**Министерство науки и высшего образования Российской Федерации**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ  
**НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО**

**Факультет безопасности информационных технологий**

**Дисциплина:**

«**Алгоритмы и структуры данных»**

**ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ № 4**

«Реализация алгоритма сортировки Radix на основе кольцевой очереди»

**Выполнил:**

Арендаренко М.М, студент группы N3247

*\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

(подпись)

**Проверил:**

Ерофеев С. А.

*\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

(подпись)

Санкт-Петербург

2024 г.

**СОДЕРЖАНИЕ**

[Введение 3](#_Toc169191557)

[1 Описание функционала программы 5](#_Toc169191558)

[2 Блок-схемы 8](#_Toc169191559)

[3 Описание алгоритма сортировки на примере 13](#_Toc169191560)

[4 Код программы 14](#_Toc169191561)

[5 Результаты тестирования 20](#_Toc169191562)

[Заключение 23](#_Toc169191563)

Введение

Задача работы – разработать программу сортировки Radix, используя кольцевую очередь на базе массива. Результат сортировки нужно сохранить в красно-чёрное дерево. Для реализации поставленной цели мне необходимо решить следующие

задачи:

• Реализовать структуру кольцевой очереди на базе массива;

• Адаптировать алгоритм сортировки Radix под написанную

структуру;

• Реализовать функцию вставки элемента в красно-чёрное дерево;

• Реализовать функцию вывода элементов красно-чёрного дерево;

В данной работе реализация кольцевая очередь необходима для временного хранения элементов в каждом разряде числа. Алгоритм сортировки Radix работает путем сортировки чисел по разрядам, начиная с младшего разряда и двигаясь к старшим.

В процессе сортировки Radix каждое число разбивается на отдельные разряды (обычно в десятичной системе это цифры). Затем числа распределяются в соответствии с текущим разрядом. Коль используется для временного хранения чисел в каждой корзине. При этом числа добавляются в очередь с одной стороны и извлекаются из него с другой стороны, чтобы сохранить правильный порядок.

После того, как все числа были размещены в соответствующих корзинах, они извлекаются из очереди и объединяются в правильном порядке. Затем процесс повторяется для следующего разряда до тех пор, пока все разряды не будут обработаны.

Использование кольцевой очереди в алгоритме сортировки Radix позволяет эффективно управлять порядком обработки чисел и сохранять правильный порядок при объединении отсортированных разрядов.

Для реализации алгоритма был выбран язык программирования C++ стандарта C++17, для написания и отладки кода использовалась среда разработки Replit версии 2.66.9.

# Описание функционала программы

Кольцевая очередь в radix-сортировке используется для временного хранения элементов, которые ожидают сортировки. Когда происходит проход по данным для сортировки, элементы размещаются в зависимости от значащих разрядов.

Кольцевая очередь в radix-сортировке позволяет эффективно организовать этот процесс, так как она обеспечивает быстрый доступ к элементам, а также позволяет эффективно управлять переполнением корзин. После того как все элементы размещены, они извлекаются в порядке их размещения, чтобы завершить сортировку.

Использование кольцевой очереди в radix-сортировке помогает упростить и оптимизировать процесс сортировки элементов по разрядам, делая его более эффективным и масштабируемым для больших объемов данных

Красно-черное дерево может используется в сортировке Radix для упорядочивания элементов по их разрядам. Красно-черное дерево обеспечивает эффективную вставку, удаление и поиск элементов, что может быть полезно при выполнении операций, связанных со сравнением и упорядочиванием элементов по разрядам в процессе сортировки Radix..

Для реализации алгоритма потребовалось реализовать следующие функции:  
1. Класс Queue:

- Реализует структуру данных "очередь" с использованием динамического массива.

- Поддерживает операции добавления элемента в конец очереди, извлечения элемента из начала очереди, получения значения первого и последнего элементов, а также проверку на пустоту.

- Внутренние операции класса выполняются с временной сложностью O(1) или O(n), где n - размер очереди.

2. RBTree: Это класс, представляющий само красно-черное дерево. Он содержит приватные члены данных и методы для работы с деревом.

3. Insert: Этот метод позволяет вставить новый узел в красно-черное дерево. При вставке узла дерево поддерживает свойства красно-черного дерева.

4. LeftTurn и RightTurn: Эти методы выполняют левый и правый повороты соответственно. Они используются при балансировке дерева после вставки нового узла.

5. Print: Этот метод выводит содержимое дерева в консоль. Каждый узел выводится с указанием его значения и цвета (красный или черный).

6. Clear: Этот метод рекурсивно очищает память, выделенную для всех узлов дерева.

7.Функция counting\_sort:

- Реализует алгоритм цифровой сортировки для заданного разряда чисел.

- Использует вектор для хранения отсортированных значений и массив очередей для подсчета количества цифр в каждом разряде.

3. Функция radix\_sort:

- Определяет максимальное и минимальное значение во входном массиве, а также максимальное количество разрядов у чисел.

- Вызывает функцию counting\_sort для каждого разряда, начиная с младшего и заканчивая старшим.

4. Функция main:

- Читает входные данные из файла "input.txt" и загружает их в вектор arr.

- Вызывает функцию radix\_sort для сортировки вектора arr.

- Записывает отсортированные данные в файл "output.txt".

# Блок-схемы

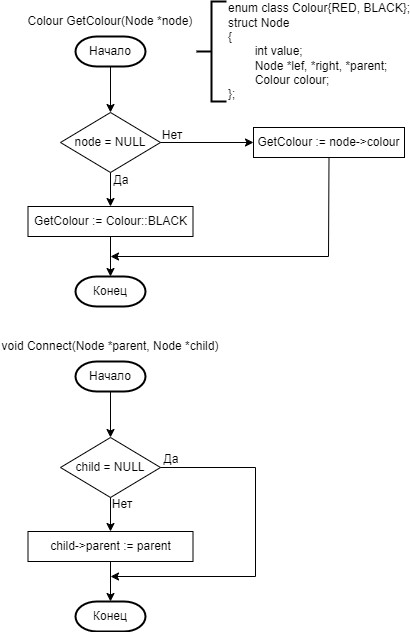


Рис. 1 Блок-схемы GetColour, Connect

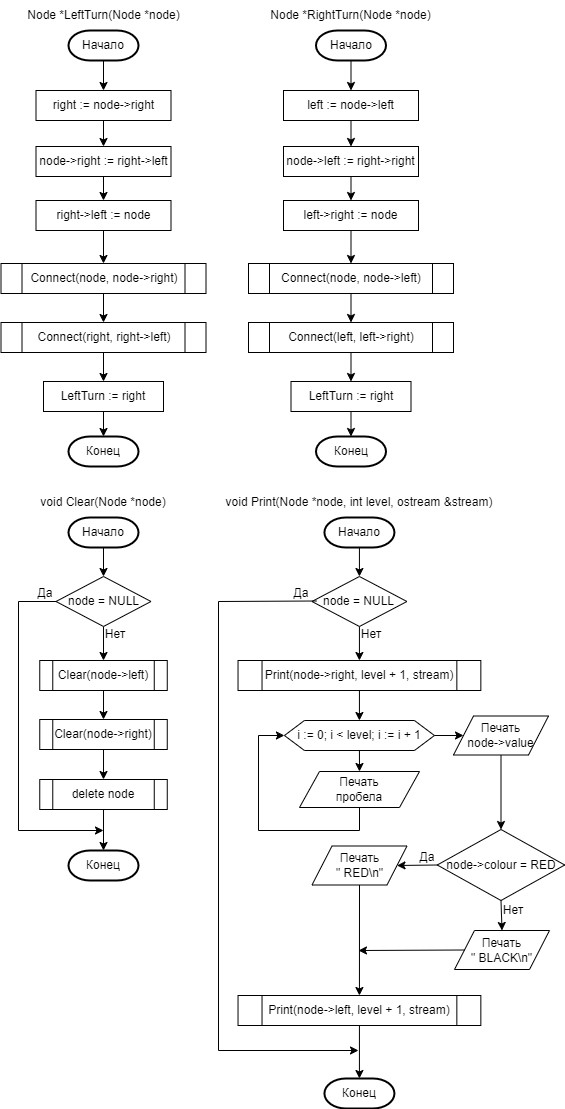


Рис. 2 Блок-схемы LeftTurn, RightTurn, Clear, Print

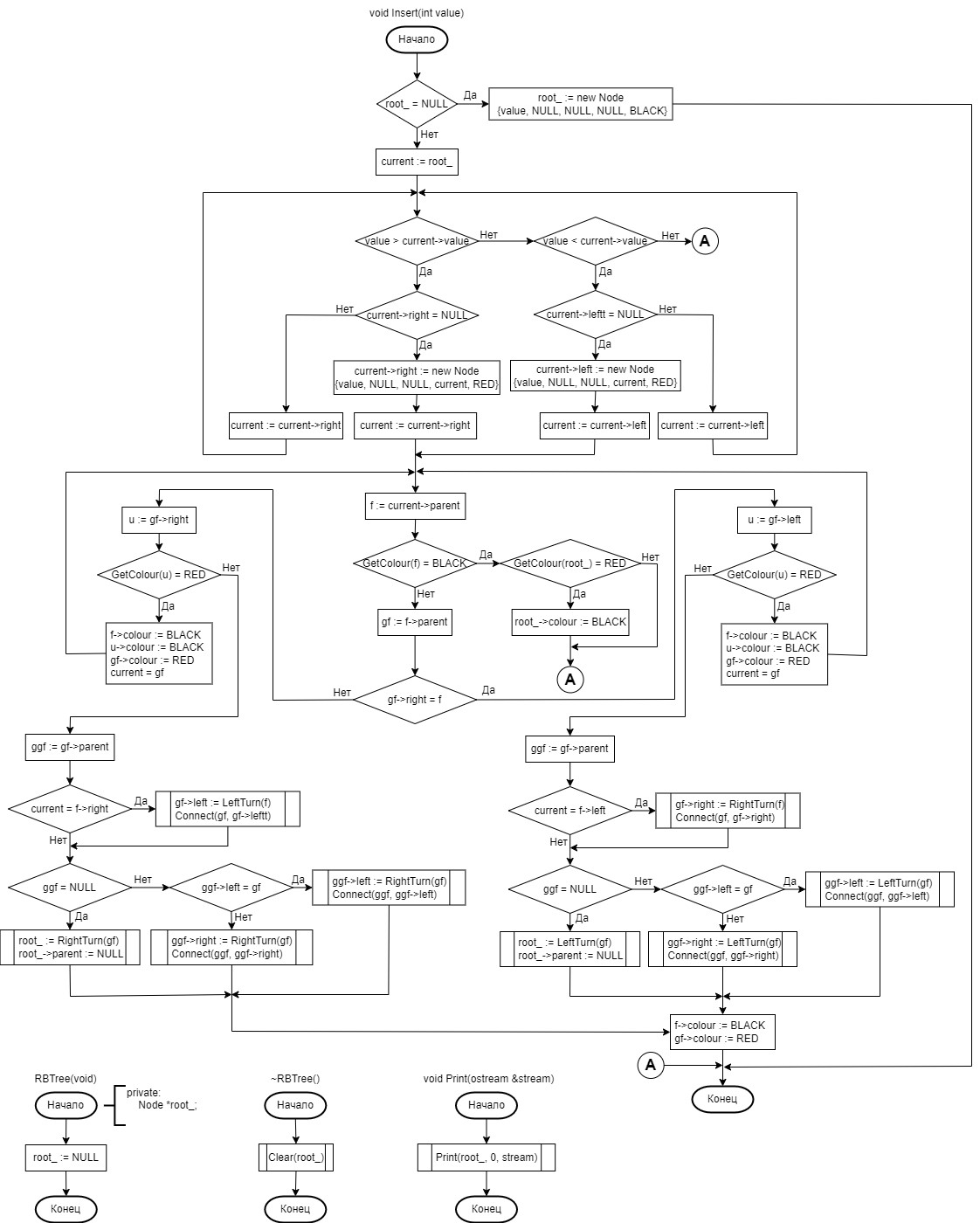


Рис. 3 Блок-схемы Insert, RBTree, ~RBTree, Print

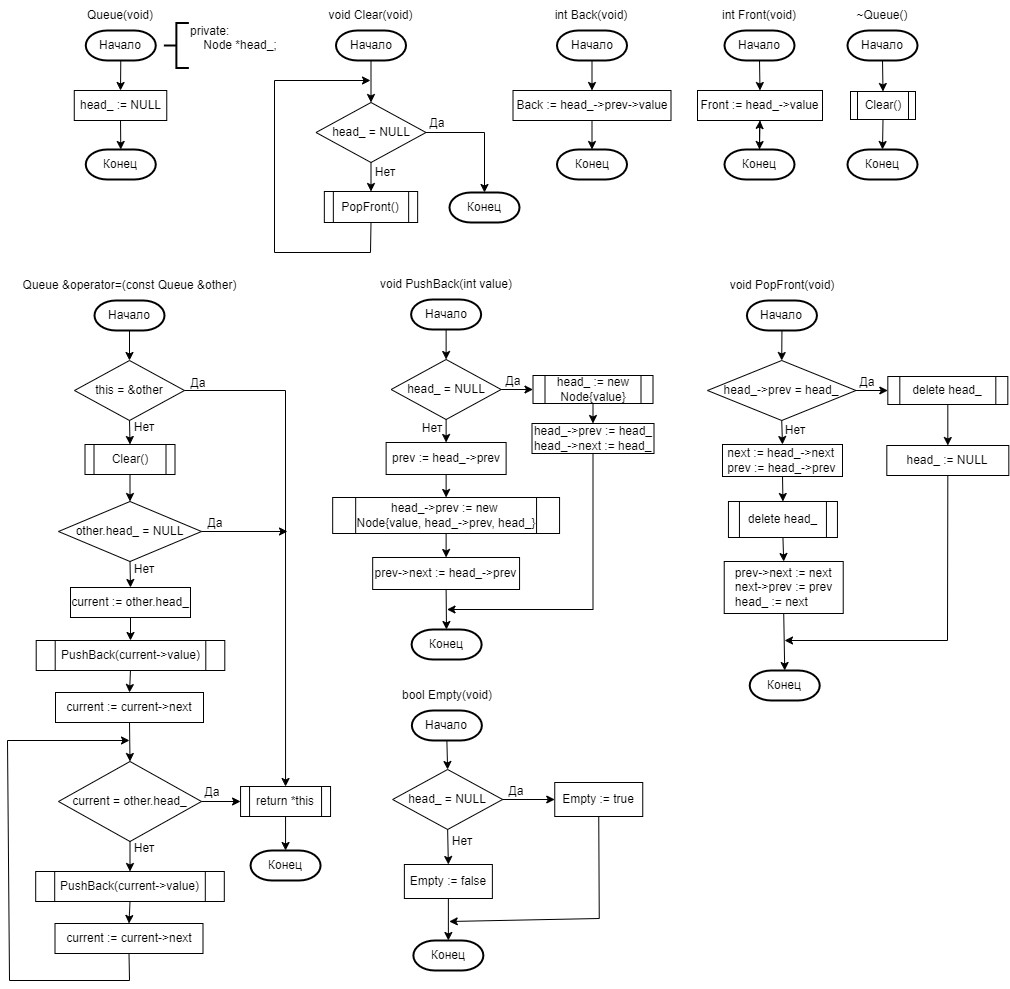


Рис. 4 Блок-схемы класса очереди

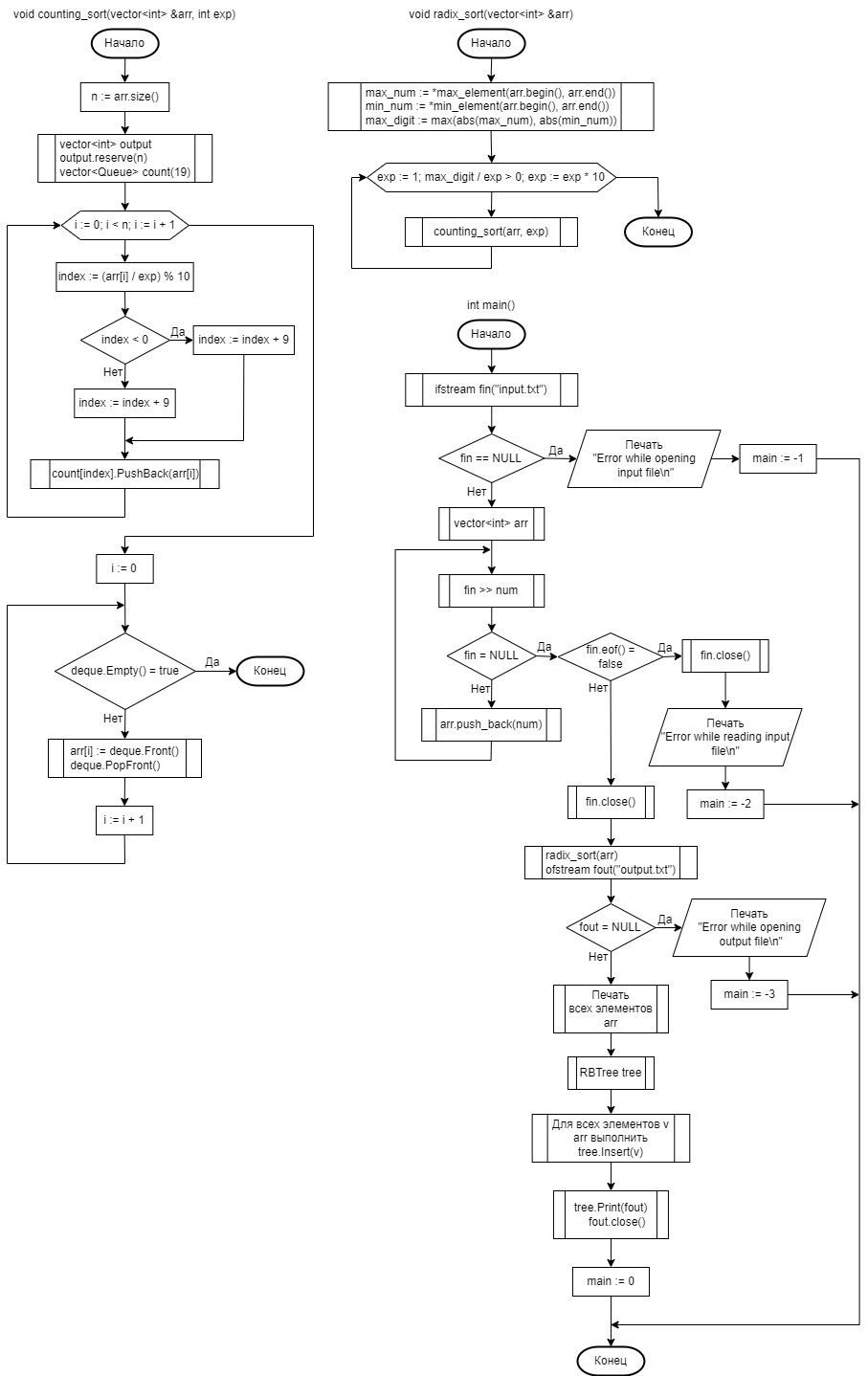


Рис. 5 Блок-схемы counting\_sort, radic\_sort, main

# Описание алгоритма сортировки на примере

Предположим, у нас есть массив чисел [170, 45, 75, 90, 802, 24, 2, 66]. Мы хотим отсортировать их при помощи поразрядной сортировки.

1. Поиск максимального числа в массиве и определение количества разрядов. В данном примере максимальное число 802, которое имеет три разряда, поэтому нам понадобятся три прохода для сортировки.

2. Начинаем с разряда единиц: сортируем числа по этому разряду и помещаем их в соответствующие корзины (0-9).

3. После этого собираем числа обратно в массив, сохраняя порядок в корзинах.

4. Повторяем процесс для разряда десятков, а затем для разряда сотен (если есть).

5. После третьего прохода весь массив будет отсортирован по всем разрядам.

В итоге, собрав все числа вместе после трех проходов, мы получим отсортированный массив: [2, 24, 45, 66, 75, 90, 170, 802].

# Код программы

|  |
| --- |
| #include<iostream>  #include<vector>  #include<fstream>  #include <algorithm>  class RBTree {  private:  enum class Colour{  RED,  BLACK,  };  struct Node {  int value;  Node\* left;  Node\* right;  Node\* parent;  Colour colour;  };  private:  static Colour GetColour(Node\* node) {  if(node == nullptr) {  return Colour::BLACK;  }  return node->colour;  }  static void Connect(Node\* parent, Node\* child) {  if(child != nullptr) {  child->parent = parent;  }  }  static Node\* LeftTurn(Node\* node) {  Node\* right = node->right;  node->right = right->left;  right->left = node;  Connect(node, node->right);  Connect(right, right->left);  return right;  }  static Node\* RightTurn(Node\* node) {  Node\* left = node->left;  node->left = left->right;  left->right = node;  Connect(node, node->left);  Connect(left, left->right);  return left;  }  static void Clear(Node\* node) {  if(node == nullptr) {  return;  }  Clear(node->left);  Clear(node->right);  delete node;  }  static void Print(Node\* node, int level, std::ostream& stream) {  if(node == NULL) {  return;  }  Print(node->right, level + 1, stream);  for(int i = 0; i < level; ++i) {  stream << " ";  }  stream << node->value;  if(node->colour == Colour::RED) {  stream << " RED\n";  } else {  stream << " BLACK\n";  }  Print(node->left, level + 1, stream);  }  private:  Node\* root\_;  public:  RBTree() : root\_(nullptr) {}  void Insert(int value) {  if(root\_ == nullptr) {  root\_ = new Node {value, nullptr, nullptr, nullptr, Colour::BLACK};  return;  }  Node\* current = root\_;  while(true) {  if(value > current->value) {  if(current->right == nullptr) {  current->right = new Node{value, nullptr, nullptr, current, Colour::RED};  current = current->right;  break;  }  current = current->right;  } else if (value < current->value) {  if(current->left == nullptr) {  current->left = new Node{value, nullptr, nullptr, current, Colour::RED};  current = current->left;  break;  }  current = current->left;  } else {  return;  }  }  while(true) {  Node\* f = current->parent;  if(GetColour(f) == Colour::BLACK) {  break;  }  Node\* gf = f->parent;  Node\* u;  if(gf->right == f) {  u = gf->left;  if(GetColour(u) == Colour::RED) {  f->colour = Colour::BLACK;  u->colour = Colour::BLACK;  gf->colour = Colour::RED;  current = gf;  continue;  }  Node\* ggf = gf->parent;  if(current == f->left) {  gf->right = RightTurn(f);  Connect(gf, gf->right);  }  if(ggf != nullptr) {  if(ggf->left == gf) {  ggf->left = LeftTurn(gf);  Connect(ggf, ggf->left);  } else {  ggf->right = LeftTurn(gf);  Connect(ggf, ggf->right);  }  } else {  root\_ = LeftTurn(gf);  root\_->parent = nullptr;  }  f->colour = Colour::BLACK;  gf->colour = Colour::RED;  return;  } else {  u = gf->right;  if(GetColour(u) == Colour::RED) {  f->colour = Colour::BLACK;  u->colour = Colour::BLACK;  gf->colour = Colour::RED;  current = gf;  continue;  }  Node\* ggf = gf->parent;  if(current == f->right) {  gf->left = LeftTurn(f);  Connect(gf, gf->left);  }  if(ggf != nullptr) {  if(ggf->left == gf) {  ggf->left = RightTurn(gf);  Connect(ggf, ggf->left);  } else {  ggf->right = RightTurn(gf);  Connect(ggf, ggf->right);  }  } else {  root\_ = RightTurn(gf);  root\_->parent = nullptr;  }  gf->colour = Colour::RED;  f->colour = Colour::BLACK;  return;  }  }  if(GetColour(root\_) == Colour::RED) {  root\_->colour = Colour::BLACK;  }  }  void Print(std::ostream& stream) const{  Print(root\_, 0, stream);  }  ~RBTree() {  Clear(root\_);  }  };  class Queue {  private:  struct Node {  int value;  Node\* prev;  Node\* next;  };  private:  Node\* head\_;  public:  Queue() : head\_(nullptr) {}  Queue(const Queue& other) : Queue() {  if(other.head\_ == nullptr) {  return;  }  Node\* current = other.head\_;  PushBack(current->value);  current = current->next;  while(current != other.head\_) {  PushBack(current->value);  current = current->next;  }  }  void Clear() {  while(head\_ != nullptr) {  PopFront();  }  }  Queue& operator=(const Queue& other) {  if(this == &other) {  return \*this;  }  Clear();  if(other.head\_ == nullptr) {  return \*this;  }  Node\* current = other.head\_;  PushBack(current->value);  current = current->next;  while(current != other.head\_) {  PushBack(current->value);  current = current->next;  }  return \*this;  }  ~Queue() {  Clear();  }  int Back() const {  return head\_->prev->value;  }  int Front() const {  return head\_->value;  }  void PushBack(int value) {  if(head\_ == nullptr) {  head\_ = new Node{value};  head\_->prev = head\_;  head\_->next = head\_;  return;  }  Node\* prev = head\_->prev;  head\_->prev = new Node{value, head\_->prev, head\_};  prev->next = head\_->prev;  }  void PopFront() {  if(head\_->prev == head\_) {  delete head\_;  head\_ = nullptr;  return;  }  Node\* next = head\_->next;  Node\* prev = head\_->prev;  delete head\_;  prev->next = next;  next->prev = prev;  head\_ = next;  }  bool Empty() const {  return head\_ == nullptr;  }  };  void counting\_sort(std::vector<int>& arr, int exp) {  int n = arr.size();  std::vector<int> output;  output.reserve(n);  std::vector<Queue> count(19);  for (int i = 0; i < n; i++) {  int index = (arr[i] / exp) % 10;  index = index < 0 ? index + 9 : index + 9;  count[index].PushBack(arr[i]);  }  size\_t i = 0;  for (auto& deque : count) {  while (!deque.Empty()) {  arr[i++] = deque.Front();  deque.PopFront();  }  }  }  void radix\_sort(std::vector<int>& arr) {  int max\_num = \*std::max\_element(arr.begin(), arr.end());  int min\_num = \*std::min\_element(arr.begin(), arr.end());  int max\_digit = std::max(std::abs(max\_num), std::abs(min\_num));  for (int exp = 1; max\_digit / exp > 0; exp \*= 10) {  counting\_sort(arr, exp);  }  }  int main()  {  std::ifstream fin("input.txt");  if (!fin) {  std::cout << "Error while opening input file\n";  return -1;  }  std::vector<int> arr;  int num;  while (fin >> num) {  arr.push\_back(num);  }  if(!fin.eof()) {  fin.close();  std::cout << "Error while reading input file\n";  return -2;  }  fin.close();  radix\_sort(arr);  std::ofstream fout("output.txt");  if (!fout) {  std::cout << "Error while opening output file\n";  return -3;  }  for (const auto& number : arr) {  fout << number << " ";  }  fout << std::endl;  RBTree tree;  for(const auto& number : arr) {  tree.Insert(number);  }  tree.Print(fout);  fout.close();  } |

# Результаты тестирования

Было проведено тестирование программы с различными входными данными. Сортировка входных данных производится успешно, в случае некорректных данных программа выводит соответствующие сообщения на экран. Тестирование производилось с помощью среды разработки Replit версии 2.66.9. Ниже представлены скриншоты входного и выходного файла для кольцевой очереди, состоящей из 100 элементов, а также случай некорректных входных данных.

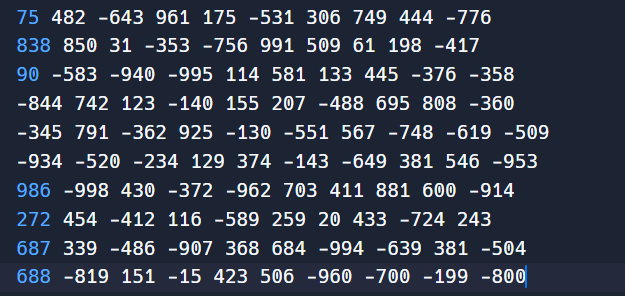


Рис. 6 Файл "input.txt"

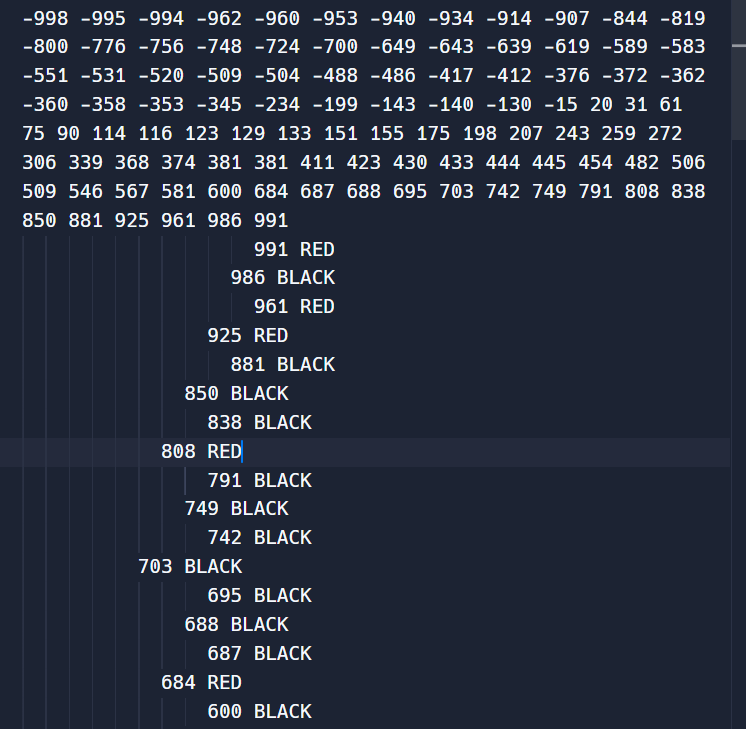
**

Рис. 7 Файл "output.txt"

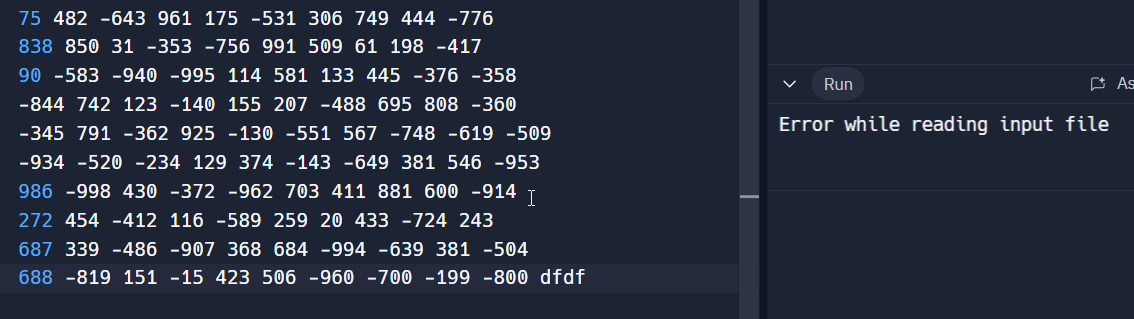


Рис. 8 Некорректные для сортировки данные

# Заключение

В ходе выполнения лабораторной работы были достигнуты все поставленные задачи. В рамках работы была разработана программа сортировки Radix, использующая кольцевую очередь на основе массива. Результат записывается в красно-черное дерево.

Тестирование программы было проведено с использованием среды разработки Replit версии 2.66.9 и языка программирования C++17. В процессе тестирования были использованы различные входные данные, и программа продемонстрировала корректную работу, а сортировка была выполнена верно. Были обнаружены и обработаны некорректные случаи ввода.