|  |
| --- |
| МИНОБРНАУКИ РОССИИ |
| Федеральное государственное бюджетное образовательное учреждение  высшего образования |
| **«МИРЭА – Российский технологический университет»** |
| **РТУ МИРЭА** |
|  |

| **Отчет по выполнению практического задания № 4** | |
| --- | --- |
| **Тема:** | |
| **«Алгоритмы внешних сортировок»** | |
| Дисциплина: «Структуры и алгоритмы обработки данных» | |
|  | Выполнил студент: Балыкова М.А. |
|  | Группа: ИКБО-74-23 |

Москва – 2024

СОДЕРЖАНИЕ

[1 ЦЕЛЬ 3](#_gjdgxs)

[2 ЗАДАНИЕ №1 4](#_30j0zll)

[2.1 Формулировка задачи 4](#_1fob9te)

[2.2 Описание выполнения алгоритма прямого слияния и его блок-схема 4](#_2et92p0)

[2.3 Реализация алгоритма на языке C++ и проведение тестирования 6](#)

[2.4 Адаптация программы к новым данным 7](#_2s8eyo1)

[2.5 Практическая сложность алгоритма 16](#_3rdcrjn)

[2.7 Вывод по заданию №1 17](#_26in1rg)

[3 ЗАДАНИЕ №2 19](#_lnxbz9)

[3.1 Формулировка задачи 19](#_35nkun2)

[3.2 Описание выполнения естественного слияния и его блок-схема 19](#_1ksv4uv)

[3.3 Реализация алгоритма на языке C++ и проведение тестирования 20](#_2jxsxqh)

[3.4 Адаптация программы к новым данным 22](#_1y810tw)

[3.5 Практическая сложность алгоритма 33](#_2xcytpi)

[3.6 Вывод по заданию №2 34](#_1ci93xb)

[4 ВЫВОДЫ 35](#_3whwml4)

[5 ЛИТЕРАТУРА 36](#_2bn6wsx)

# **1 ЦЕЛЬ**

Освоить приёмы сортировки данных из файлов

# **2 ЗАДАНИЕ №1**

## **2.1 Формулировка задачи**

(В списке №4, Вариант 4)

Разработать программу и применить алгоритм внешней сортировки прямого слияния к сортировке файла данных индивидуального варианта.

1) Реализовать функцию сортировки (возможно, с вспомогательными функциями) и основную подпрограмму main.

2) Отладить программу, протестировать на примере из п.2.

3) Предварительно подготовить файл данных в соответствии с вариантом (не менее 32 записей).

4) Адаптировать программу для сортировки файла с записями, протестировать на подготовленном ранее файле.

5) Определить практическую сложность алгоритма для файлов с увеличивающимся количеством записей (8, 16, 32). Сформировать таблицу результатов, указав количество записей и время сортировки.

Индивидуальный вариант: Сведения о жителе: Фамилия, Город, Адрес: улица, дом, квартира.

## **2.2 Описание выполнения алгоритма прямого слияния и его блок-схема**

Алгоритм прямого слияния (также известный как метод сортировки слиянием) является классическим алгоритмом сортировки, который основан на принципе разделения и слияния. Его основная идея заключается в том, что большие массивы разбиваются на меньшие подмассивы, которые затем сортируются независимо друг от друга, а затем объединяются в один отсортированный массив.

Процесс выполнения алгоритма прямого слияния можно описать следующим образом:

1. Разделение: Исходный массив делится на две половины (или более), пока не останется подмассивы размером в один элемент.

2. Сортировка: Каждый подмассив сортируется отдельно, например, с использованием рекурсивного вызова алгоритма.

3. Слияние: Отсортированные подмассивы объединяются в один отсортированный массив путем сравнения элементов и помещения их в правильном порядке.

4. Повторение: Процесс разделения, сортировки и слияния повторяется, пока все подмассивы не будут объединены в один отсортированный массив.

Алгоритм прямого слияния имеет временную сложность O(n log n) в наихудшем и среднем случаях, что делает его эффективным для сортировки больших массивов данных. Однако, его недостатком является то, что он требует дополнительной памяти для хранения временных подмассивов, что может быть проблематично для больших объемов данных.

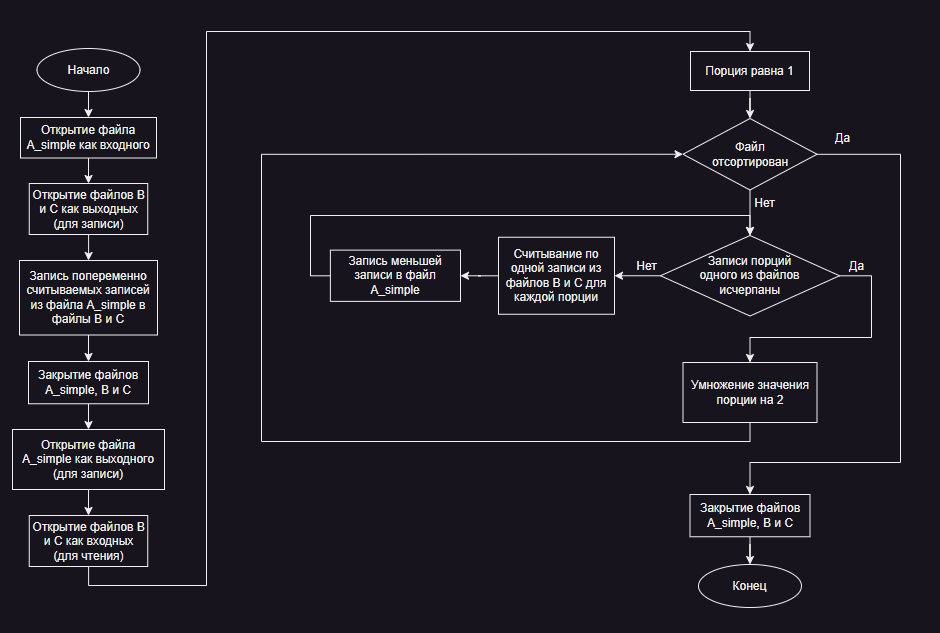
Реализация данного алгоритма будет предоставлена на рисунке 1.   


Рисунок 1 – Блок-схема алгоритма прямого слияния

## **2.3 Реализация алгоритма на языке C++ и проведение тестирования**

Пример:  
 Пусть файл А содержит данные, которые подлежат сортировке:

8 2 13 4 15 6 9 11 3 7 5 10 1 12 14

Для реализации алгоритма будем использовать два файла, в которые будем разливать данные файла: файл В и файл С.

Сначала разбиваем по одному элементу:

B: 8 13 15 9 3 5 1 14

C: 2 4 6 11 7 10 12

Сливаем в упорядоченные двойки:

А: 2 8 4 13 6 15 9 11 3 7 5 10 1 12 14

Разливаем по два:

B: 2 8 6 15 3 7 1 12

C: 4 13 9 11 5 10 14

Сливаем в упорядоченные четверки и т. д. пока длина порции не станет равной длине массива.

Реализация данного примера будет предоставлена в блоке кода 1.

Результаты тестирования предоставим на рисунке 2.

| #include <iostream> #include <vector> #include <algorithm> using namespace std;  void mergeSort(vector<int>& arr) {  if (arr.size() <= 1)  return;   int mid = arr.size() / 2;  vector<int> left(arr.begin(), arr.begin() + mid);  vector<int> right(arr.begin() + mid, arr.end());   mergeSort(left);  mergeSort(right);   int i = 0, j = 0, k = 0;  while (i < left.size() && j < right.size()) {  if (left[i] <= right[j])  arr[k++] = left[i++];  else  arr[k++] = right[j++];  }   while (i < left.size())  arr[k++] = left[i++];   while (j < right.size())  arr[k++] = right[j++]; }  int main() {  vector<int> arr = { 8, 2, 13, 4, 15, 6, 9, 11, 3, 7, 5, 10, 1, 12, 14 };   mergeSort(arr);   for (int num : arr)  cout << num << ' ';   return 0; } |
| --- |

Блок кода 1 – Программа прямого слияния



Рисунок 2 - Тестирование программы с входными данными из примера

## **2.4 Адаптация программы к новым данным**

Адаптируем программу к данным по условию индивидуального варианта. Для этой цели будет использован пользовательский заголовочный файл "p.h"(блок кода 2) и main.cpp(блок кода 3).

| #pragma once #include <iostream> #include <fstream> #include <algorithm> #include <vector> #include <string> #include <functional> using namespace std;  template<class SType, char DSep = ' '> class SExternalMergeSort {  public:  explicit SExternalMergeSort(const string& filename, unsigned int way\_count = 2) : seriesLength\_{ 0 }, wayCount\_{ way\_count }, inputFilename\_{ filename }  {  if (wayCount\_ < 2) wayCount\_ = 2;  };   virtual ~SExternalMergeSort() = default;   template<class Compare>  void sort(fstream& input, Compare&& cmp = [](const SType& a, const SType& b) { return a < b; });  private:  void computeSeriesLength(fstream& input);  string getExternalFilename(const unsigned int way);  vector<fstream> getExternalFiles();  void reopenExternalFiles(vector<fstream>& ext, ios\_base::openmode mode);  void removeExternalFiles();  void reopenInputFile(fstream& input, ios\_base::openmode mode);   void split(fstream& input, vector<fstream>& ext, const unsigned int subgroup\_length);  template<class Compare>  void merge(fstream& input, vector<fstream>& ext, const unsigned int subgroup\_length, Compare& cmp);   unsigned int seriesLength\_;   unsigned int wayCount\_;  string inputFilename\_; };  template<class SType, char DSep> void SExternalMergeSort<SType, DSep>::computeSeriesLength(fstream& input\_series) {  this->seriesLength\_ = 0;  SType value{};   while (input\_series >> value)  {  this->seriesLength\_++;  } }  template<class SType, char DSep> string SExternalMergeSort<SType, DSep>::getExternalFilename(const unsigned int way) {  return "/" + to\_string(way + 1); }  template<class SType, char DSep> vector<fstream> SExternalMergeSort<SType, DSep>::getExternalFiles() {  vector<fstream> ext(this->wayCount\_);   for (unsigned int i = 0; i < this->wayCount\_; ++i)  {  const auto filename = this->getExternalFilename(i);  ofstream f(filename);  ext[i] = fstream(filename);  }  return ext; }  template<class SType, char DSep> void SExternalMergeSort<SType, DSep>::reopenExternalFiles(vector<fstream>& ext, ios\_base::openmode mode) {  for (int i = 0; i < ext.size(); ++i)  {  ext[i].close();   const auto filename = getExternalFilename(i);  ext[i].open(filename, mode);  } }  template<class SType, char DSep> void SExternalMergeSort<SType, DSep>::removeExternalFiles() {  for (unsigned int i = 0; i < this->wayCount\_; ++i)  {  const auto filename = getExternalFilename(i);  const auto removeResult = remove(filename.c\_str());  cout << "Remove external file (" << filename << "): " << boolalpha << removeResult << endl;  } }  template<class SType, char DSep> void SExternalMergeSort<SType, DSep>::reopenInputFile(fstream& input, ios\_base::openmode mode) {  input.close();  input.open(this->inputFilename\_, mode); }  template<class SType, char DSep> void SExternalMergeSort<SType, DSep>::split(fstream& input\_series, vector<fstream>& ext, const unsigned int subgroup\_length) {  while (!input\_series.eof())  {  for (auto&& f : ext)  {  SType value{};  unsigned int step = 0;  while (input\_series >> value)  {  f << value << DSep;  cout << value << "\t";  ++step;  if (step == subgroup\_length) break;  }  cout << " | ";  }  cout << endl;  } }  template<class SType, char DSep> template<class Compare> void SExternalMergeSort<SType, DSep>::merge(  fstream& input\_series, vector<fstream>& ext,  const unsigned int subgroup\_length,  Compare& cmp) {  if (subgroup\_length == 0) return;   vector<SType> buf;  buf.reserve(subgroup\_length \* this->wayCount\_);   const unsigned int fullGroupCount = this->seriesLength\_ / subgroup\_length / this->wayCount\_;   const unsigned int add = (this->seriesLength\_ / subgroup\_length) % this->wayCount\_ == 0 ? 0 : 1;   const unsigned int stepsCount = fullGroupCount + add;  unsigned int step = 0;   while (step < stepsCount)  {  for (auto& f : ext)  {  SType value{};  unsigned int substep = 0;  while (substep < subgroup\_length && f >> value)  {  buf.push\_back(value);  ++substep;  }  }   std::sort(begin(buf), end(buf), cmp);   const auto bufLength = buf.size();   for (int i = 0; i < bufLength; ++i)  {  input\_series << buf[i] << DSep;  cout << buf[i] << DSep;  }  cout << "\t";   buf.clear();  buf.resize(0);  step++;  }  cout << endl; }  template<class SType, char DSep> template<class Compare> void SExternalMergeSort<SType, DSep>::sort(fstream& input\_series, Compare&& cmp) {  this->computeSeriesLength(input\_series);   if (this->seriesLength\_ <= 1)  {  return;  }   auto&& cmp\_ = forward<Compare>(cmp);  vector<fstream> ext = this->getExternalFiles();   for (unsigned int subgroup\_length = 1; subgroup\_length < this->seriesLength\_; subgroup\_length \*= 2)  {  reopenInputFile(input\_series, ios\_base::in);  reopenExternalFiles(ext, ios\_base::out);   cout << "Subgroup with length " << to\_string(subgroup\_length) << ":" << endl;  this->split(input\_series, ext, subgroup\_length);   reopenInputFile(input\_series, ios\_base::out);  reopenExternalFiles(ext, ios\_base::in);   cout << "Merge to groups with length " << to\_string(subgroup\_length) << endl;  this->merge(input\_series, ext, subgroup\_length, cmp\_);  }   input\_series.flush();  this->removeExternalFiles(); } |
| --- |

Блок кода 2 - заголовочный файл "prost.h"

| #include <fstream> #include <vector> #include "p.h" #include <string> #include <sstream> using namespace std;  // Функция разделения  vector<string>& split(const string& s, char delim, vector<string>& elems) {  stringstream ss(s);  string item;  while (getline(ss, item, delim))  {  elems.push\_back(item);  }  return elems; }  // структура файла struct Consumer {  string lastname;  string city;  string street;  string houseNumber;  string apartmentNumber; };  istream& operator>>(istream& lhv, Consumer& rhv) {  string tmp;  getline(lhv, tmp);   vector<string> e;  split(tmp, ',', e);  const auto elementsCount = e.size();   if (elementsCount >= 5)  {  rhv.lastname = e[0];  rhv.city = e[1];  rhv.street = e[2];  rhv.houseNumber = e[3];  rhv.apartmentNumber = e[4];  }   return lhv; }  ostream& operator<<(ostream& lhv, Consumer& rhv) {  lhv << rhv.lastname << "," << rhv.city << "," << rhv.street << "," << rhv.houseNumber << "," << rhv.apartmentNumber;  return lhv; }  int main(int argc, char\*\* argv) {  //выбирает категорию классификации символов языка С  setlocale(LC\_CTYPE, "Rus");   const auto input\_file = "A";  fstream input(input\_file, ios\_base::in | ios\_base::out);   if (!input)  {  perror("Ошибка");  return EXIT\_FAILURE;  }   SExternalMergeSort<Consumer, '\n'>(input\_file).sort(input, [](const Consumer& a, const Consumer& b) { return a.lastname < b.lastname; });  input.close();  system("Пауза");  //успешное выполнение программы  return EXIT\_SUCCESS; } |
| --- |

Блок кода 3 - Файл main.cpp

Протестируем адаптированный код. Результаты тестирования будут продемонстрированы на рисунке 3-6.



Рисунок 3 - Тестирование адаптированной программы

****

Рисунок 4 - Тестирование адаптированной программы



Рисунок 5 - Тестирование адаптированной программы



Рисунок 6 - Тестирование адаптированной программы

## **2.5 Практическая сложность алгоритма**

Сложность алгоритма прямого слияния зависит от количества элементов, которые необходимо отсортировать. Общая сложность данного алгоритма составляет O(n log n), где n - количество элементов в последовательности.

В лучшем случае сложность алгоритма также будет O(n log n), так как на каждом уровне рекурсии необходимо выполнить слияние n элементов.

В худшем случае, сложность алгоритма также будет O(n log n), так как даже если последовательность уже отсортирована, все равно будут производиться рекурсивные вызовы для разделения и слияния элементов.

Таким образом, практическая сложность алгоритма прямого слияния составляет O(n log n) в среднем и в худшем случае.

Продемонстрируем подсчёт времени на разном количестве записей в файле данных в таблице 1.

Таблица 1 - Сводная таблица результатов

| **Количество записей** | **Время(мкс)** |
| --- | --- |
| 8 | 25732 |
| 16 | 475933 |
| 32 | 2156493 |

## **2.7 Вывод по заданию №1**

На основе выполнения, сложности и эффективности алгоритма прямого слияния можно сделать следующие выводы:

1. Алгоритм прямого слияния эффективен при сортировке больших массивов данных, так как его сложность составляет O(n log n). Это позволяет быстро и эффективно сортировать большие объемы информации, что делает его привлекательным для работы с большими наборами данных.

2. Алгоритм прямого слияния легко реализуется и понимается, что делает его доступным для широкого круга разработчиков. Он прост в освоении и может быть применен для решения различных задач сортировки.

3. Несмотря на то, что алгоритм прямого слияния имеет сложность O(n log n), у него есть недостатки: он требует дополнительной памяти для хранения временных массивов при слиянии. Это может быть проблемой при работе с очень большими массивами данных, так как потребление памяти может быть значительным.

В целом, алгоритм прямого слияния является эффективным и надежным методом сортировки, который часто используется для сортировки больших объемов данных. Он имеет хорошую сложность и отличается простотой реализации, что делает его привлекательным выбором для многих приложений. Однако, необходимо учитывать использование дополнительной памяти при его применении.

# **3 ЗАДАНИЕ №2**

## **3.1 Формулировка задачи**

Разработать программу и применить алгоритм сортировки естественного слияния к сортировке файла с данными варианта (файл уже должен быть подготовлен в задании 1).

1) Реализовать функцию сортировки (возможно, с вспомогательными функциями) и основную подпрограмму main.

2) Отладить программу, протестировать на примере из п.4.

3) Адаптировать программу для сортировки файла с записями, протестировать на подготовленном ранее файле.

4) Сформировать таблицу результатов, указав количество записей и время сортировки.

## **3.2 Описание выполнения естественного слияния и его блок-схема**

Алгоритм естественного слияния (natural merge sort) является вариантом алгоритма прямого слияния, который основан на идее объединения уже отсортированных последовательностей.

Вот описание выполнения алгоритма естественного слияния:

1. На первом этапе алгоритма мы разбиваем исходный массив на отсортированные подпоследовательности. Для этого проходим по массиву и находим все упорядоченные подмассивы.

2. Затем мы объединяем эти подпоследовательности парами, начиная с первых двух подпоследовательностей, затем следующих двух и т.д. Если длины объединяемых подпоследовательностей увеличиваются, мы продолжаем объединять их пока не получим один отсортированный массив.

3. Повторяем шаг 2 до тех пор, пока не получим сортированный массив.

Для выполнения алгоритма естественного слияния не требуется дополнительной памяти для временных массивов, так как мы используем уже отсортированные подпоследовательности для их последующего слияния.

Сложность выполнения алгоритма естественного слияния составляет O(n log n) в среднем и в худшем случае, где n - количество элементов в массиве. Алгоритм эффективен на отсортированных или частично отсортированных данных, но может быть менее эффективным на случайных данных из-за наличия длинных упорядоченных подпоследовательностей.

Таким образом, алгоритм естественного слияния представляет собой улучшенную версию алгоритма прямого слияния, который позволяет эффективно сортировать данные, не требуя дополнительной памяти для слияния.

Реализация данного описания выполнения алгоритма представлена в виде блок-схемы (рис.7).

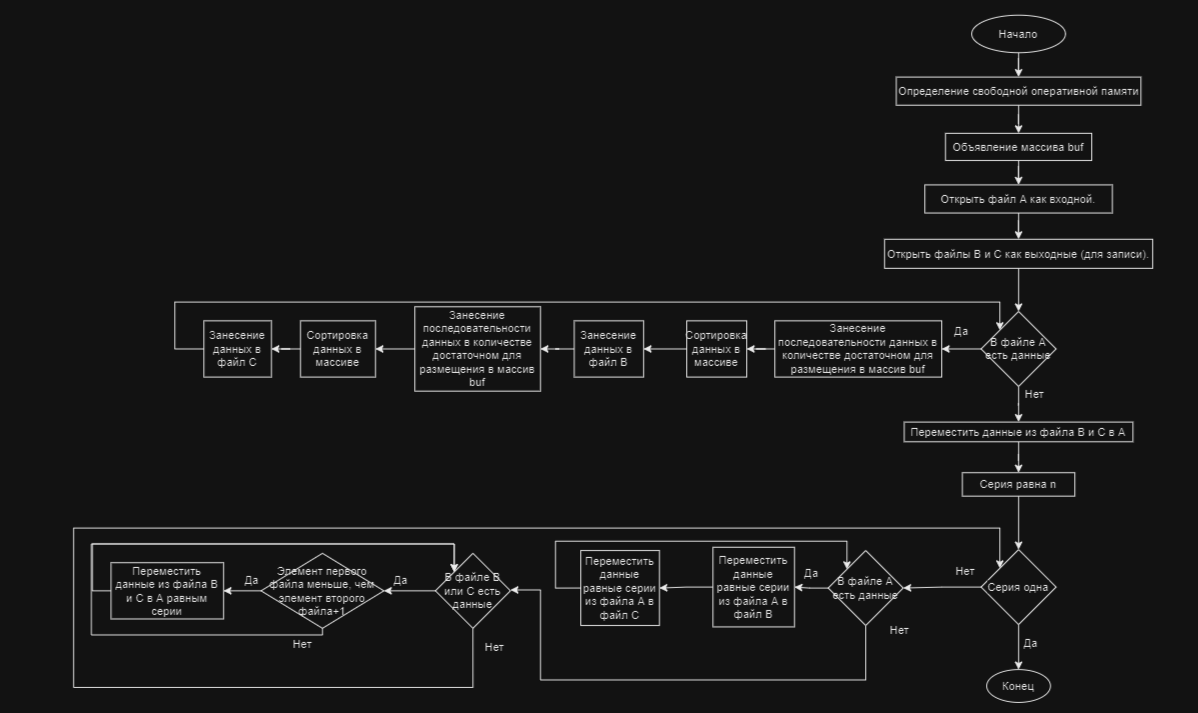


Рисунок 7 – Блок-схема алгоритма естественного слияния

## **3.3 Реализация алгоритма на языке C++ и проведение тестирования**

Пусть есть файл А, содержащий записи с ключами:

17 31 5 59 13 41 43 67 11 23 29 47 3 7 71 2 19 57 37 61

Выделим серии, завершая запятой, чтобы было нагляднее:

17 31’ 5 59’ 13 41 43 67’ 11 23 29 47’ 3 7 71’ 2 19 57’ 37 61

Получилось 7 серий.

Разделим файл на два файла В и С, переписывая в них поочередно по серии:

B: 17 31’ 13 41 43 67’ 3 7 71’ 37 61

C: 5 59’ 11 23 29 47’ 2 19 57

Сольем файлы в файл А, сливая серии в упорядоченные серии

А:5 17 31 59’ 11 13 23 29 41 43 47 67’2 3 7 19 57 71’37 61

Опять разольем в В и С поочередно переписывая серии

B: 5 17 31 59’2 3 7 19 57 71

C: 11 13 23 29 41 43 47 67’37 61

Сливаем в файл А по сериям

А: 5 11 13 17 23 29 31 41 43 47 59 67’2 3 7 19 37 57 61 71

Разливаем …..

и продолжаем до тех пор, пока в массив А не будет переписана серия длины n.

Реализация данного примера будет предоставлена в блоке кода 4.

Результаты тестирования предоставим на рисунке 8.

| #include <fstream> #include <iostream> #include <vector> #include <algorithm> #include <chrono> using namespace std; void naturalMergeSort(vector<int>& arr) {  if (arr.size() <= 1)  return;   vector<int> temp(arr.size());  for (int width = 1; width < arr.size(); width \*= 2) {  for (int i = 0; i < arr.size(); i += 2 \* width) {  auto middle = i + width < arr.size() ? i + width : arr.size();  auto end = i + 2 \* width < arr.size() ? i + 2 \* width : arr.size();   merge(arr.begin() + i, arr.begin() + middle, arr.begin() + middle, arr.begin() + end, temp.begin() + i);  }  copy(temp.begin(), temp.end(), arr.begin());  } }  int main() {  vector<int> arr = {17, 31, 5, 59, 13, 41, 43, 67, 11, 23, 29, 47, 3, 7, 71, 2, 19, 57, 37, 61};   naturalMergeSort(arr);   for (int num : arr)  cout << num << ' ';   return 0; } |
| --- |

Блок кода 4 - Естественное слияние



Рисунок 8 - Тестирование программы с входными данными из примера

## **3.4 Адаптация программы к новым данным**

Адаптируем программу к данным по условию индивидуального варианта. Для этой цели будет использован пользовательский заголовочный файл "e.h"(блок кода 5) и main.cpp(блок кода 6).

| #include <iostream> #include <fstream> #include <algorithm> #include <vector> #include <string> using namespace std;  constexpr auto kBlockSeparator = "|";  // SType - тип сортируемых данных // DSep - разделитель данных в файле template<class SType, char DSep = ' '> class NExternalMergeSort { public:  // way\_count -- количество путей (вспомогательных файлов) сортировки  explicit NExternalMergeSort(const string& filename, unsigned int way\_count = 2) : seriesLength\_{ 0 }, wayCount\_{ way\_count }, inputFilename\_{ filename }  {  if (wayCount\_ < 2) wayCount\_ = 2;  };  virtual ~NExternalMergeSort() = default;   template<class Compare>  void sort(fstream& input, Compare&& cmp = [](const SType& a, const SType& b) { return a < b; });  private:    void computeSeriesLength(fstream& input);   string getExternalFilename(const unsigned int way);   vector<fstream> getExternalFiles();   void reopenExternalFiles(vector<fstream>& ext, ios\_base::openmode mode);   void removeExternalFiles();   void reopenInputFile(fstream& input, ios\_base::openmode mode);   bool fullEOF(vector<fstream>& ext);   template<class Compare>  bool isSorted(fstream& input, Compare& cmp);   template<class Compare>  void split(fstream& input, vector<fstream>& ext, Compare& cmp);   template<class Compare>  void merge(fstream& input, vector<fstream>& ext, Compare& cmp);   unsigned int seriesLength\_;   unsigned int wayCount\_ = 2;   string inputFilename\_; };  template<class SType, char DSep> void NExternalMergeSort<SType, DSep>::computeSeriesLength(fstream& input\_series) {  this->seriesLength\_ = 0;  SType value{};   while (input\_series >> value)  {  this->seriesLength\_++;  } }  template<class SType, char DSep> string NExternalMergeSort<SType, DSep>::getExternalFilename(const unsigned int way) {  return "/" + to\_string(way + 1); }  template<class SType, char DSep> vector<fstream> NExternalMergeSort<SType, DSep>::getExternalFiles() {  vector<fstream> ext(this->wayCount\_);   for(unsigned int i = 0; i < this->wayCount\_; ++i)  {  const auto filename = getExternalFilename(i);  ofstream f(filename);  ext[i] = fstream(filename);  }  return ext; }  template<class SType, char DSep> void NExternalMergeSort<SType, DSep>::reopenExternalFiles(vector<fstream>& ext, ios\_base::openmode mode) {  for (int i = 0; i < ext.size(); ++i)  {  ext[i].close();   const auto filename = getExternalFilename(i);  ext[i].open(filename, mode);  } }  template<class SType, char DSep> void NExternalMergeSort<SType, DSep>::removeExternalFiles() {  for (unsigned int i = 0; i < this->wayCount\_; ++i)  {  const auto filename = getExternalFilename(i);  const auto removeResult = remove(filename.c\_str());  cout << "Remove external file (" << filename << "): " << boolalpha << removeResult << endl;  } }  template<class SType, char DSep> void NExternalMergeSort<SType, DSep>::reopenInputFile(fstream& input, ios\_base::openmode mode) {  input.close();  input.open(this->inputFilename\_, mode); }  template<class SType, char DSep> bool NExternalMergeSort<SType, DSep>::fullEOF(vector<fstream>& ext) {  bool fullEOF = true;  for (auto& f : ext)  {  fullEOF &= f.eof();  if (!fullEOF) break;  }   return fullEOF; }  template<class SType, char DSep> template<class Compare> bool NExternalMergeSort<SType, DSep>::isSorted(fstream& input, Compare& cmp) {  SType value{};  input >> value;  SType oldValue = value;   bool sorted = false;  while (input >> value)  {  const auto cmp1 = cmp(oldValue, value);  const auto cmp2 = cmp(value, oldValue);  sorted = cmp1 || (cmp1 == false && cmp2 == false);   oldValue = value;  if (!sorted) break;  }   return sorted; }  template<class SType, char DSep> template<class Compare> void NExternalMergeSort<SType, DSep>::split(  fstream& input\_series, vector<fstream>& ext,  Compare& cmp  ) {  SType value{};  SType oldValue{};  unsigned int way = 0;   if (input\_series >> value)  {  ext[way] << value << DSep;  oldValue = value;  cout << value << DSep;  }   while(input\_series >> value)  {  if (!cmp(oldValue, value) && cmp(value, oldValue))  {  way = (way + 1) % wayCount\_;  }   ext[way] << value << DSep;  oldValue = value;  cout << value << DSep;  }  cout << endl; }  template<class SType, char DSep> template<class Compare> void NExternalMergeSort<SType, DSep>::merge(  fstream& input\_series, vector<fstream>& ext,  Compare& cmp ) {  while(!this->fullEOF(ext))  {  vector<SType> oldValues(this->wayCount\_);  vector<bool> eogs(this->wayCount\_, { false }); // end of group (аналог end of file)  for (unsigned int way = 0; way < this->wayCount\_; ++way)  {  SType value{};  if(ext[way] >> value) oldValues[way] = value;  else eogs[way] = true;  }   bool fullEOG = false; // флаг достижения конца групп во всех путях   // выбираем путь, с которого будем начинать слияние  while (!fullEOG)  {  int wayWithMin = -1;    for (unsigned int checkedWay = 0; checkedWay < this->wayCount\_; ++checkedWay)  {  if (eogs[checkedWay] || ext[checkedWay].eof()) continue;  wayWithMin = checkedWay;  }  if (-1 == wayWithMin) break;   for (unsigned int checkedWay = 0; checkedWay < this->wayCount\_; ++checkedWay)  {  if (eogs[checkedWay] || ext[checkedWay].eof()) continue;  if (cmp(oldValues[checkedWay], oldValues[wayWithMin])) wayWithMin = checkedWay;  }    input\_series << oldValues[wayWithMin] << DSep;  cout << oldValues[wayWithMin] << DSep;    SType value{};  if (ext[wayWithMin] >> value)   {  oldValues[wayWithMin] = value;  }  else  {  eogs[wayWithMin] = true;  ext[wayWithMin].clear();  }   fullEOG = true;  for (auto eog : eogs)  {  fullEOG &= eog;  if (!fullEOG) break;  }  }   for (unsigned int way = 0; way < this->wayCount\_; ++way)  {  if (eogs[way])   {  ext[way].get();  ext[way].get();  }  }  }   cout << endl; }  template<class SType, char DSep> template<class Compare> void NExternalMergeSort<SType, DSep>::sort(fstream& input\_series, Compare&& cmp) {   this->computeSeriesLength(input\_series);  if (this->seriesLength\_ <= 1)  {  return;  }   auto&& cmp\_ = forward<Compare>(cmp);  vector<fstream> ext = this->getExternalFiles();   reopenInputFile(input\_series, ios\_base::in);  bool sorted = isSorted(input\_series, cmp\_);  cout << "Sorted: " << boolalpha << sorted << endl;  while(!sorted)  {  reopenInputFile(input\_series, ios\_base::in);  reopenExternalFiles(ext, ios\_base::out);   cout << "Subgroup: " << endl;  this->split(input\_series, ext, cmp\_);   reopenInputFile(input\_series, ios\_base::out);  reopenExternalFiles(ext, ios\_base::in);   cout << "Merge: " << endl;  this->merge(input\_series, ext, cmp\_);   reopenInputFile(input\_series, ios\_base::in);  sorted = isSorted(input\_series, cmp\_);   cout << "Sorted: " << boolalpha << sorted << endl;  }   input\_series.flush();  this->removeExternalFiles(); } |
| --- |

Блок кода 5 - заголовочный файл "e.h"

| #include <fstream> #include <vector> #include "ems\_natural.h" #include <string> #include <sstream> using namespace std;  // Функция разделения строки по символу-разделителю vector<string>& split(const string& s, char delim, vector<string>& elems) {  stringstream ss(s);  string item;  while (getline(ss, item, delim))  {  elems.push\_back(item);  }  return elems; }  //  struct Consumer {  string lastname;  string city;  string street;  string houseNumber;  string apartmentNumber; };  istream& operator>>(istream& lhv, Consumer& rhv) {  string tmp;  getline(lhv, tmp);   vector<string> e;  split(tmp, ',', e);  const auto elementsCount = e.size();   if (elementsCount >= 5)  {  rhv.lastname = e[0];  rhv.city = e[1];  rhv.street = e[2];  rhv.houseNumber = e[3];  rhv.apartmentNumber = e[4];  }   return lhv; }  ostream& operator<<(ostream& lhv, Consumer& rhv) {  lhv << rhv.lastname << "," << rhv.city << "," << rhv.street << "," << rhv.houseNumber << "," << rhv.apartmentNumber;  return lhv; }  int main(int argc, char\*\* argv) {  //выбирает категорию классификации символов языка С  setlocale(LC\_CTYPE, "Rus");   const auto input\_file = "A";  fstream input(input\_file, ios\_base::in | ios\_base::out);   if (!input)  {  perror("Ошибка с входным файлом");  return EXIT\_FAILURE;  }   NExternalMergeSort<Consumer, '\n'>(input\_file).sort(input, [](const Consumer& a, const Consumer& b) { return a.lastname < b.lastname; });  input.close();  system("Пауза");  //успешное выполнение программы  return EXIT\_SUCCESS; } |
| --- |

Блок кода 6 - Файл main.cpp

Протестируем адаптированный код Результаты тестирования будут продемонстрированы на рисунке 9 и 10.



Рисунок 9 - Тестирование адаптированной программы

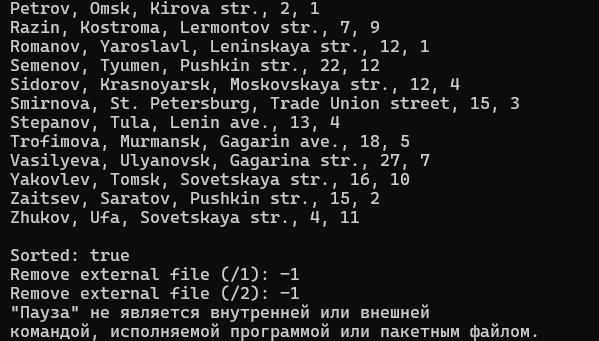


Рисунок 10 - Тестирование адаптированной программы

## **3.5 Практическая сложность алгоритма**

Практическая сложность алгоритма естественного слияния зависит от структуры данных, которые нужно отсортировать.

В худшем случае, когда все элементы в неотсортированном массиве находятся в обратном порядке, сложность алгоритма естественного слияния будет O(n2), так как придется объединять все подпоследовательности по одному элементу, пока не получим один сортированный массив.

В среднем случае, сложность алгоритма естественного слияния также оценивается как O(n2), так как склейка подпоследовательностей может потребовать большого числа операций на некоторых этапах выполнения алгоритма.

Однако, в лучшем случае, когда исходный массив уже отсортирован или содержит несколько упорядоченных подпоследовательностей, сложность алгоритма естественного слияния снижается до O(n), так как мы можем сразу объединить все упорядоченные подпоследовательности и получить сортированный массив.

Таким образом, практическая сложность алгоритма естественного слияния зависит от структуры исходных данных, и может быть как O(n) в лучшем случае, так и O(n2) в худшем и среднем случае.

Продемонстрируем подсчёт времени на разном количестве записей в файле данных в таблице 2.

Таблица 2 - Сводная таблица результатов

| **Количество записей** | **Время(мкс)** |
| --- | --- |
| 8 | 294746 |
| 16 | 1056374 |
| 32 | 3575638 |

## **3.6 Вывод по заданию №2**

Метод естественного слияния в сортировке делит список на упорядоченные подсписки и объединяет их, что обеспечивает эффективность при работе с большими списками за счет отсутствия необходимости в дополнительной памяти.

Эффективность алгоритмов прямого слияния и естественного слияния зависит от трёх основных факторов, которые мы рассмотрим ниже.

1. Сложность: Оба алгоритма имеют среднюю сложность O(n log n) при сортировке случайных данных, где n - количество элементов в массиве. В лучшем случае, алгоритм естественного слияния может иметь сложность O(n) при сортировке уже отсортированных данных, в то время как алгоритм прямого слияния сохраняет сложность O(n log n).

2. Дополнительная память: Алгоритм прямого слияния требует дополнительной памяти для временных массивов при слиянии подпоследовательностей, в то время как в алгоритме естественного слияния дополнительной памяти не требуется, так как мы используем уже отсортированные подпоследовательности для их слияния.

3. Производительность на практике: В худшем и среднем случае, алгоритм прямого слияния может оказаться менее эффективным из-за использования дополнительной памяти и наличия худшего случая сложности O(n2). Алгоритм естественного слияния может быть более эффективным в определенных случаях, таких как наличие упорядоченных подпоследовательностей в исходных данных.

Исходя из этих факторов, можно сделать вывод, что эффективность алгоритма естественного слияния может быть выше в определенных сценариях, особенно когда исходные данные уже в какой-то мере упорядочены. Однако, в общем случае алгоритм прямого слияния является более универсальным и широко применяемым благодаря своей простоте реализации и эффективности.

# **4 ВЫВОДЫ**

В процессе выполнения практической работы были достигнуты следующие цели:

* Изучены методы сортировки данных из файлов;
* Проанализированы алгоритмы прямого и естественного слияния;
* Написаны программы для реализации этих алгоритмов;
* Проведено тестирование программ;
* Программы адаптированы под индивидуальные требования;
* Проведено тестирование на различных объемах данных;
* Сделан вывод о эффективности алгоритмов.

Таким образом, основную цель работы - изучение методов сортировки данных из файлов - можно считать достигнутой.

# **5 ЛИТЕРАТУРА**

1. Бхаргава А. Грокаем алгоритмы. Иллюстрированное пособие для программистов и любопытствующих. – СПб: Питер, 2017. – 288 с.

2. Вирт Н. Алгоритмы + структуры данных = программы. – М.: Мир, 1985. – 406 с.

3. Кнут Д.Э. Искусство программирования, том 3. Сортировка и поиск, 2-е изд. – М.: ООО «И.Д. Вильямс», 2018. – 832 с.

4. Кораблин Ю.П. Структуры и алгоритмы обработки данных: учебно-методическое пособие / Ю.П. Кораблин, В.П. Сыромятников, Л.А. Скворцова. – М.: РТУ МИРЭА, 2020. — 219 с.

5. Кормен Т.Х. и др. Алгоритмы: построение и анализ, 3-е изд. – М.: ООО «И.Д.Вильямс», 2013. – 1328 с.

6. Макконнелл Дж. Основы современных алгоритмов. Активный обучающий метод. 3-е доп. изд., - М.: Техносфера, 2018. – 416 с.

7. Седжвик Р. Фундаментальные алгоритмы на C++. Анализ/Структуры данных/Сортировка/Поиск. – К.: Издательство «Диасофт», 2001. – 688 с.

8. Скиена С. Алгоритмы. Руководство по разработке, - 2-е изд. – СПб: БХВ-Петербург, 2011. – 720 с.

9. Хайнеман Д. и др. Алгоритмы. Справочник с примерами на C, C++, Java и Python, 2-е изд. – СПб: ООО «Альфа-книга», 2017. – 432 с.

10. AlgoList – алгоритмы, методы, исходники [Электронный ресурс]. URL: http://algolist.manual.ru/ (дата обращения 15.03.2022).

11. Алгоритмы – всё об алгоритмах / Хабр [Электронный ресурс]. URL: https://habr.com/ru/hub/algorithms/ (дата обращения 15.03.2022).