемента push() и удаления элемента pop() в классе-наследнике D (для четных вариантов D – Стек, для нечетных – Очередь) согласно схеме: для класса Стек элементы добавляются в конец, извлекаются с конца; для класса Очереди элементы добавляются в конец, извлекаются с начала. Введите очистку памяти, выделенной под элементы списка, в деструкторе.

Постройте наследник класса D. Переопределите функцию добавления нового элемента таким образом, чтобы контейнер оставался упорядоченным.

Реализуйте функцию filter() из пункта 1 – результатом должен стать объект типа D. Реализуйте функцию универсального фильтра, принимающего список произвольного типа (участвующего в схеме наследования) и возвращающего список произвольного типа (тип обрабатываемого списка не обязан совпадать с типом списка-результата).

Код 1.3. Абстрактный класс для связного списка LinkedListParent (функции push() и pop() чисто виртуальные) и IteratedLinkedList (введен механизм работы итераторов) и другие вспомогательные классы

```
    #include <iostream>
    #include <fstream>
    using namespace std;
    template <class T>
    class Element
    {//элемент связного списка
    private:
     //указатель на предыдущий и следующий элемент
     Element* next;
     Element* prev;
     //информация, хранимая в поле
    T field;
    public:
    Element(T value = 0, Element<T> * next_ptr = NULL, Element<T>
* prev_ptr = NULL)
     {
          field = value;
          next = next_ptr;
          prev - prev_ptr;
     }
```

```
    //доступ к полю *next
     virtual Element* getNext() { return next; }
     virtual void setNext(Element* value) { next = value; }

     //доступ к полю *prev
     virtual Element* getPrevious() { return prev; }
     virtual void setPrevious(Element* value) { prev = value; }

     //доступ к полю с хранимой информацией field
     virtual T getValue() { return field; }
     virtual void setValue(T value) { field = value; }

     template<class T> friend ostream& operator<< (ostream&
ustream, Element<T>& obj);
    };

    template<class T>
    ostream& operator << (ostream& ustream, Element<T>& obj)
    {
     ustream << obj.field;
     return ustream;
    }

    template <class T>
    class LinkedListParent
    {
    protected:
     //достаточно хранить начало и конец
     Element<T>* head;
     Element<T>* tail;
     //для удобства храним количество элементов
     int num;
    public:
    virtual int Number() { return num; }
    virtual Element<T>* getBegin() { return head; }
    virtual Element<T>* getEnd() { return tail; }

     LinkedListParent()
     {
          //конструктор без параметров
          cout << "\nParent constructor";
          head = NULL;
          num = 0;
```

```
}
//чисто виртуальная функция: пока не определимся с типом списка, не сможем реализовать добавление
     virtual Element<T>* push(T value) = 0;

     //чисто виртуальная функция: пока не определимся с типом списка, не сможем реализовать удаление
     virtual Element<T>* pop() = 0;

     virtual ~LinkedListParent()
     {
          //деструктор - освобождение памяти
          cout << "\nParent destructor";
     }

     //получение элемента по индексу - какова асимптотическая оценка этого действия?
     virtual Element<T>* operator[](int i)
     {
          //индексация
          if (i<0 || i>num) return NULL;
          int k = 0;

          //ищем i-й элемент - вставем в начало и отсчитываем i шагов вперед
          Element<T>* cur = head;
          for (k = 0; k < i; k++)
          {
               cur = cur->getNext();
          }
          return cur;
     }
    template<class T> friend ostream& operator<< (ostream& ustream, LinkedListParent<T>& obj);
     template<class T> friend istream& operator>> (istream& ustream, LinkedListParent<T>& obj);
    };
    template<class T>
    ostream& operator << (ostream& ustream, LinkedListParent<T>& obj)
    {
     if (typeid(ustream).name() == typeid(ofstream).name())
     {
          ustream << obj.num << "\n";
```

```
          for (Element<T>* current = obj.getBegin(); current !=
NULL; current = current->getNext())
               ustream << current->getValue() << " ";
          return ustream;
     }

     ustream << "\nLength: " << obj.num << "\n";
     int i = 0;
     for (Element<T>* current = obj.getBegin(); current != NULL;
current = current->getNext(), i++)
          ustream << "arr[" << i << "] = " << current->getValue()
<< "\n";

     return ustream;
    }

    template<class T>
    istream& operator >> (istream& ustream, LinkedListParent<T>&
obj)
    {
     //чтение из файла и консоли совпадают
     int len;
     ustream >> len;
     //здесь надо очистить память под obj, установить obj.num = 0
     double v = 0;
    for (int i = 0; i < len; i++)
     {
          ustream >> v;
          obj.push(v);
     }
     return ustream;
    }
    template<typename ValueType>
    class ListIterator : public
std::iterator<std::input_iterator_tag, ValueType>
    {
    private:

    public:
     ListIterator() { ptr = NULL; }
     //ListIterator(ValueType* p) { ptr = p; }
     ListIterator(Element<ValueType>* p) { ptr = p; }
     ListIterator(const ListIterator& it) { ptr = it.ptr; }
```

```
     bool operator!=(ListIterator const& other) const { return
ptr != other.ptr; }
     bool operator==(ListIterator const& other) const { return
ptr == other.ptr; }//need for BOOST_FOREACH
     Element<ValueType>& operator*()
     {
          return *ptr;
     }
     ListIterator& operator++() { ptr = ptr->getNext(); return
*this; }
     ListIterator& operator++(int v) { ptr = ptr->getNext();
return *this; }

     ListIterator& operator=(const ListIterator& it) { ptr =
it.ptr; return *this; }
     ListIterator& operator=(Element<ValueType>* p) { ptr = p;
return *this; }
    private:
     Element<ValueType>* ptr;
    };
    template <class T>
    class IteratedLinkedList : public LinkedListParent<T>
    {
    public:
     IteratedLinkedList() : LinkedListParent<T>() { cout <<
"\nIteratedLinkedList constructor"; }
     virtual ~IteratedLinkedList() { cout <<
"\nIteratedLinkedList destructor"; }
ListIterator<T> iterator;

     ListIterator<T> begin() { ListIterator<T> it =
LinkedListParent<T>::head; return it; }
     ListIterator<T> end() { ListIterator<T> it =
LinkedListParent<T>::tail; return it; }
    };
```

Задание 1.4

Постройте итераторы для перемещения по списку (введите операции чтение элемента списка по итератору *, операции перемещения по списку ++ и --, вспомогательные операции). Переопределите функцию вывода содержимого списка с помощью итераторов. Итератор не должен быть полем

в контейнере. Введите исключение для попытки чтения значения в случае, когда итератор не связан ни с каким элементом.

Задание 1.5

Постройте шаблон класса списка D (из задания в пункте 3), который хранит объекты класса C (из задания в пункте 2), сохраняя упорядоченность по приоритету: полю или группе полей, указанных в варианте. Переопределите операции добавления и удаления элементов с использованием итераторов.