**МИНОБРНАУКИ РОССИИ**

**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**

**«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)**

**Кафедра АПУ**

**ОТЧЕТ**

**по лабораторной работе №4**

**по дисциплине «Программирование»**

| Студент гр. 3391 | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | Николаев В.Ю. |
| --- | --- | --- |
| Преподаватель | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | Писарев А.С. |

Санкт-Петербург

2024 г.

# Исходная формулировка

Необходимо реализовать программу обеспечивающую обработку линейного двусвязного списка. Узлы списка должны хранить информацию о городах. В состав этой информации входят:

* название города;
* название региона;
* количество жителей.

Над списком требуется произвести следующие операции:

1. построение и заполнение узлов списка данными с клавиатуры;
2. вывести на экран названия регионов в порядке убывания суммарной численности городского населения;
3. удалить узлы, хранящие информацию о городах заданного (с клавиатуры) региона;
4. очистить список.

# Ход работы программы

В начале работы программы создаётся список.

| 5 | List<City> cities; |
| --- | --- |

Затем в консоль выводится запрос о вводе списка городов.

| 7  8  9 | std::cout << "Enter list of cities" << std::endl;  Utils::enterListOfCities(cities);  std::cout << std::endl; |
| --- | --- |

После этого выводится введённый пользователем список.

| 11  12  13 | std::cout << "List of cities" << std::endl;  Utils::printListOfCities(cities);  std::cout << std::endl; |
| --- | --- |

Затем выводятся на экран названия регионов в порядке убывания суммарной численности городского населения.

| 15  16 | Utils::printRegionsInDescendingOrderOfPopulation(cities);  std::cout << std::endl; |
| --- | --- |

После этого удаляются узлы, хранящие информацию о городах заданного (с клавиатуры) региона и выводится новый список всех городов.

| 18  19  20  21 | Utils::removeCitiesFromSpecifiedRegion(cities);  std::cout << "List of cities after removing cities from specified region" << std::endl;  Utils::printListOfCities(cities);  std::cout << std::endl; |
| --- | --- |

Затем все удаляются все элементы из списка и выводится пустой список.

| 23  24  25  26 | Utils::clearCityList(cities);  std::cout << "List of cities after clearing" << std::endl;  Utils::printListOfCities(cities);  std::cout << std::endl; |
| --- | --- |

# Текст программы

| Main.cpp | |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | #include "Utils.h"  int main()  {  List<City> cities;    std::cout << "Enter list of cities" << std::endl;  Utils::enterListOfCities(cities);  std::cout << std::endl;  std::cout << "List of cities" << std::endl;  Utils::printListOfCities(cities);  std::cout << std::endl;    Utils::printRegionsInDescendingOrderOfPopulation(cities);  std::cout << std::endl;  Utils::removeCitiesFromSpecifiedRegion(cities);  std::cout << "List of cities after removing cities from specified region" << std::endl;  Utils::printListOfCities(cities);  std::cout << std::endl;  Utils::clearCityList(cities);  std::cout << "List of cities after clearing" << std::endl;  Utils::printListOfCities(cities);  std::cout << std::endl;  return 0;  } |
| Utils.h | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63 | #pragma once  #include "List.h"  #include "City.h"  #include <ostream>  #include <string>  class Utils  {  public:  static void enterListOfChars(List<char>& list)  {  std::cin.unsetf(std::ios::skipws);  char temp;  while (std::cin >> temp && temp == '\n');  do  list.push\_back(temp);  while (std::cin >> temp && temp != '\n');  std::cin.setf(std::ios::skipws);  }  static void enterCity(City& city)  {  std::cout << "Enter city: ";  enterListOfChars(city.city);  std::cout << "Enter region: ";  enterListOfChars(city.region);  std::cout << "Enter population: ";  std::cin >> city.population;  }  static void enterListOfCities(List<City>& cities)  {  int n;  std::cout << "Enter number of cities: ";  std::cin >> n;  for (int i = 0; i < n; i++)  {  City city;  enterCity(city);  cities.push\_back(city);  }  }  static void printCity(City& city)  {  for (int i = 0; i < city.city.size(); i++)  std::cout << city.city[i];  std::cout << ", ";  for (int i = 0; i < city.region.size(); i++)  std::cout << city.region[i];  std::cout << ", " << city.population << std::endl;  }  static void printListOfCities(List<City>& cities)  {  for (int i = 0; i < cities.size(); i++)  printCity(cities[i]);  }  static void printRegionsInDescendingOrderOfPopulation(List<City>& cities)  {  List<List<char>> regions;  List<int> populations;  for (int i = 0; i < cities.size(); i++)  {  bool found = false;  for (int j = 0; j < regions.size(); j++)  {  if (cities[i].region == regions[j])  {  populations[j] += cities[i].population;  found = true;  break;  }  }  if (!found)  {  regions.push\_back(cities[i].region);  populations.push\_back(cities[i].population);  }  }  for (int i = 0; i < regions.size(); i++)  {  for (int j = i + 1; j < regions.size(); j++)  {  if (populations[i] < populations[j])  {  std::swap(regions[i], regions[j]);  std::swap(populations[i], populations[j]);  }  }  }  for (int i = 0; i < regions.size(); i++)  {  for (int j = 0; j < regions[i].size(); j++)  std::cout << regions[i][j];  std::cout << ", " << populations[i] << std::endl;  }  }  static void removeCitiesFromSpecifiedRegion(List<City>& cities)  {  List<char> region;  std::cout << "Enter region to remove cities from: ";  enterListOfChars(region);  for (int i = 0; i < cities.size(); i++)  {  if (cities[i].region == region)  {  cities.erase(i);  i--;  }  }  }  static void clearCityList(List<City>& cities)  {  cities.clear();  }  }; |
| City.h | |
| 12345678910 | #pragma once  #include <iostream>  #include "List.h"  struct City  {  List<char> city;  List<char> region;  int population;  }; |
| List.h | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137  138  139  140  141  142  143  144  145  146  147  148  149  150  151  152  153  154  155  156  157  158  159  160  161  162  163  164  165  166  167  168  169  170  171  172  173  174  175  176  177  178  179  180  181  182  183  184  185  186  187  188  189  190  191  192  193  194  195  196  197  198  199  200  201  202  203  204  205  206  207  208  209  210  211  212  213  214  215  216  217  218  219  220  221  222  223  224  225  226  227  228  229  230  231  232  233  234  235  236  237  238  239  240  241  242  243  244  245  246  247  248  249  250  251  252  253  254  255  256  257  258  259  260  261  262  263  264  265  266  267  268  269  270  271  272  273  274  275  276  277  278  279  280  281  282 | #pragma once  #include <iostream>  template <typename T>  class List  {  private:  struct Node  {  T data;  Node\* next;  Node\* prev;  Node(const T& data): data(data), next(nullptr), prev(nullptr) {}  ~Node() {}  Node(const Node& other) = delete;  Node& operator=(const Node& other) = delete;  };  Node\* head;  Node\* tail;  int length;  public:  List(): head(nullptr), tail(nullptr), length(0) {}  void push\_back(const T& data)  {  Node\* newNode = new Node(data);  if (head == nullptr)  {  head = newNode;  tail = newNode;  }  else  {  tail->next = newNode;  newNode->prev = tail;  tail = newNode;  }  length++;  }  void push\_front(const T& data)  {  Node\* newNode = new Node(data);  if (head == nullptr)  {  head = newNode;  tail = newNode;  }  else  {  newNode->next = head;  head->prev = newNode;  head = newNode;  }  length++;  }    void pop\_back()  {  if (empty())  {  return;  }  Node\* temp = tail;  tail = tail->prev;  delete temp;  if (tail == nullptr)  {  head = nullptr;  }  else  {  tail->next = nullptr;  }  length--;  }  void pop\_front()  {  if (empty())  {  return;  }  Node\* temp = head;  head = head->next;  delete temp;  if (head == nullptr)  {  tail = nullptr;  }  else  {  head->prev = nullptr;  }  length--;  }  void insert(const T& data, int index)  {  if (index < 0 || index > length)  {  return;  }  if (index == 0)  {  push\_front(data);  return;  }  if (index == length)  {  push\_back(data);  return;  }  Node\* newNode = new Node(data);  Node\* temp = head;  for (int i = 0; i < index - 1; i++)  {  temp = temp->next;  }  newNode->next = temp->next;  newNode->prev = temp;  temp->next->prev = newNode;  temp->next = newNode;  length++;  }  void erase(int index)  {  if (index < 0 || index >= length)  {  return;  }  if (index == 0)  {  pop\_front();  return;  }  if (index == length - 1)  {  pop\_back();  return;  }  Node\* temp = head;  for (int i = 0; i < index; i++)  {  temp = temp->next;  }  temp->prev->next = temp->next;  temp->next->prev = temp->prev;  delete temp;  length--;  }  T& operator[](int index)  {  if (index < 0 || index >= length)  {  std::cout << "Invalid index" << std::endl;  exit(1);  }  Node\* temp = head;  for (int i = 0; i < index; i++)  {  temp = temp->next;  }  return temp->data;  }  void clear()  {  Node\* current = head;  while (current != nullptr)  {  Node\* next = current->next;  delete current;  current = next;  }  head = nullptr;  tail = nullptr;  length = 0;  }    int size() const  {  return length;  }  bool empty() const  {  return length == 0;  }  ~List()  {  clear();  }    bool operator==(const List<T>& other) const  {  if (length != other.length)  {  return false;  }  Node\* current = head;  Node\* otherCurrent = other.head;  while (current != nullptr)  {  if (current->data != otherCurrent->data)  {  return false;  }  current = current->next;  otherCurrent = otherCurrent->next;  }  return true;  }  bool operator!=(const List<T>& other) const  {  return !(\*this == other);  }  List(const List<T>& other): head(nullptr), tail(nullptr), length(0)  {  Node\* current = other.head;  while (current != nullptr)  {  push\_back(current->data);  current = current->next;  }  }  List<T>& operator=(const List<T>& other)  {  if (this != &other)  {  clear();  Node\* current = other.head;  while (current != nullptr)  {  push\_back(current->data);  current = current->next;  }  }  return \*this;  }  }; |

# Результаты работы программы

