Отчёт

А.А.Федяшов, Д.Е.Теплюков

11.07.2017

1 Постановка задачи

1.Написать класс двусвязный список с внешним итератором
Класс должен содержать:
1)добавление элемента в начало, конец списка;
2)удаление элемента из начала, конца списка;
3)подсчёт количества элементов в списке;
4)проверка списка на пустоту;
5)получение итератора начала, конца списка;
6)вставка значения до, после итератора;
7)удаление значения по итератору;
8)поиск значения в списке и получения итератора с указанным значением;
9)ввод/вывод в поток.
2.**Используя двусвязный список целых чисел решить задачу:**
Пользователем вводится последовательность элементов, признак окончания последовательности – не число (некорректный ввод). Реализовать хранение чисел в списке в отсортированном виде, без повторяющихся значений. По окончании ввода вывести значения в порядке возрастания.
3.**Реализовать QtTest для класса список и класса итератор.**

2 Содержание проекта

1)Заголовочные файлы(.h)

dlist.h-содержит объявление методов класса DList и DIterator

qtest1.h-содержит объявление функций тестирования классов DList и DIterator

utility.h-содержит объявление функции решения поставленной задачи для класса DList

2)Файлы реализации(.cpp)

main.cpp-главный файл проекта, содержит вызов тестов из заголовочного файла qtest1.h и вызов программы из заголовочного файла utility.h. Также в файле реализован цикл для многократного ввода значений для программы из файла utility.h, что позволяет тестировать программу столько раз, сколько необходимо пользователю

dlist.cpp-содержит реализацию методов из заголовочного файла dlist.h для классов DList и DIterator

qtest1.cpp-содержит реализацию функций тестирования из заголовочного файла qtest1.h для классов DList и DIterator

utility.cpp-содержит реализацию функции решения поставленной задачи из заголовочного файла utility.h для класса DList

3 Описание классов DList И DIterator

DList

Класс список в представленном коде имеет название DList, содержит вложенный класс DIterator.

Следующие конструкторы и методы имеют права доступа public:

1)**Конструктор по умолчанию, возвращает пустой список:**

DList()

2)**Конструктор копии:**

DList(const DList &l)

3)**Конструктор, реализованный посредством объекта initializer_list<int>, - список инициализации контейнера**

DList(const std::initializer_list<int> & list)

Пример:

DList mc={1,2,3,4}

Класс DList содержит перегрузки следующих операторов: присваивания(=), сравнения на равенство(==) и вывода(«).
Также этот класс имеет следующий перечень методов:

1)**Добавление элемента в конец списка:**

void push_back(const int &x)

2)**Добавление элемента в начало списка:**

void push_front(const int &x)

3)**Удаление первого элемента списка списка:**

void pop_front()

В случае, когда список пустой, удаление не происходит, т.е список остается в неизменном состоянии

4)**Удаление последнего элемента списка:**

void pop_back()

В случае, когда список пустой, удаление не происходит, т.е список остается в неизменном состоянии

5)**Возвращение значения первого элемента списка**

int front() const

Возвращаемое значение типа int

6)**Возвращение значения последнего элемента списка**

int back() const

Возвращаемое значение типа int

7)**Очистка содержимого списка:**

void erase()

В результате работы функции получается пустой список

8)**Проверка на пустоту**

bool is_empty() const

Если список пуст, то возвращается значение true, иначе false

9)**Нахождение размера списка**

size_t size() const

В поле private списка DList содержится структура node, которая хранит в себе основные поля списка

int data - значение элемента списка
*node *prev* - ссылку на предыдущий элемент списка
*node *next* - ссылку на следующий элемент списка

Также с правами доступа private

*node *first* - указатель на первый элемент списка
*node *last* - указатель на последний элемент списка

и 2 функции

*void copy_DList(const node *from_first, const node *from_last)*-копирования содержимого списка в другой список
void delete_DList()-удаления списка

Класс DIterator

Следующие конструкторы и методы имеют права доступа public

Конструктор с параметрами, где первый - указатель на голову списка, а второй - указатель на текущий элемент списка:

*DIterator(DList *collection, node *current)*

Класс DIterator содержит перегрузки следующих операторов: инкремент (префиксный/постфиксный)(++), декремент (префиксный/постфиксный)(–), разыменование(*), сравнение на равенство(==), неравенство(!=), больше(>) и меньше(<)
Также этот класс имеет следующий перечень методов:

1)**Возвращение итератора на начало списка:**

DIterator begin()

2)**Возвращение итератора на конец списка:**

DIterator end()

3)**Поиск элемента с заданным значением:**

DIterator find(const int &x)

Входное значение типа int, функция возвращает итератор на искомый элемент

4)**Поиск первого элемента больше заданного:**

DIterator findFEGG(const int &x)

Входное значение типа int, функция возвращает итератор на искомый элемент

Этой функции уделим отдельное внимание потому, что она занимает одну из ключевых ролей в работе программы по предложенной задаче. Сначала в функции идет проверка на кол-во элементов списка:
if (this->size()>1)

Эта проверка имеет смысл. Если кол-во элементов меньше или ровно 1, то не стоит выполнять достаточно большой блок действий, который содержится внутри true ветки if. Для обработки этой ситуации достаточно такой проверки:

if (DList_DIterator.current->data>x)

Если условие верно, то возвращаем указатель на наш единственный элемент списка.

Если кол-во элементов списка > 1, то осуществляется поиск такого элемента, который больше заданного и элемент перед ним меньше заданного. Такое условие поиска гарантирует нам, что мы не нарушим формирующуюся возрастающую последовательность в программе для задачи.
После нахождения первого такого элемента, мы возвращаем на него итератор.

Хочется отметить, что эта функция специально писалась для предложенной задачи, и ряд решений в этой функции адаптирован специально для того, чтобы не испортить правильно формирующуюся последовательность в программе для задачи.

5)**Вставка элемента, после элемента на который указывает итератор:**

void insert_after(const DIterator &it, const int &x)

Входные значения: итератор на элемент const DIterator &it, после которого нужно вставить const int &x

6)**Вставка элемента, до элемента на который указывает итератор:**

void insert_before(const DIterator &it, const int &x)

Входные значения: итератор на элемент const DIterator &it, до которого нужно вставить const int &x

7)**Удаление элемента, на который указывает итератор:**

void remove(const DIterator &it)

Входные значения: указатель на элемент, который нужно удалить

4 Тестирование классов

Класс Test_DList(тестирующий класс)

При создании класса использовалась библиотека <QObject>
Класс Test_DList создан посредством наследования от исходного класса QObject

В поле private этого класса хранятся функции тестирования методов классов DList и DIterator
Также используются некоторые специфические методы тестирования из QTest, такие как:

void initTestCase()-функция первичной инициализации, которая выполняет некоторую последовательность действий перед запуском тестов

void cleanupTestCase()-функция, которая выполняет некоторую последовательность действий после выполнения тестов

Функции initTestCase() и cleanupTestCase() использованы для более корректного и наглядного вывода информации о ходе тестирования

Пример теста

Пример теста будет показан на примере метода push_back()

```
void Test_DList::push_back()
{
DList mc={1,2,3,4};
DList expected={1,2,3,4,5};
mc.push_back(5);
QVERIFY(mc==expected);
}
```

Объявляем исходный список mc, и ожидаемый expected.
Вызываем метод push_back() для класса mc
В функции QVERIFY происходит сравнение модифицированного списка и ожидаемого. В случае если в результате сравнения вернется true, то тест выдаст сообщение Pass!, означающие, что тест пройден успешно, иначе, если вернется значение false, то в терминал будет выведено сообщение об ошибке. Есть возможность конкретизировать ошибку и воспользоваться функцией QVERIFY2, которая в качестве 2 параметра принимает текстовое сообщение об ошибке.

5 Особенности реализации программы

Для начала необходимо было перехватить поток терминала, для этого выполнили следующее действие:

QTextStream IN(stdin)

Обработали условие, что если сразу получаем символ конца строки '.', то список не формируется.
Повторений одинаковых элементов избегаем с помощью проверки следующего условия:

if (list1.find(b.toInt())==list1.end())

Ключевым моментом программы является то, что мы ищем текущий максимум в списке.
Если нововходящий элемент больше максимума, то, очевидно, этот элемент нужно добавить в конец, иначе его нужно вставить перед первым элементом больше него. Осуществляется это следующим образом:

list1.insert_before(list1.findFEGG(b.toInt()),b.toInt())

В конце программы выводим уже отсортированный список.