МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

«МИРЭА – Российский технологический университет»

РТУ МИРЭА

Отчет по выполнению практического занятия 7

Тема: Двунаправленные динамические списки

Дисциплина: Структуры и алгоритмы обработки данных

Выполнил студент Хвостов В. В.

Группа ИКБО-01-20

Москва 2021

Содержание

Главное задание

1.1. Постановка задачи

Разработать многомодульную программу, которая демонстрирует выполнение всех операций, определенных вариантом, над линейным двунаправленным динамическим списком и функции для работы с этой структурой:

1.2. Определение операций над списком

Определение структуры узла двунаправленного списка

Согласно варианту No9 в качестве информационной части узла списка используются поля: марка автомобиля, страна изготовитель, год выпуска.

Описание обязательных операций над списком

Обязательные:

1. Вывод списка в двух направлениях.
2. Поиск узла с заданным значением.

Функции дополнительного задания варианта:

3. Вставка узла по автомобилю в списке своего модельного ряда перед узлом, год выпуска которого меньше.
4. Формирование нового списка с узлом вида ("модельный" узел): марка автомобиля и указатель на начало модельного ряда данной марки в исходном списке.
5. Удаление информации обо всех автомобилях заданной марки.

Алгоритмы операций над списком

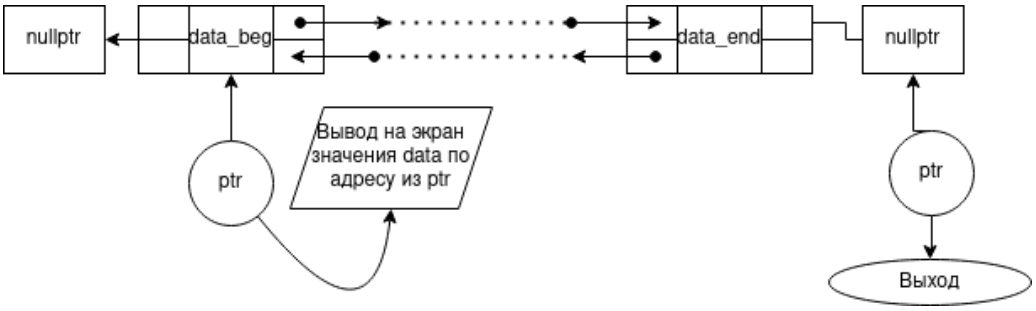

Рис. 1 - Алгоритм вывода списка как в прямом, так и обратном направлениях

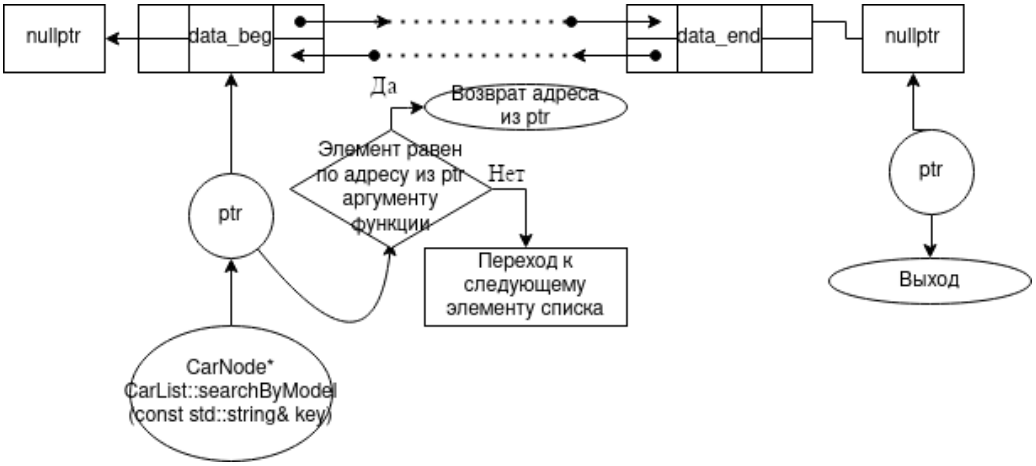

Рис. 2 - Алгоритм поиска первого элемента с заданным названием модели

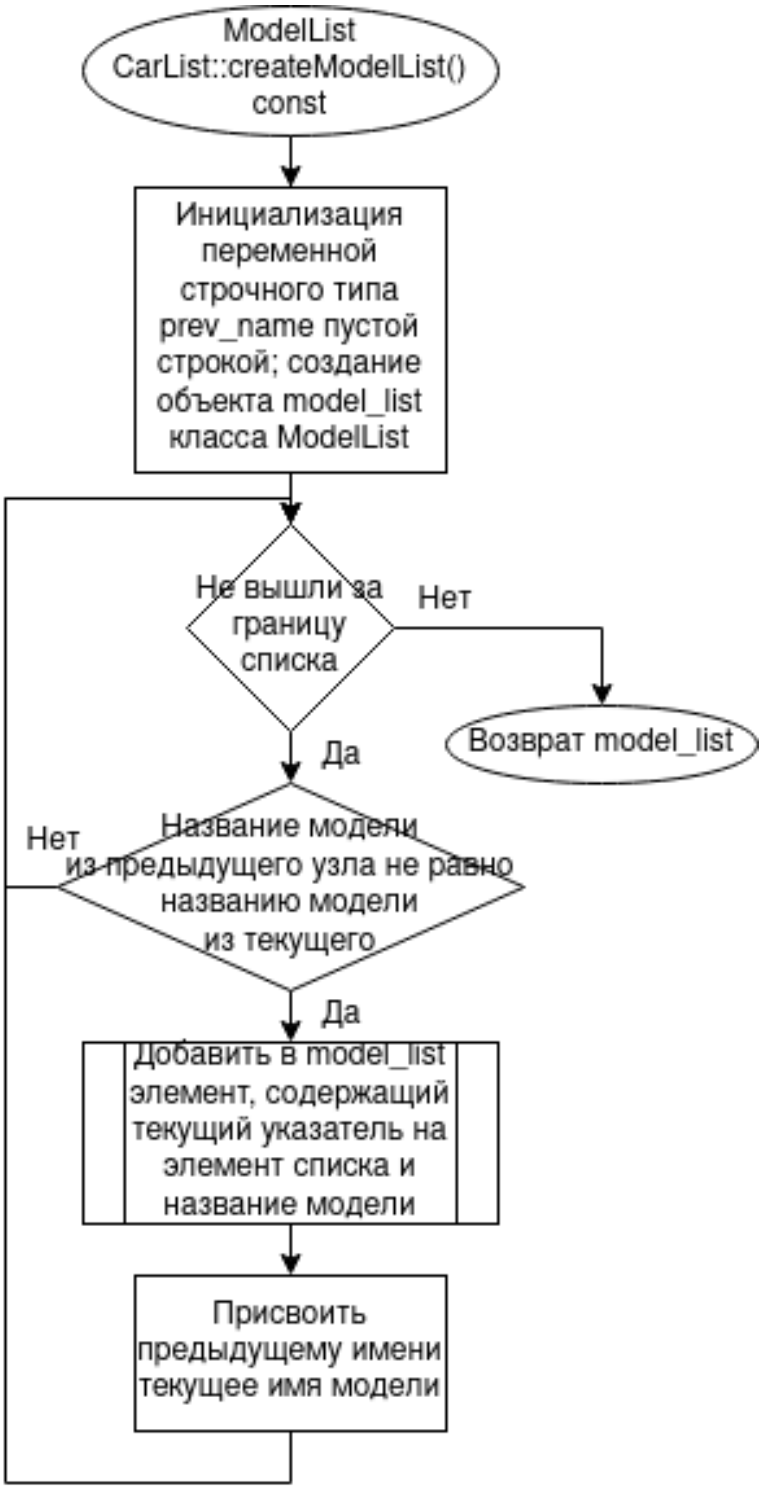

Рис. 3 - Алгоритм создания списка с "модельными" узлами

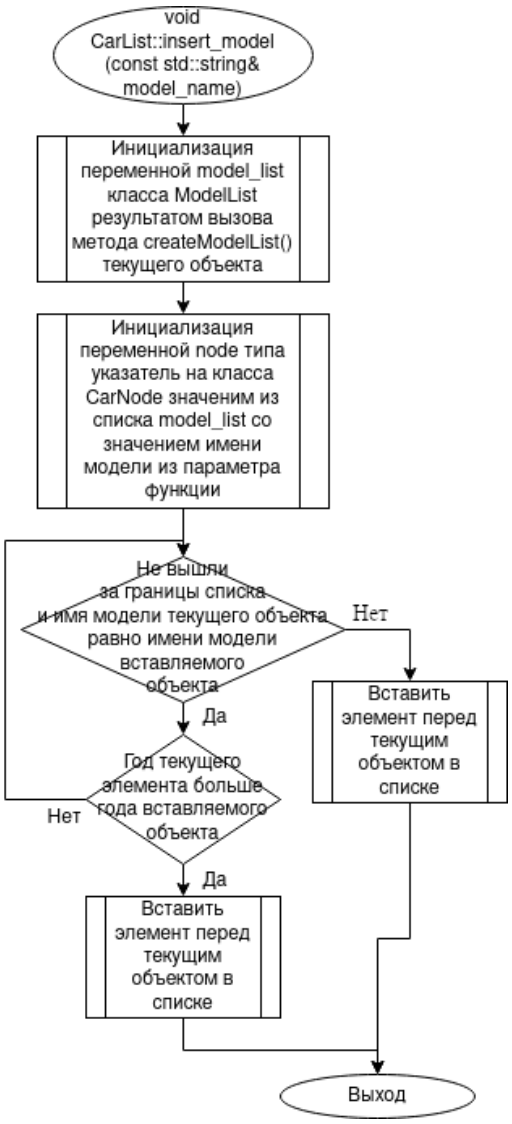

Рис. 4 - Алгоритм вставки элемента перед той же моделью с меньшим годом

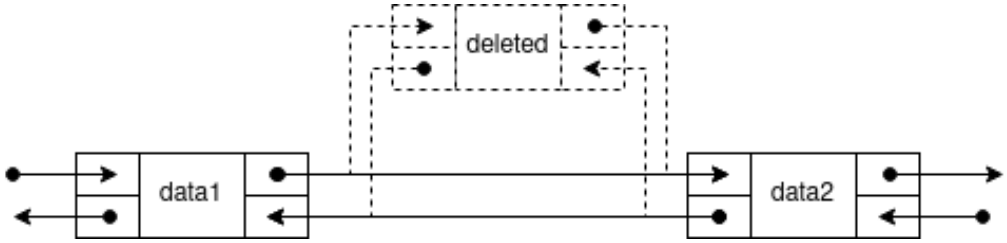

Рис. 5 - Алгоритм удаления одного узла (требуется в следующем алгоритме)

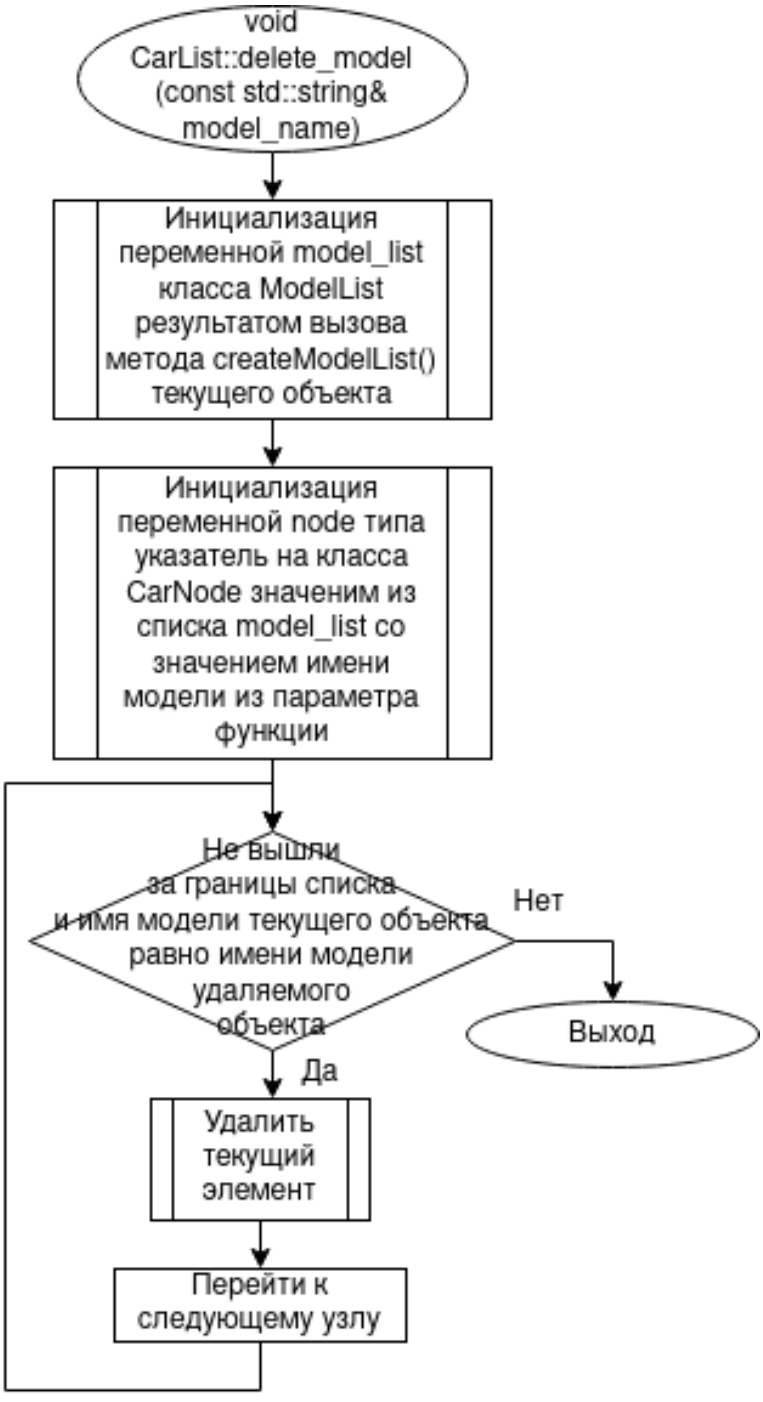

Рис. 6 - Алгоритм удаления всех элементов с заданным названием модели

1.3. Код программы

Реализация на C++:

Код файла car_list.h

```
#ifndef _CAR_LIST_H
#define _CAR_LIST_H
#include <iostream>
#include <string>

// I'm so lazy to create the iterator for this container
// + iterator to iterate only through specific models hmmm

struct CarNode {
  struct CarData {
    std::string car_model;
    std::string country;
    int year;
    friend std::ostream& operator<<(std::ostream& stream, const CarData& cdata);
  };
  CarData data;
  CarNode* prev;
  CarNode* next;
};

typedef CarNode::CarData CarData;

// Why I create this ... (std::list : hello bro)
class ModelList {
  struct ModelNode {
    std::string model_name;
    CarNode* model_first;
    ModelNode* prev;
    ModelNode* next;
  };
public:
  void push_back(const std::string& name, CarNode* model_ptr);
  CarNode* getFirstNode(const std::string& name);
private:
  ModelNode* m_begin = nullptr;
  ModelNode* m_end = nullptr;
  size_t m_size = 0;
};

class CarList {
public:
  CarList() = default;
  ~CarList();
```

```
    void    printList  ( )  const ;
    void     reversePrintList   ( )  const ;

    void    insert  ( const   CarData& value ,    size_t    pos ) ;
    void    insert  ( const   CarData& value ,  CarNode* ptr ) ;
    void    push_front  ( const   CarData& value ) ;
    void   push_back ( const   CarData& value ) ;

    CarNode* searchByModel ( const   std :: string & key)  const ;
    CarData& operator [] ( size_t    pos ) ;

    void    sort ( ) ;

    void    delete_row (CarNode* node ) ;
    void   pop_back ( ) ;
    void    pop_front ( ) ;

    //  List   should   be   already    sorted ,   or   the   next    functions    won't  work

    void    insert_model  ( const   CarData&  car_data ) ;
    ModelList    createModelList  ( )  const ;
    void    delete_model  ( const   std :: string & model_name);
  private :
    CarNode* m_begin  =   nullptr  ;
    CarNode* m_end =   nullptr  ;
     size_t    m_size  = 0;
} ;
# endif   //  _CAR_LIST_H
```

Код файла car_list.cpp

```
# include   < iostream >
# include   < string >
# include   < vector >

# include   "  car_list   .h"

//  I'm so   lazy   to   create    the    iterator    for    this    container
//  +   iterator    to    iterate    only   through    specific    models  hmmm
//  Also  ...

std :: ostream & operator <<( std :: ostream & stream ,   const   CarData& cdata )  {
    stream   <<  cdata . car_model   <<  "   "  <<  cdata . country
       <<  "   "  <<  cdata . year ;
    return    stream ;
}

typedef   CarNode :: CarData   CarData ;
```

```
void ModelList::push_back(const std::string& name, CarNode* model_ptr) {
  ModelNode* node = new ModelNode();
  node->model_name = name;
  node->model_first = model_ptr;
  node->next = nullptr;
  node->prev = m_end;
  if (m_end != nullptr) {
    m_end->next = node;
  } else {
    m_begin = (m_end = node);
  }
  m_end = node;
  ++m_size;
}

CarNode* ModelList::getFirstNode(const std::string& name) {
  for (ModelNode* ptr = m_begin; ptr != nullptr; ptr = ptr->next) {
    if (ptr->model_name == name) {
      return ptr->model_first;
    }
  }
  return nullptr;
}

CarList::~CarList() {
  for (CarNode* ptr = m_begin; ptr != nullptr;) {
    CarNode* p = ptr;
    ptr = ptr->next;
    delete p;
  }
}
void CarList::printList() const {
  for (CarNode* ptr = m_begin; ptr != nullptr; ptr = ptr->next) {
    std::cout << ptr->data << '\n';
  }
}

void CarList::push_back(const CarData& value) {
  CarNode* node = new CarNode();
  node->data = value;
  node->next = nullptr;
  node->prev = m_end;
  if (m_end != nullptr) {
    m_end->next = node;
  } else {
    m_begin = (m_end = node);
  }
  m_end = node;
```

```
  ++m_size;
}

void CarList::push_front(const CarData& value) {
  CarNode* node = new CarNode();
  node->data = value;
  node->prev = nullptr;
  node->next = m_begin;
  if (m_begin != nullptr) {
    m_begin->prev = node;
  } else {
    m_begin = (m_end = node);
  }
  m_begin = node;
  ++m_size;
}

// I can speed up this algo ~ in two times, but I have no time for games =)
// Insert before the position
void CarList::insert(const CarData& value, size_t pos) {
  if (pos == 0) {
    return push_front(value);
  }
  CarNode* node = new CarNode();
  CarNode* ptr = m_begin;
  for (size_t i = 0; i < pos; ++i, ptr = ptr->next);
  node->data = value;
  node->next = ptr;
  if (ptr == nullptr) {
    node->prev = nullptr;
    m_begin = m_end = node;
    return;
  }
  node->prev = ptr->prev;
  if (ptr->prev != nullptr) {
    ptr->prev->next = node;
  }
  ptr->prev = node;
  ++m_size;
}

void CarList::insert(const CarData& value, CarNode* ptr) {
  if (ptr == m_begin) {
    return push_front(value);
  }
  CarNode* node = new CarNode();
  node->data = value;
  node->next = ptr;
```

```
    if (ptr == nullptr) {
        node->prev = nullptr;
        m_begin = m_end = node;
        return;
    }
    node->prev = ptr->prev;
    if (ptr->prev != nullptr) {
        ptr->prev->next = node;
    }
    ptr->prev = node;
    ++m_size;
}


void CarList::reversePrintList() const {
    for (CarNode* ptr = m_end; ptr != nullptr; ptr = ptr->prev) {
        std::cout << ptr->data << '\n';
    }
}

CarNode* CarList::searchByModel(const std::string& key) const {
    for (CarNode* ptr = m_begin; ptr != nullptr; ptr = ptr->next) {
        if (ptr->data.car_model == key) {
            return ptr;
        }
    }
    return nullptr;
}

// Quick sort algo
void CarList::sort() {
    std::vector<CarData> v;
    for (CarNode* ptr = m_begin; ptr != nullptr; ptr = ptr->next) {
        v.push_back(std::move(ptr->data));
    }

    std::sort(v.begin(), v.end(), [](const CarData& c1, const CarData& c2)
        { return c1.car_model < c2.car_model; });

    std::string prev_name = "";
    auto prev_it = v.begin();
    int counter = 0;
    for (auto it = v.begin(); it != v.end(); ++it) {
        if (prev_name != it->car_model) {
            std::sort(prev_it, prev_it + counter,
                [](const CarData& c1, const CarData& c2)
                { return c1.year < c2.year; });
            counter = 0;
```

```
        prev_it = it;
        prev_name = it->car_model;
      }
      ++counter;
    }
    std::sort(prev_it, prev_it + counter,
              [](const CarData& c1, const CarData& c2)
              { return c1.year < c2.year; });

    auto it = v.cbegin();
    for (CarNode* ptr = m_begin; ptr != nullptr; ptr=ptr->next, ++it) {
      ptr->data = std::move(*it);
    }
}

CarData& CarList::operator[](size_t pos) {
    CarNode* ptr = m_begin;
    for (size_t i = 0; ptr != nullptr && i < pos; ++i, ptr=ptr->next);
    return ptr->data;
}

// List should be already sorted, or the next functions won't work

void CarList::insert_model(const CarData& car_data) {
    ModelList model_list = createModelList();
    CarNode* node = model_list.getFirstNode(car_data.car_model);
    auto ptr = node;
    for (; ptr != nullptr && node->data.car_model == ptr->data.car_model; ptr=ptr->next) {
      if (car_data.year < ptr->data.year) {
        this->insert(car_data, ptr);
        return;
      }
    }
    if (ptr != nullptr) {
      this->insert(car_data, ptr);
    } else {
      this->push_back(car_data);
    }
}

ModelList CarList::createModelList() const {
    ModelList model_list;
    std::string prev_name = "";
    for (auto ptr = m_begin; ptr != nullptr; ptr=ptr->next) {
      if (prev_name != ptr->data.car_model) {
        model_list.push_back(ptr->data.car_model, ptr);
        prev_name = ptr->data.car_model;
      }
```

```
    }
    return model_list;
}

void CarList::delete_model(const std::string& model_name) {
    ModelList model_list = createModelList();
    CarNode* node = model_list.getFirstNode(model_name);
    for (auto ptr = node; ptr != nullptr && model_name == ptr->data.car_model;) {
        CarNode* p = ptr;
        ptr = ptr->next;
        this->delete_row(p);
    }
}

void CarList::delete_row(CarNode* node) {
    if (node == m_begin) {
        return pop_front();
    } else if (node == m_end) {
        return pop_back();
    } else {
        CarNode* ptr = node;
        node->prev->next = node->next;
        node->next->prev = node->prev;
        delete ptr;
    }
}

void CarList::pop_back() {
    CarNode* ptr = m_end;
    m_end = m_end->prev;
    m_end->next = nullptr;
    delete ptr;
}

void CarList::pop_front() {
    CarNode* ptr = m_begin;
    m_begin = m_begin->next;
    m_begin->prev = nullptr;
    delete ptr;
}
```

Код файла linked_list.cpp

```
#include <iostream>
#include <random>
#include <string>
#include <vector>

#include "car_list.h"
```

```
// I'm so lazy to create the iterator for this container
// + iterator to iterate only through specific models hmmm

typedef std :: vector<std :: string > Words;
const Words model_names = {"Skoda", "Bentley", "Polestar", "Dodge",
"Volvo", "Landwind", "Rimac", "Acura", "Tata", "Proton", "Audi",
"BMW", "Bugatti", "Cherry", "Chevrolet", "Citroen", "Ferrari", "GAZ",
"Honda", "Hyondai", "Jaguar", "Landrover", "Lexus", "Mitsubishi",
"Mazda", "Nissan", "Renault", "Seat", "Senova", "Suzuki", "Toyota", "Volkswagen"};
const Words countries = {"Albania", "Afghanistan", "Russia", "Ukrain", "France",
  "Japan", "China", "South_Korea", "North_Korea", "India", "Moldova", "Belarus",
  "USA", "Canada", "Brazilia", "UK", "Spain", "Italy", "Norway", "Denmark",
  "Portugal", "Czech", "Litva", "Latvia", "Egypt", "Tunis", "Turkey", "Germany",
  "Austria", "MOTHERLAND"};

CarData getRandomCarData() {
  std :: random_device rd;
  std :: mt19937 gen(rd());
  std :: uniform_int_distribution <int> dist (0,
      std :: min(model_names. size () - 1, countries . size () - 1));
  return {model_names[ dist (gen)],
    countries [ dist (gen)],
    1700 + dist (gen)* dist (gen) + dist (gen)};
}

void debugList ( const CarList & list ) {
  list . printList ();
  std :: cout << std :: endl ;
}
// Tester main
/*
int main() {
  constexpr size_t size = 15;
  std :: random_device rd;
  std :: mt19937 gen(rd());
  std :: uniform_int_distribution <int> dist (0, size );
  for ( int step = 1; step < 3; ++step) {
    std :: cout << step << " test \n";
    CarList list ;
    for ( size_t i = 0; i < size ; ++i) {
      list . push_front (getRandomCarData());
    }
    list . sort ();
    ModelList mlist = list . createModelList ();
    int id = dist (gen);
    auto ptr = mlist . getFirstNode ( list [id ]. car_model );
    std :: string car_model = ptr ->data . car_model ;
```

```
    std :: cout << "All elements with car model = " << car_model << '\n';
    for (; ptr && ptr->data.car_model == car_model; ptr = ptr->next) {
      std :: cout << ptr->data << std :: endl;
    }
    int id2 = dist(gen);
    car_model = list[id2].car_model;
    list.insert_model({car_model, "IMPOSTER", 2000});
    std :: cout << "List with inserted IMPOSTER with car_model = " << car_model
      << '\n';
    debugList(list);
    int id3 = dist(gen);
    car_model = list[id3].car_model;
    std :: cout << "List without models with name = " << car_model
      << '\n';
    list.delete_model(car_model);
    debugList(list);
  }
  return 0;
}
*/
// Menu main
int main() {
  int number_of_operation;
  CarList* list = nullptr;
  ModelList model_list;
  std :: string model_name, country;
  int prod_year;
  std :: cout << "Welcome to my program 'Auto model linked list'!";
  while (true) {
    std :: cout << "Choose your next operation :\n"
      << "1 - create a new list \n"
        << "2 - add element at the end of the list (parameters : model name, country, production year)\n
        << "3 - add element at the beginning of the list (parameters : model name, country, production
        << "4 - print list in normal order \n"
        << "5 - print list in reverse order \n"
        << "6 - delete last list element \n"
        << "7 - print first element with specific model name\n"
        << "8 - sort list \n"
        << "Next operations will work correctly only if the list is sorted :\n"
        << "9 - add element by its model name (parameters : model name, country, production year)\n"
        << "10 - create models list based on the current list \n"
        << "11 - delete all models from the list (parameter - model name)\n";
    std :: cin >> number_of_operation;

    switch (number_of_operation) {
      case 1:
        list = new CarList();
        break;
```

```
        case 2:
            std :: cin >> model_name >> country >> prod_year ;
            list != nullptr ?
                list ->push_back({model_name, country , prod_year}) : void();
            break;
        case 3:
            std :: cin >> model_name >> country >> prod_year ;
            list != nullptr ?
                list -> push_front ({model_name, country , prod_year}) : void();
            break;
        case 4:
            list != nullptr ? list -> printList () : void();
            break;
        case 5:
            list != nullptr ? list -> reversePrintList () : void();
            break;
        case 6:
            list != nullptr ? list ->pop_back() : void();
            break;
        case 7:
            std :: cin >> model_name;
            if ( list != nullptr ) {
                auto ptr = list ->searchByModel(model_name);
                std :: cout << "Element: " << ptr ->data << '\n';
            }
            break;
        case 8:
            list != nullptr ? list -> sort () : void();
            break;
        case 9:
            std :: cin >> model_name >> country >> prod_year ;
            list != nullptr ?
                list -> insert_model ({model_name, country , prod_year}) : void();
            break;
        case 10:
            std :: cin >> model_name;
            model_list = list != nullptr ?
                list -> createModelList () : ModelList();
            break;
        case 11:
            std :: cin >> model_name;
            list != nullptr ? list -> delete_model (model_name) : void();
    }
    std :: cout << '\n';
  }
  std :: cout << "See you next time!" << std :: endl ;
  return 0;
}
```

1.4. Результаты тестирования

```
1 test
All elements with car model = Landwind
Landwind Germany 2298
List with inserted IMPOSTER with car_model = Seat
Acura Portugal 1751
Acura Ukrain 2088
Bugatti China 1726
Cherry South_Korea 1719
Chevrolet North_Korea 1728
Citroen Russia 2084
Dodge Moldova 1800
Honda Brazilia 1838
Hyondai North_Korea 2099
Jaguar France 1956
Landwind Germany 2298
Mazda Egypt 1984
Nissan South_Korea 1919
Renault UK 1801
Seat Egypt 1800
Seat IMPOSTER 2000

List without models with name = Dodge
Acura Portugal 1751
Acura Ukrain 2088
Bugatti China 1726
Cherry South_Korea 1719
Chevrolet North_Korea 1728
Citroen Russia 2084
Honda Brazilia 1838
Hyondai North_Korea 2099
Jaguar France 1956
Landwind Germany 2298
Mazda Egypt 1984
Nissan South_Korea 1919
Renault UK 1801
Seat Egypt 1800
Seat IMPOSTER 2000
```

Рис. 7 - Результаты тестирования программы 1 тест

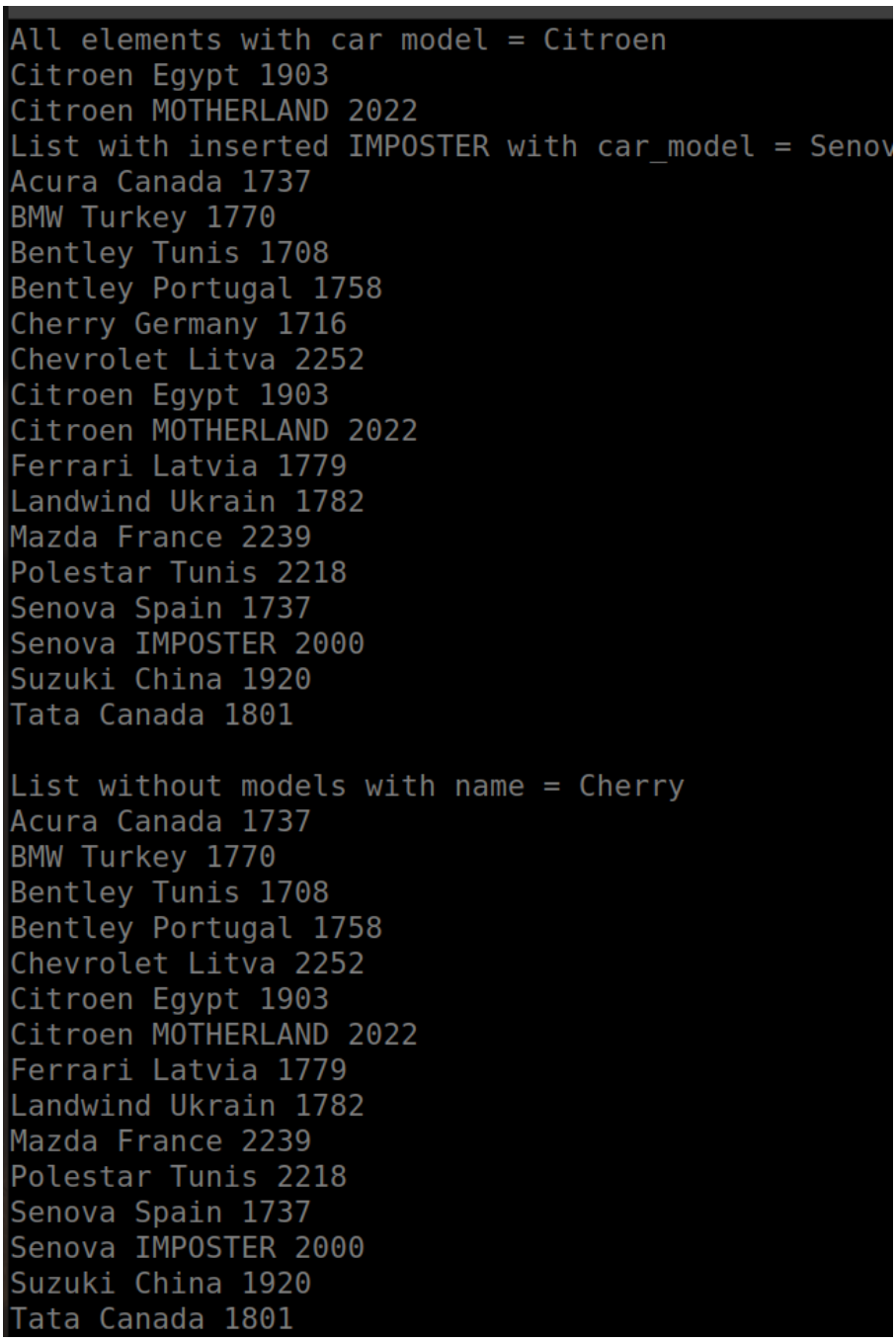

Рис. 8 - Результаты тестирования программы 2 тест

1.5. Сложность первой дополнительной операции (вставка в список)

1. На линейном динамическом списке $T(n) = \Theta(n)$ т.к. для вставки необходимо получить нужный адрес элемента, что занимает $\Theta(n)$ времени в среднем случае.

2. На одномерном массиве также $T(n) = \Theta(n)$ т.к. для вставки все равно необходимо получить адрес элемента, перед которым необходимо вставлять, что занимает $\Theta(n)$ в среднем случае.

Выводы

В ходе выполнения работы была реализована структура «двунаправленный список», поддерживающий следующие обязательные операции: вывод хранящихся значений узлов списка в прямом и обратном порядках; поиск первого элемента в списке по заданной модели; вставка нового узла перед узлом, год выпуска которого меньше; создание списка с "модельными" узлами; удаление всех элементов с заданным названием модели. Также в ходе выполнения работы были усвоены основы работы с двунаправленными списками. Тестирование подтвердило правильность работы методов.

Список информационных источников

1. Thomas H. Cormen, Clifford Stein и другие: Introduction to Algorithms, 3rd Edition. Сентябрь 2009. The MIT Press.

2. N. Wirth: Algorithms and Data Structures. Август 2004. https://people.inf.ethz.ch/wirth/AD.pdf.

3. Linked list // Wikipedia [Электронный ресурс]. URL: https://en.wikipedia.org/wiki/Linked_list (Дата обращения: 02.05.2021)

4. Курс Algorithms, part 2 // Coursera [Электронный ресурс]. URL: https://www.coursera.org/learn/algorithms-part2 (Дата обращения: 02.05.2021)