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 здесь 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 динамическиую структуру в массив.

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) {

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) {

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) {

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)) {

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());

}

}

Данный листинг содержит две функции — InsertSortMark и InsertSortName, а также функцию isAlphabetically для сравнения строк и определения их алфавитного порядка, реализующие сортировку очереди Queue, содержащей элементы типа Applicant, вставками. Функция InsertSortMark сортирует очередь по возрастанию оценки (mark), а InsertSortName — по алфавиту имени (name) с использованием вспомогательной функции isAlphabetically. Обе функции создают новую очередь (sortedQueue) и поочередно извлекают элементы из исходной очереди, вставляя их в нужное место отсортированной, сохраняя порядок. После завершения сортировки элементы возвращаются в исходную очередь. Реализация выполняет стабильную сортировку с сохранением целостности данных.

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

Листинг 2.1 – Функции isAlphabetically, GetApplicantInPosition и SwapApplicantsInPosition

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;

}

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);

}

}

Данный листинг включает три функции, предназначенные для работы с элементами очереди Queue, содержащей объекты типа Applicant. Функция isAlphabetically сравнивает две строки посимвольно и возвращает true, если первая строка должна следовать после второй по алфавиту, что используется при сортировке. Функция GetApplicantInPosition возвращает указатель на элемент очереди, находящийся на заданной позиции, не нарушая порядка очереди за счёт временной очереди temp\_queue. Функция SwapApplicantsInPositions осуществляет обмен двух элементов очереди по заданным позициям, также используя временную очередь для сохранения порядка. Все функции обеспечивают корректную обработку очереди без изменения её структуры после завершения операций.

Листинг 2.2 – Функции heapifyMark, heapifyName, heapSortQueueMark и heapSortQueueName

void heapifyMark(Queue& queue, int size, int i) {

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)

largest = left;

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

largest = right;

if (largest != i) {

SwapApplicantsInPositions(queue, i, largest);

heapifyMark(queue, size, largest);

}

}

void heapSortQueueMark(Queue& queue) {

int n = queue.GetNodeCount();

if (n < 2) return;

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

heapifyMark(queue, n, i);

}

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

SwapApplicantsInPositions(queue, 0, i);

heapifyMark(queue, i, 0);

}

}

void heapifyName(Queue& queue, int size, int i) {

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)) {

largest = left;

}

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

largest = right;

}

if (largest != i) {

SwapApplicantsInPositions(queue, i, largest);

heapifyName(queue, size, largest);

}

}

void heapSortQueueName(Queue& queue) {

int n = queue.GetNodeCount();

if (n < 2) return;

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

heapifyName(queue, n, i);

}

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

SwapApplicantsInPositions(queue, 0, i);

heapifyName(queue, i, 0);

}

}

Данный листинг реализует сортировку очереди Queue, содержащей элементы типа Applicant, с использованием пирамидальной (heap) сортировки. Функции heapSortQueueMark и heapifyMark обеспечивают сортировку по оценке (mark), а heapSortQueueName и heapifyName — по алфавитному порядку имени (name) с использованием вспомогательной функции isAlphabetically. Сначала формируется куча, начиная с середины структуры, а затем происходит поочередный обмен корня с последним элементом и восстановление свойства кучи. Доступ к элементам осуществляется через GetApplicantInPosition, а перестановка — через SwapApplicantsInPositions, что позволяет адаптировать алгоритм пирамидальной сортировки для работы с очередью, не нарушая ее целостности.

**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.