МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

Учреждение образования «Полоцкий государственный университет имени Евфросинии Полоцкой»

Факультет информационных технологий

Кафедра технологий программирования

**ЛАБОРАТОРНАЯ РАБОТА №9**

по дисциплине: **«Основы алгоритмизации и программирования»**

на тему: **«Сортировка»**

ВЫПОЛНИЛ студент группы 24-МС

Ломако И.Д.

ПРОВЕРИЛА Макарычева В. А.

Полоцк, 2025 г.

**Цель работы:** изучить быстрые и медленные алгоритмы сортировки.

Для лучшего понимания происходящего в листинге 1.1 будет представлена моя структура очереди Queue.

Листинг 1.1 – Функции внутри структуры Queue

Queue applicants;

Queue::Queue() {

front\_node = nullptr;

back\_node = nullptr;

node\_count = 0;

}

void Queue::Init() {

front\_node = nullptr;

back\_node = nullptr;

node\_count = 0;

}

void Queue::Display() {

Queue temp\_queue;

while (!isEmpty()) {

Applicant person = \*applicants.pop\_front();

temp\_queue.push\_back(person);

cout << person.name << " " << person.addr << " " << person.mark << " " << person.date << endl;

}

while (!temp\_queue.isEmpty()) {

applicants.push\_back(\*temp\_queue.pop\_front());

}

}

void Queue::push\_back(Applicant data) {

Node\* new\_node = (Node\*)AllocateMemory(sizeof(Node));

new\_node->data = data;

new\_node->next\_ptr = nullptr;

if (!back\_node) {

back\_node = front\_node = new\_node;

}

else {

back\_node->next\_ptr = new\_node;

back\_node = new\_node;

}

node\_count++;

}

Applicant\* Queue::pop\_front() {

if (isEmpty()) { return nullptr; }

int name\_length = strlen(front\_node->data.name) + 1;

int addr\_length = strlen(front\_node->data.addr) + 1;

int date\_length = strlen(front\_node->data.date) + 1;

Applicant\* person = (Applicant\*)AllocateMemory(sizeof(Applicant));

(\*person).name = (char\*)AllocateMemory(sizeof(char) \* name\_length);

(\*person).addr = (char\*)AllocateMemory(sizeof(char) \* addr\_length);

(\*person).date = (char\*)AllocateMemory(sizeof(char) \* date\_length);

strcpy\_s((\*person).name, name\_length, front\_node->data.name);

strcpy\_s((\*person).addr, addr\_length, front\_node->data.addr);

strcpy\_s((\*person).date, date\_length, front\_node->data.date);

(\*person).mark = front\_node->data.mark;

Node\* temp = front\_node->next\_ptr;

front\_node->next\_ptr = nullptr;

free(front\_node);

front\_node = temp;

if (!temp) { back\_node = temp; }

node\_count--;

return person;

}

int Queue::GetNodeCount() {

return node\_count;

}

bool Queue::isEmpty() {

return node\_count == 0;

}

void Queue::ClearDataInApplicant(Applicant& applicant) {

if (applicant.name) {

free(applicant.name);

applicant.name = nullptr;

}

if (applicant.addr) {

free(applicant.addr);

applicant.addr = nullptr;

}

if (applicant.date) {

free(applicant.date);

applicant.date = nullptr;

}

applicant.mark = 0;

}

void Queue::Clear() {

while (front\_node) {

ClearDataInApplicant(front\_node->data);

Node\* temp = front\_node->next\_ptr;

front\_node->next\_ptr = nullptr;

free(front\_node);

front\_node = temp;

}

back\_node = nullptr;

node\_count = 0;

}

В листинге была реализована структура данных "очередь" (Queue) для хранения объектов типа Applicant, содержащих информацию о абитуриентах (имя, адрес, оценка и дата). Структура Queue включает базовые методы управления: конструктор, инициализацию, добавление элементов в конец (push\_back), удаление из начала (pop\_front), отображение содержимого очереди (Display), очистку данных (Clear и ClearDataInApplicant) и проверку на пустоту (isEmpty). Для работы с памятью используются функции AllocateMemory и free, а метод Display временно извлекает элементы из очереди, выводит их и возвращает обратно, не нарушая структуру. Данная реализация обеспечивает базовую функциональность очереди с учетом хранения и обработки данных абитуриентов. Приставка перед названием функции говорит о том, что это функция принадлежит структуре Queue, это необходимо так как я также создал заголовочный файл .h, где и была расписана сама структура Queue.

**Задание 1.** Реализуйте сортировку вставками по двум полям динамической структуры курсовой работы (Лабораторные работы 5-6). При реализации алгоритма сортировки НЕ преобразовывать динамическую структуру в массив.

Реализация функций сортировок вставками для полей mark и name, а также функция для определения какая из двух строк ближе к началу алфавита представлены в листинге 1.2. Результат работы программы показан на рисунках 1.1, 1.2, 1.3, 1.4 и 1.5.

Листинг 1.2 – Функции InsertSortMark, InsertSortName и isAlphabetically

bool isAlphabetically(char\* first\_str, char\* second\_str) {

int min\_length = 0;

if (strlen(first\_str) > strlen(second\_str)) { min\_length = strlen(second\_str); }

else { min\_length = strlen(first\_str); }

bool str\_swap = false;

int i = 0;

for (; i < min\_length && (int)first\_str[i] == (int)second\_str[i]; i++);

if ((int)first\_str[i] > (int)second\_str[i]) { str\_swap = true; }

return str\_swap;

}

void InsertSortMark(Queue& queue, bool reverse) {

unsigned int queue\_length = queue.GetNodeCount();

if (queue\_length < 2) { return; }

Queue sortedQueue;

sortedQueue.push\_back(\*queue.pop\_front());

while (!queue.isEmpty()) {

Applicant extracted = \*queue.pop\_front();

Queue tempQueue;

bool inserted = false;

while (!sortedQueue.isEmpty()) {

Applicant current = \*sortedQueue.pop\_front();

if (!inserted && current.mark > extracted.mark == !reverse) {

tempQueue.push\_back(extracted);

inserted = true;

}

tempQueue.push\_back(current);

}

if (!inserted) {

tempQueue.push\_back(extracted);

}

while (tempQueue.GetNodeCount() > 0) {

sortedQueue.push\_back(\*tempQueue.pop\_front());

}

}

while (sortedQueue.GetNodeCount() > 0) {

queue.push\_back(\*sortedQueue.pop\_front());

}

}

void InsertSortName(Queue& queue, bool reverse) {

unsigned int queue\_length = queue.GetNodeCount();

if (queue\_length < 2) { return; }

Queue sortedQueue;

sortedQueue.push\_back(\*queue.pop\_front());

while (!queue.isEmpty()) {

Applicant extracted = \*queue.pop\_front();

Queue tempQueue;

bool inserted = false;

while (!sortedQueue.isEmpty()) {

Applicant current = \*sortedQueue.pop\_front();

if (!inserted && isAlphabetically(current.name, extracted.name) == !reverse) {

tempQueue.push\_back(extracted);

inserted = true;

}

tempQueue.push\_back(current);

}

if (!inserted) {

tempQueue.push\_back(extracted);

}

while (tempQueue.GetNodeCount() > 0) {

sortedQueue.push\_back(\*tempQueue.pop\_front());

}

}

while (sortedQueue.GetNodeCount() > 0) {

queue.push\_back(\*sortedQueue.pop\_front());

}

}

В листинге 1.2 реализованы два метода сортировки очереди структур Applicants по оценке и по имени с использованием алгоритма сортировки вставками, а также вспомогательная функция для лексикографического сравнения строк. Функция isAlphabetically(char\* first\_str, char\* second\_str) сравнивает две строки посимвольно до минимальной длины, возвращая значение, указывающее на необходимость обмена, если обнаружено, что символ из первой строки превышает соответствующий символ второй строки. Функции InsertSortMark(Queue& queue, bool reverse) и InsertSortName(Queue& queue, bool reverse) работают схожим образом: начально извлекается первый элемент исходной очереди, который помещается в новую очередь, затем последовательно извлекаются оставшиеся элементы и вставляются в отсортированное место промежуточной очереди с помощью сравнения либо по полю mark, либо путем вызова isAlphabetically(char\* first\_str, char\* second\_str) для поля name; параметр reverse позволяет инвертировать порядок сортировки. После завершения процесса сортировки отсортированные элементы возвращаются в исходную очередь.

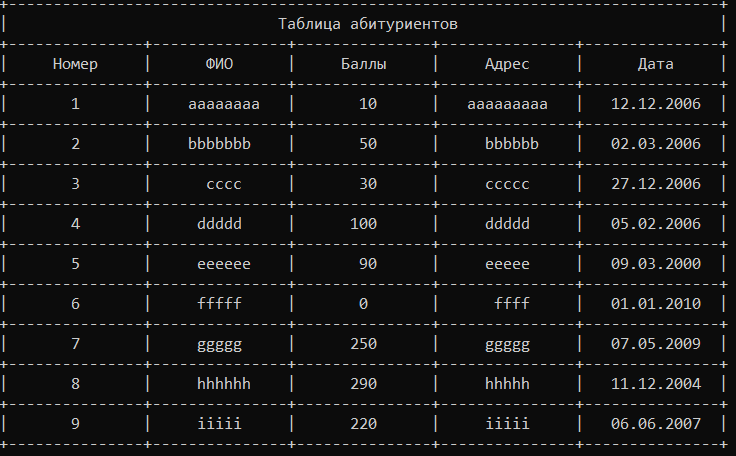


Рисунок 1.1 – База данных до сортировки

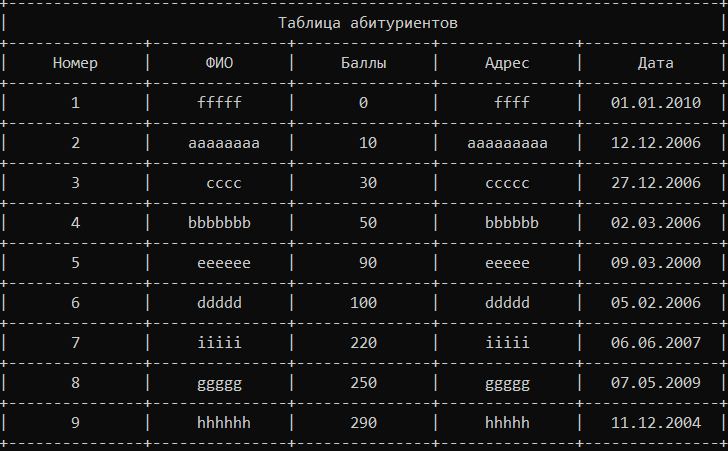


Рисунок 1.2 – База данных после сортировки по полю “Баллы” (от меньшего к большему)

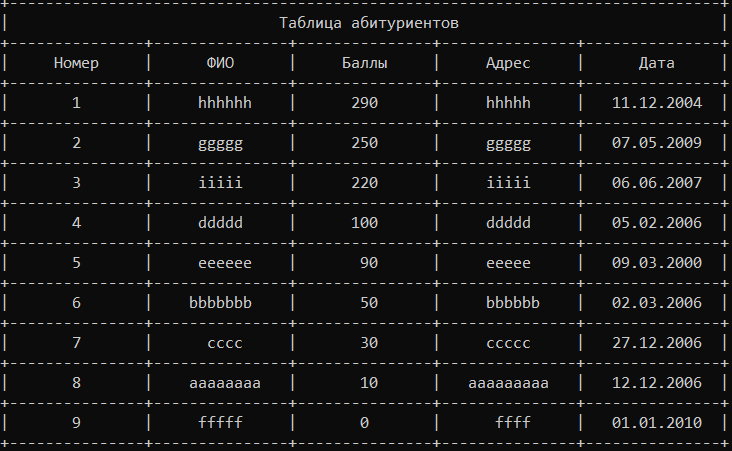


Рисунок 1.3 – База данных после сортировки по полю “Баллы” (от большего к меньшему)

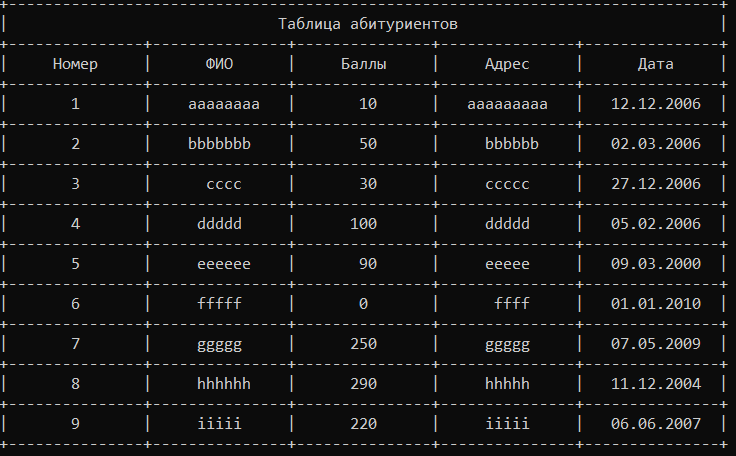


Рисунок 1.4 – База данных после сортировки вставками по полю “ФИО” (в алфавитном порядке)

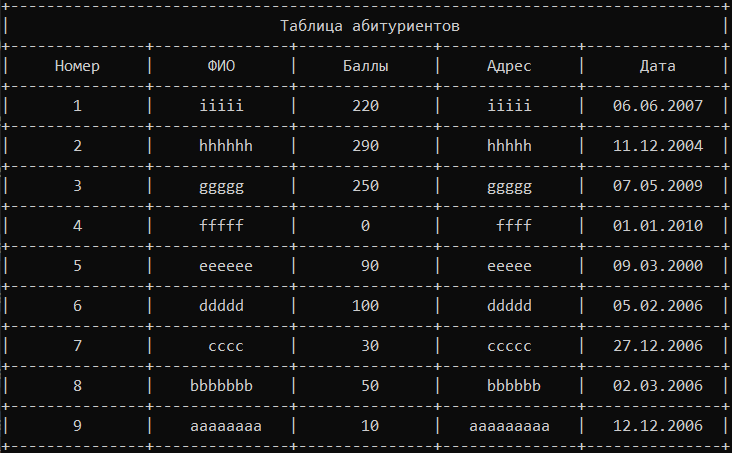


Рисунок 1.5 – База данных после сортировки вставками по полю “ФИО” (в обратном алфавитном порядке)

**Задание 2.** Реализуйте пирамидальную сортировку Флойда (Heap Sort) по двум полям динамической структуры курсовой работы (Лабораторные работы 5-6). При реализации алгоритма сортировки НЕ преобразовывать динамическую структуру в массив.

Реализация функций сортировок Флойда (Heap Sort) для полей mark и name, а также функция для определения какая из двух строк ближе к началу алфавита представлены в листинге 2.1. Результат работы программы показан на рисунках 2.1, 2.2, 2.3, 2,4 и 2.5.

Листинг 2.1 – Функции isAlphabetically, GetApplicantInPosition и SwapApplicantsInPosition

Applicant\* GetApplicantInPosition(Queue& queue, int position) {

Queue temp\_queue;

Applicant\* find\_person = nullptr;

if (position >= queue.GetNodeCount() || position < 0) { return nullptr; }

for (int i = 0; !queue.isEmpty(); i++) {

Applicant\* person = queue.pop\_front();

if (i == position) {

find\_person = person;

}

temp\_queue.push\_back(\*person);

}

for (int i = 0; !temp\_queue.isEmpty(); i++) {

queue.push\_back(\*temp\_queue.pop\_front());

}

return find\_person;

}

void SwapApplicantsInPositions(Queue& queue, int position1, int position2) {

int size = queue.GetNodeCount();

Applicant\* person1 = nullptr;

Applicant\* person2 = nullptr;

Queue temp\_queue;

if ((position1 >= size || position1 < 0) || (position2 >= size || position2 < 0) || position1 == position2) { return; }

for (int i = 0; !queue.isEmpty(); i++) {

Applicant\* extracted = queue.pop\_front();

if (i == position1) { person1 = extracted; }

if (i == position2) { person2 = extracted; }

temp\_queue.push\_back(\*extracted);

}

if (!person1 || !person2) { position1 = position2 = -1; }

for (int i = 0; !temp\_queue.isEmpty(); i++) {

Applicant extracted = \*temp\_queue.pop\_front();

if (i == position1) { extracted = \*person2; }

if (i == position2) { extracted = \*person1; }

queue.push\_back(extracted);

}

}

bool isAlphabetically(char\* first\_str, char\* second\_str) {

int min\_length = 0;

if (strlen(first\_str) > strlen(second\_str)) { min\_length = strlen(second\_str); }

else { min\_length = strlen(first\_str); }

bool str\_swap = false;

int i = 0;

for (; i < min\_length && (int)first\_str[i] == (int)second\_str[i]; i++);

if ((int)first\_str[i] > (int)second\_str[i]) { str\_swap = true; }

return str\_swap;

}

void HeapifyMark(Queue& queue, int size, int i, bool reverse) {

int largest = i;

int left = 2 \* i + 1;

int right = 2 \* i + 2;

Applicant\* rootApplicant = GetApplicantInPosition(queue, i);

Applicant\* leftApplicant = GetApplicantInPosition(queue, left);

Applicant\* rightApplicant = GetApplicantInPosition(queue, right);

if (left < size && leftApplicant->mark > rootApplicant->mark == !reverse)

largest = left;

if (right < size && rightApplicant->mark > GetApplicantInPosition(queue, largest)->mark == !reverse)

largest = right;

if (largest != i) {

SwapApplicantsInPositions(queue, i, largest);

HeapifyMark(queue, size, largest, reverse);

}

}

void HeapSortQueueMark(Queue& queue, bool reverse) {

int n = queue.GetNodeCount();

if (n < 2) return;

for (int i = n / 2 - 1; i >= 0; i--) {

HeapifyMark(queue, n, i, reverse);

}

for (int i = n - 1; i > 0; i--) {

SwapApplicantsInPositions(queue, 0, i);

HeapifyMark(queue, i, 0, reverse);

}

}

void HeapifyName(Queue& queue, int size, int i, bool reverse) {

int largest = i;

int left = 2 \* i + 1;

int right = 2 \* i + 2;

Applicant\* rootApplicant = GetApplicantInPosition(queue, i);

Applicant\* leftApplicant = GetApplicantInPosition(queue, left);

Applicant\* rightApplicant = GetApplicantInPosition(queue, right);

if (left < size && isAlphabetically(leftApplicant->name, rootApplicant->name) == !reverse) {

largest = left;

}

if (right < size && isAlphabetically(rightApplicant->name, GetApplicantInPosition(queue, largest)->name) == !reverse) {

largest = right;

}

if (largest != i) {

SwapApplicantsInPositions(queue, i, largest);

HeapifyName(queue, size, largest, reverse);

}

}

void HeapSortQueueName(Queue& queue, bool reverse) {

int n = queue.GetNodeCount();

if (n < 2) return;

for (int i = n / 2 - 1; i >= 0; i--) {

HeapifyName(queue, n, i, reverse);

}

for (int i = n - 1; i > 0; i--) {

SwapApplicantsInPositions(queue, 0, i);

HeapifyName(queue, i, 0, reverse);

}

}

В листинге представлены функции для работы с очередями структур типа Applicant, обеспечивающие выбор элемента по индексу, обмен элементов и сортировку на основе двух критериев: оценки (mark) и имен посредством лексикографического сравнения. Функция GetApplicantInPosition(Queue& queue, int position) извлекает искомый элемент, временно перемещая все узлы очереди в дополнительную структуру для сохранения исходного порядка, а SwapApplicantsInPositions(Queue& queue, int position1, int position2) реализует обмен двумя абитуриентами через последовательное извлечение элементов, замену на заданных позициях и восстановление порядка очереди. Функция isAlphabetically(char\* first\_str, char\* second\_str) сравнивает строки по символам до минимальной длины для определения необходимости перестановки при сортировке по имени. Алгоритмы сортировки кучи реализованы в функциях HeapifyMark(Queue& queue, int size, int i, bool reverse) и HeapSortQueueMark(Queue& queue, bool reverse) для поля mark и аналогично в HeapifyName(Queue& queue, int size, int i, bool reverse) и HeapSortQueueName(Queue& queue, bool reverse) для имен, где на каждом шаге происходит корректировка структуры очереди с учетом параметра reverse, позволяющего инвертировать порядок сортировки.

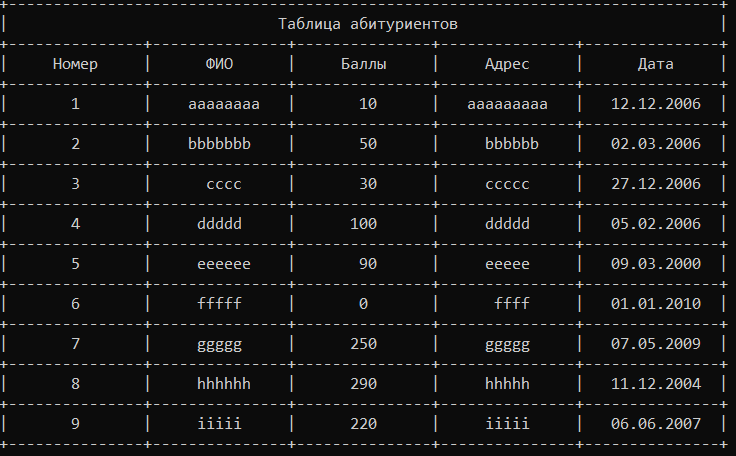


Рисунок 2.1 – База данных до сортировки

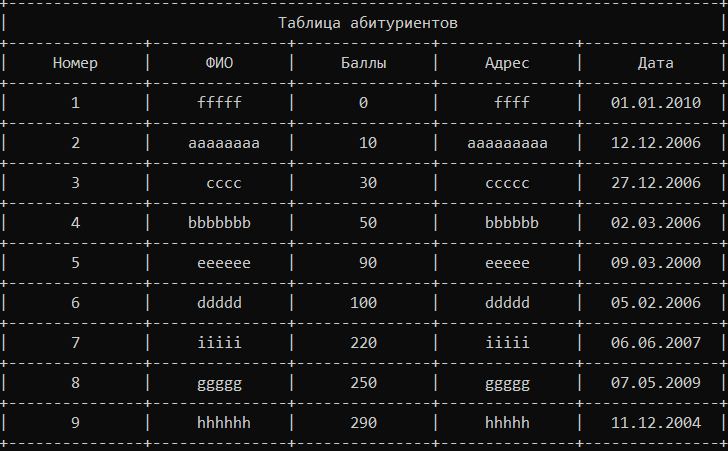


Рисунок 2.2 – База данных после сортировки Флойда по полю “Баллы” (от меньшего к большему)

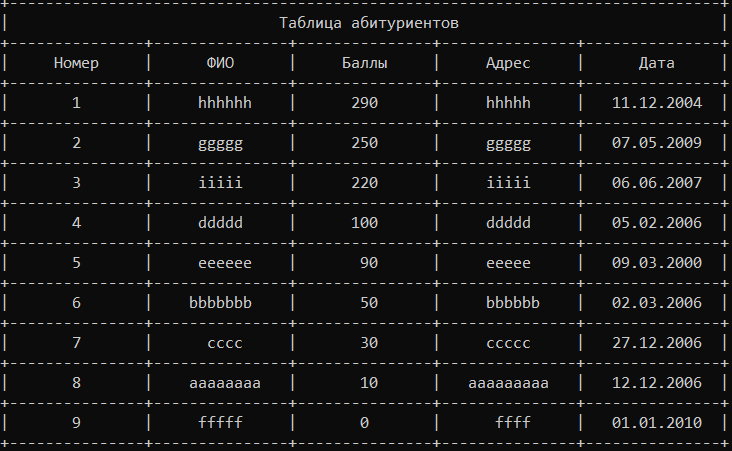


Рисунок 2.3 – База данных после сортировки Флойда по полю “Баллы” (от большего к меньшему)

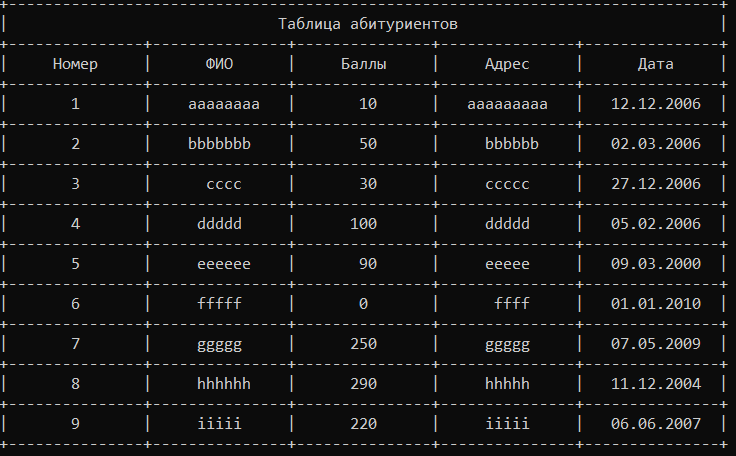


Рисунок 2.4 – База данных после сортировки Флойда по полю “ФИО” (в алфавитном порядке)

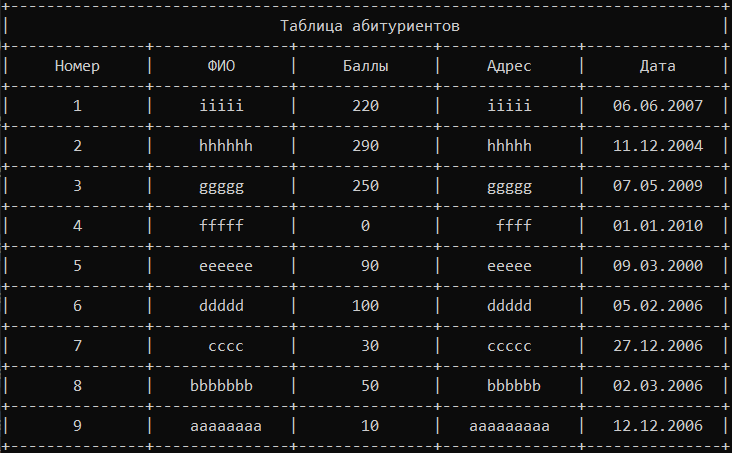


Рисунок 2.5 – База данных после сортировки Флойда по полю “ФИО” (в обратном алфавитном порядке)

**ВЫВОД**

Во время выполнения лабораторной работы были реализованы и сравнены различные алгоритмы сортировки, применённые к структуре данных "очередь", содержащей элементы типа Applicant. Были рассмотрены как медленные (вставками) методы сортировки – InsertSortMark и InsertSortName, так и более эффективные алгоритмы – пирамидальная сортировка (heapSortQueueMark и heapSortQueueName). Также были реализованы вспомогательные функции для работы с очередью: извлечение и обмен элементов по позиции, сравнение строк по алфавиту. Медленные сортировки характеризуются простотой реализации, но низкой производительностью при большом объёме данных, в то время как быстрая пирамидальная сортировка показывает лучшую эффективность за счёт построения и упорядочивания кучи. Работа продемонстрировала принципы реализации и применения различных алгоритмов сортировки, а также особенности их интеграции в пользовательские структуры данных.