Министерство образования Республики Беларусь

Учреждение образования «Белорусский государственный университет информатики и радиоэлектроники»

Факультет компьютерных систем и сетей

Кафедра информатики

Дисциплина: Архитектура вычислительных систем

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
к курсовому проекту
на тему

«Библиотека алгоритмов во внешней памяти»

БГУИР КП 1-40 04 01 009 ПЗ

Студент гр. 853501
В.А. Шавель
Руководитель
Ст. преподаватель кафедры информатики
В. В. Шиманский

Минск 2020

Учреждение образования
«БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ»
Факультет компьютерных систем и сетей
Кафедра информатики

УТВЕРЖДАЮ
Заведующий кафедрой ИИТП
_____Волорова Н. А.
«____» _____ 2021 г.

ЗАДАНИЕ
по курсовому проекту

Группа 853501

Студенту *Шавелю Валерию Александровичу*

1.Тема проекта: Библиотека алгоритмов во внешней памяти
2.Сроки сдачи студентом законченного проекта: 21.12.2021 г.
3.Исходные данные к проекту: Для написания курсового проекта была выбрана среда разработки Microsoft Visual Studio 2017, а также язык программирования C++.
4.Содержание расчетно-пояснительной записки (перечень подлежащих разработке вопросов):
Введение
Раздел 1. Анализ предметной области
Раздел 2. Разработка программного средства
Раздел 3. Демонстрация работы
Раздел 4. Анализ производительности алгоритмов
Заключение. Список использованных источников. Приложение
5.Перечень графического материала (с указанием обязательных чертежей и графиков):
_____
6.Консультанты по проекту: Шиманский В. В.
7.Дата выдачи задания: 20.09.2020 г
8.Календарный график работы над проектом на весь период проектирования (с указанием сроков выполнения и трудоемкости отдельных этапов):

№ п/п	Наименование этапов курсового проекта	Срок выполнения этапов проекта	Примечание
1.	1-я опроцентовка (введение, раздел 1)	16.10.2020	30%
2.	2-я опроцентовка (раздел 2, раздел 3, раздел 4,)	28.10.2020	60%
3.	3-я опроцентовка (демонстрация, заключение)	12.11.2020	100%
4.	Защита курсового проекта	21.12.2020	Согласно графику

Руководитель _____ (Шиманский В. В.)
Задание принял к исполнению 15.09.2020 _____ (_____)

Содержание

ВВЕДЕНИЕ 4
1. АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ 6
1.1 Обзор основных алгоритмов внешней памяти 6
1.2 Обоснование выбора языка программирования и среды разработки 6
1.3 Постановка задачи 7
2. РАЗРАБОТКА ПРОГРАММНОГО СРЕДСТВА 8
2.1 Алгоритм внешней сортировки 8
2.2 Алгоритм Карацубы во внешней памяти 9
2.3 B-дерево во внешней памяти 10
2.3.1 Поиск 11
2.3.2 Добавление ключа 11
2.3.3 Удаление ключа 11
2.4 Разработка серверной части 12
2.5 Разработка клиентской части 12
3. ДЕМОНСТРАЦИЯ РАБОТЫ 14
4. АНАЛИЗ ПРОИЗВОДИТЕЛЬНОСТИ АЛГОРИТМОВ 19
ЗАКЛЮЧЕНИЕ 21
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 22
ПРИЛОЖЕНИЕ. Исходный код алгоритма внешней сортировки 23

ВВЕДЕНИЕ

В вычислительной технике алгоритмы внешней памяти - это алгоритмы, разработанные для обработки данных, которые слишком велики для того, чтобы одновременно помещаться в память компьютера. Такие алгоритмы должны быть оптимизированы для эффективного извлечения и доступа к данным, хранящимся в медленной объемной памяти (вспомогательной памяти), такой как жесткие диски или ленточные накопители, или когда память находится в компьютерной сети. Алгоритмы внешней памяти анализируются в модели внешней памяти.

Модель внешней памяти - это абстрактная машина, похожая на модель машины с ОЗУ, но с кешем в дополнение к основной памяти. Модель отражает тот факт, что операции чтения и записи выполняются намного быстрее в кеше, чем в основной памяти, и что чтение длинных смежных блоков происходит быстрее, чем чтение в случайном порядке с использованием головки чтения и записи на диске. Время работы алгоритма в модели внешней памяти определяется количеством операций чтения и записи в основную память. Модель была представлена Алоком Аггарвалом и Джеффри Виттером в 1988 году. Модель внешней памяти связана с cache-oblivious, но алгоритмы в модели внешней памяти могут знать как размер блока, так и размер кэша.

Модель состоит из процессора с внутренней памятью или кешем размера M, подключенного к неограниченной внешней памяти. Как внутренняя, так и внешняя память разделены на блоки размера B. Одна операция ввода / вывода или передачи памяти состоит в перемещении блока из B смежных элементов из внешней во внутреннюю память, а время работы алгоритма определяется числом эти операции ввода / вывода.

Модель внешней памяти фиксирует иерархию памяти, которая не моделируется в других общих моделях, используемых при анализе структур данных, таких как машина произвольного доступа, и полезна для доказательства нижних границ для структур данных. Модель также полезна для анализа алгоритмов, которые работают с наборами данных, слишком большими, чтобы поместиться во внутренней памяти.

Типичным примером являются географические информационные системы, особенно цифровые модели рельефа, где полный набор данных легко превышает несколько гигабайт или даже терабайт данных.

Эта методология распространяется не только на процессоры общего назначения, но также включает вычисления на GPU и классическую цифровую обработку сигналов. В вычислениях общего назначения на графических процессорах (GPGPU) мощные графические карты (GPU) с небольшим объемом памятииспользуются с относительно медленной CPU-GPU передачей памяти.

В такой модели многие классические алгоритмы видоизменяются в сторону оптимизации количества операций ввода-вывода, а некоторые из них становятся неприменимы из-за слишком низких показателей. Поэтому в

современных условиях, с постоянным ростом объемов данных, исследование алгоритмов в модели внешней памяти является важной задачей.

1. АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ

1.1 Обзор основных алгоритмов внешней памяти

Алгоритмы в модели внешней памяти используют тот факт, что при извлечении одного объекта из внешней памяти получается целый блок размером B. Это свойство иногда называют локальностью.

Поиск элемента среди N объектов возможен в модели внешней памяти с использованием B-дерева с коэффициентом ветвления B. Использование B-дерева позволяет осуществлять поиск, вставку и удаление за $O(\log_B N)$ время (в нотации большее O). Теоретически, это минимальное время выполнения, возможное для этих операций, поэтому использование B-дерева является асимптотически оптимальным.

Внешняя сортировка - это сортировка в модели внешней памяти. Внешнюю сортировку можно выполнить с помощью сортировки распределения, аналогичной быстрой сортировке, или с помощью $\frac{M}{B}$-мерной сортировки слиянием. Оба варианта достигают асимптотически оптимального времени выполнения $O(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B})$ для сортировки N объектов. Эта граница также относится к быстрому преобразованию Фурье в модели внешней памяти.

Задача перестановки состоит в том, чтобы переставить N элементов в соответствии с конкретной перестановкой. Это может быть сделано либо путем сортировки, которая требует вышеуказанного времени выполнения сортировки, либо путем вставки каждого элемента по порядку и игнорирования преимущества локальности. Таким образом, перестановку можно выполнить за $O(\min\left(N, \frac{N}{B} log_{\frac{M}{B}}\frac{N}{B}\right))$ время.

1.2 Обоснование выбора языка программирования и среды разработки

В качестве языка программирования был выбран объектно-ориентированный язык C++. Этот язык является одним из наиболее подходящих языков для реализации алгоритмов, так как обладает высокими показателями скорости выполнения. Кроме того, в этом языке быстро работают операции ввода/вывода, что критично конкретно для алгоритмов внешней памяти.

Использование именно языка C++, а не языка более низкого уровня C, обусловлено возможностью примения объектно-ориентированного подхода при реализации алгоритмов. Таким образом, B-дерево может быть реализовано в качестве класса, предоставляющего все необходимые функции пользователю и скрывающего внутреннюю реализацию модификатором private. Также, применение шаблонного класса позволяет компилятору используя один и тот же исходный код сгенерировать реализации B-дерева для всех подходящих типов данных.

В качестве среды разработки было выбрано программное средство Visual Studio 2017. Это программное средство хорошо зарекомендовало себя в том числе и при разработке крупных промышленных проектов, а также обладает большим количество функций облегчающих написание кода, таких как IntelliSense.

1.3 Постановка задачи

В задачу курсового проекта входит разработать:

1. Библиотеку алгоритмов, содержащую следующие алгоритмы:
   a) Внешняя сортировка
   b) Алгоритм Карацубы для перемножения многочленов в модели внешней памяти
   c) B-дерево в модели внешней памяти
2. Серверную часть, обеспечивающую хранение данных по принципу ключ-значение с использованием B-дерева
3. Клиентскую часть, обеспечивающую взаимодействие пользователя с серверной частью и локальное использование алгоритма Карацубы и внешней сортировки

2. РАЗРАБОТКА ПРОГРАММНОГО СРЕДСТВА

2.1 Алгоритм внешней сортировки

Внешняя сортировка - это класс алгоритмов сортировки, которые могут обрабатывать большие объемы данных. Внешняя сортировка требуется, когда сортируемые данные не помещаются в основную память вычислительного устройства (обычно ОЗУ) и вместо этого они должны находиться в более медленной внешней памяти, обычно на жестком диске. Таким образом, алгоритмы внешней сортировки являются алгоритмами внешней памяти и, следовательно, к ней применима модель внешней памяти.

Внешние алгоритмы сортировки обычно делятся на два типа: сортировка распределения, которая напоминает быструю сортировку, и внешнюю сортировку слиянием. Последняя обычно использует гибридную стратегию сортировки слиянием. На этапе сортировки фрагменты данных, достаточно маленькие для размещения в основной памяти, считываются, сортируются и записываются во временный файл. На этапе объединения отсортированные подфайлы объединяются в один больший файл.

Для реализации был выбран алгоритм внешней сортировки слиянием.

Алгоритм сначала сортирует M элементов одновременно и помещает отсортированные списки обратно во внешнюю память. Затем он рекурсивно выполняет $\frac{M}{B}$-мерное слияние этих отсортированных списков. Для этого слияния B элементов из каждого отсортированного списка загружаются во внутреннюю память, и минимум выводится повторно.

Например, для сортировки 900 мегабайт данных, используя только 100 мегабайт оперативной памяти:

1. Прочитать 100 МБ данных в основной памяти и отсортировать их обычным способом, например быстрой сортировкой.
2. Записать отсортированные данные на диск.
3. Повторять шаги 1 и 2 до тех пор, пока все данные не будут отсортированы по 100 МБ (900 МБ / 100 МБ = 9), которые теперь необходимо объединить в один выходной файл.
4. Считать первые 10 МБ (= 100 МБ / (9 блоков + 1)) каждого отсортированного фрагмента во входные буферы в основной памяти и выделить оставшиеся 10 МБ для выходного буфера. (На практике можно обеспечить лучшую производительность, увеличив размер выходного буфера и немного уменьшив входные буферы.)
5. Выполнить 9-мерное слияние и сохранить результат в выходном буфере. Когда выходной буфер заполняется, записать его в последний отсортированный файл и очистить его. Всякий раз, когда какой-либо из 9 входных буферов очищается, заполнить его следующими 10 МБ связанного с ним отсортированного фрагмента размером 100 МБ до тех пор, пока они не закончатся. Это ключевой шаг, который заставляет внешнюю сортировку слияния работать внешне - поскольку алгоритм слияния делает только один

проход последовательно через каждый из файлов, каждый файл не должен загружаться полностью; последовательные части куска могут быть загружены по мере необходимости.

Предыдущий пример - однопроходная сортировка. Сортировка заканчивается одиночным k-мерным слиянием, а не серией двумерных, как в типичной сортировке в памяти.

Ограничение однопроходного слияния состоит в том, что с увеличением количества фрагментов память будет разделяться на большее количество буферов, поэтому каждый буфер будет меньше. Это вызывает много меньших чтений. Таким образом, для сортировки, скажем, 50 ГБ в 100 МБ ОЗУ использование одного прохода слияния неэффективно: диск пытается заполнить входные буферы данными каждого 500 фрагментов по 200 КБ за раз, и это занимает большую часть времени сортировки. Использование двух проходов слияния решает проблему. Процесс сортировки может выглядить так:

1. Запустить начальный проход сортировки фрагментов, как и раньше.
2. Выполнить первый проход слияния, объединяющий 25 фрагментов за раз, в результате чего получается 20 больших отсортированных фрагментов.
3. Выполнить второй проход слияния, чтобы объединить 20 больших отсортированных фрагментов.

В итоге, асимптотика алгоритма $O(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B})$ [1].

2.2 Алгоритм Карацубы во внешней памяти

Алгоритм Карацубы умножения многочленов является одним из алгоритмов, показывающих оценку лучше чем $O(n*m)$, где n,m – длины многочленов. Хотя он и уступает алгоритму, использующему быстрое преобразование Фурье, асимтотически, для определенных длин многочленов он является оптимальным на практике.

Пусть $A(x), B(x)$ – перемножаемые многочлены, имеющие одинаковую четную длину $n, n = 2k$. В случае невыполнения этого условия можно дополнить многочлены ведущими нулями. Тогда представим $A(x) = A_1(x) + x^k A_2(x)$ и, аналогично, $B(x) = B_1(x) + x^k B_2(x)$. Распишем произведение многочленов следующим образом:

$$A(x) * B(x) = A_1(x) * B_1(x) + x^k \Big(\big(A_1(x) + A_2(x)\big) * \big(B_1(x) + B_2(x)\big) - \big(A_1(x) * B_1(x) + A_2(x) * B_2(x)\big)\Big) + x^n A_2(x) * B_2(x).$$

Таким образом, мы свели задачу о перемножении двух многочленов n к 3 различным перемножениям многочленов имеющих вдвое меньшую длину, некоторому количетсву операций сложения и умножения на степень x. На практике это дает асимптотику $O\big(n^{\log_2 3}\big) \approx O(n^{1,5849})$.

Рассмотрим особенности реализации этого алгоритма во внешней памяти. Будем рекурсивно разбивать пару файлов на 3 меньшие пары файлов по принципу алгоритма Карацубы. Затем, когда в определенный момент

многочлены в паре файлов начинают помещаться в ОЗУ, можно запустить классический алгоритм Карацубы для этой пары. Таким образом, маленькие пары буду решены быстро, и для них не будет необходимости создавать новые файлы. Так же можно применить еще одну оптимизацию, и когда размер многочленов становится совсем маленьким применять квадратичный алгоритм их перемножения.

Отдельно рассмотрим операцию разбиения одной пары файлов на три других пары. Ее реализация для одного файла из пары проиллюстрирована рисунком 2.1.

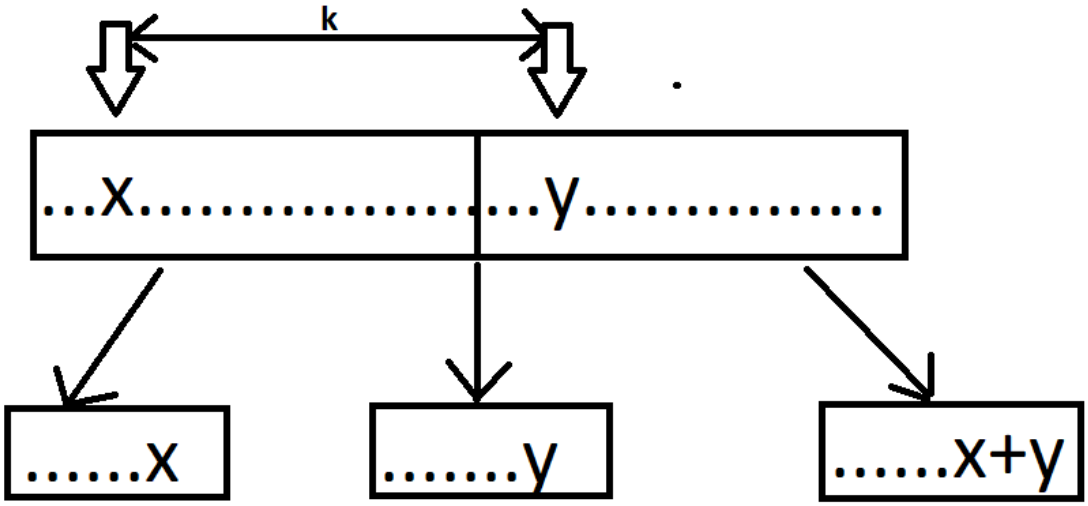

Рисунок 2.1 Схема разбиения одного файла на три меньших

2.3 B-дерево во внешней памяти

B-деревом называется дерево, удовлетворяющее следующим свойствам:

1. Ключи в каждом узле обычно упорядочены для быстрого доступа к ним. Корень содержит от 1 до $2t - 1$ ключей. Любой другой узел содержит от $t - 1$ до $2t - 1$ ключей. Листья не являются исключением из этого правила. Здесь t — параметр дерева, не меньший 2 (и обычно принимающий значения от 50 до 2000) [2].
2. У листьев потомков нет. Любой другой узел, содержащий $K_1, K_2, \dots, K_n$, ключи содержит $n + 1$ потомков. При этом
   a) Первый потомок и все его потомки содержат ключи из интервала $(-\infty, K_1)$
   b) Для $2 \le i \le n$, i-й потомок и все его потомки содержат ключи из интервала (K_{i-1}, K_i)
   c) $(n + 1)$-й потомок и все его потомки содержат ключи из интервала (K_n, ∞)
3. Глубина всех листьев одинакова.

Свойство 2 можно сформулировать иначе: каждый узел B-дерева, кроме листьев, можно рассматривать как упорядоченный список, в котором чередуются ключи и указатели на потомков.

2.3.1 Поиск

Если ключ содержится в корне, он найден. Иначе определяем интервал и идём к соответствующему потомку. Повторяем.

2.3.2 Добавление ключа

Будем называть *деревом потомков некоего узла* поддерево, состоящее из этого узла и его потомков.

Вначале определим функцию, которая добавляет ключ K к дереву потомков узла x. После выполнения функции во всех пройденных узлах, кроме, может быть, самого узла x, будет меньше $2t-1$, но не меньше $t-1$ ключей.

1. Если x — не лист,
   a) Определяем интервал, где должен находиться K. Пусть y — соответствующий потомок.
   b) Рекурсивно добавляем K к дереву потомков y.
   c) Если узел y полон, то есть содержит $2t-1$ ключей, расщепляем его на два. Узел y_1 получает первые $t-1$ из ключей y и первые t его потомков, а узел y_2 — последние $t-1$ из ключей y и последние t его потомков. Медианный из ключей узла y попадает в узел x, а указатель на y в узле x заменяется указателями на узлы y_1 и y_2.
2. Если x — лист, просто добавляем туда ключ K.

Теперь определим добавление ключа K ко всему дереву. Буквой R обозначается корневой узел.

1. Добавим K к дереву потомков R.
2. Если R содержит теперь $2t-1$ ключей, расщепляем его на два. Узел R_1 получает первые $t-1$ из ключей R и первые t его потомков, а узел R_2 — последние $t-1$ из ключей R и последние t его потомков. Медианный из ключей узла R попадает во вновь созданный узел, который становится корневым. Узлы R_1 и R_2 становятся потомками данного узла.

2.3.3 Удаление ключа

Если корень одновременно является листом, то есть в дереве всего один узел, мы просто удаляем ключ из этого узла. В противном случае сначала находим узел, содержащий ключ, запоминая путь к нему. Назовем этот узел x.

Если x — лист, удаляем оттуда ключ. Если в узле x осталось не меньше $t-1$ ключей, мы на этом останавливаемся. Иначе мы смотрим на количество ключей в следующем, а потом в предыдущем узле. Если следующий узел есть, и в нём не менее t ключей, мы добавляем в x ключ-разделитель между ним и следующим узлом, а на его место ставим первый ключ следующего узла, после чего останавливаемся. Если это не так, но есть предыдущий узел, и в нём не менее t ключей, мы добавляем в x ключ-разделитель между ним и предыдущим узлом, а на его место ставим последний ключ предыдущего узла, после чего останавливаемся. Наконец, если и с

предыдущим ключом не получилось, мы объединяем узел x со следующим или предыдущим узлом, и в объединённый узел перемещаем ключ, разделяющий два узла. При этом в родительском узле может остаться только $t-2$ ключей. Тогда, если это не корень, мы выполняем аналогичную процедуру с ним. Если мы в результате дошли до корня, и в нём осталось от 1 до $t-1$ ключей, делать ничего не надо, потому что корень может иметь и меньше $t-1$ ключей. Если же в корне не осталось ни одного ключа, исключаем корневой узел, а его единственный потомок делаем новым корнем дерева.

Если x — не лист, а K — его i-й ключ, удаляем самый правый ключ из поддерева потомков i-го потомка x, или, наоборот, самый левый ключ из поддерева потомков $i+1$-го потомка x. После этого заменяем ключ K удалённым ключом. Удаление ключа происходит так, как описано в предыдущем абзаце.

2.4 Разработка серверной части

Программа, демонстрирующая работу алгоритмов, была разделена на две части: клиентскую и серверную.

Серверная часть отвечает за работу с базой данных по принципу ключ-значение. Это означает, что каждая запись содержит некоторый ключ и некоторое значение, доступное по этому ключу. Реализована работа этой базы данных была с помощью алгоритма B-дерева, описанного в предыдущем разделе. Таким образом, достигается скорость ответа в $O(\log_t n)$ дисковых операций на запрос, где t – параметр дерева, а n – количество записей в нем.

Связь между серверной и клиентской частью обеспечивается с помощью библиотеки WinSock2.

2.5 Разработка клиентской части

Клиентская часть отвечает за работу алгоритмов внешней сортировки и Карацубы во внешней памяти, а также за связь с серверной частью.

Если в текущий момент сервер работает, то клиентская часть может подключиться к нему и совершить необходимые запросы. Клиентская часть также использует библиотеку WinSock2 [3].

Алгоритмы внешней сортировки и Карацубы были реализованы для работы с бинарными файлами, содержащими числа типа int. Пользователь может указать путь к файлам на своей машине и запустить необходимый алгоритм.

Схема взаимодействия клиентской части и серверной части продемонстрирована на рисунке 2.2.

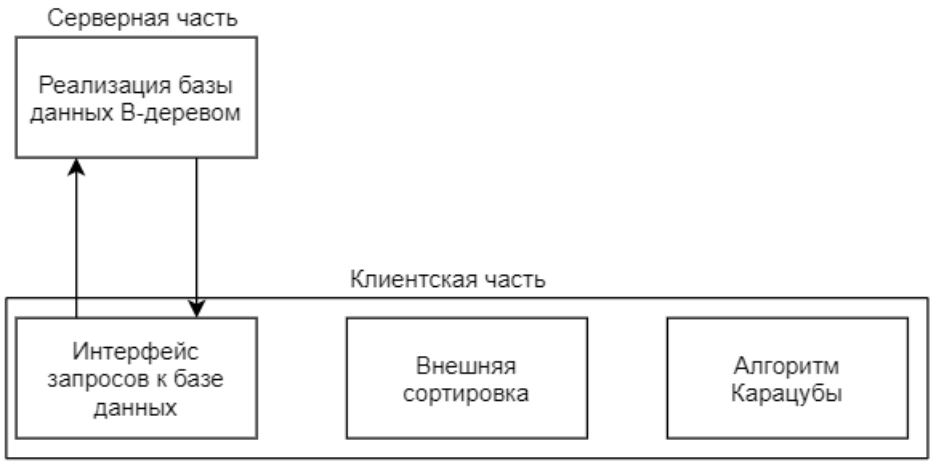

Рисунок 2.2 Взаимоидействие клиентской и серверной части

3. ДЕМОНСТРАЦИЯ РАБОТЫ

Изначально клиентская часть встречает пользователя простым меню, а серверная часть сообщает о ожидании подключения, что продемонстрировано на рисунке 3.1.

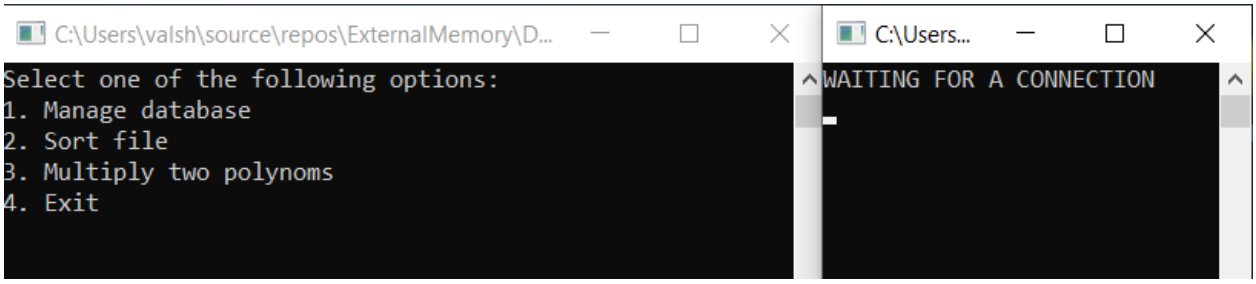

Рисунок 3.1 Начальные сообщения

При входе в режим работы с базой данных, клиентская часть выводит меню возможных запросов, а серверная сообщает о подключении, что продемонстрировано на рисунке 3.2.

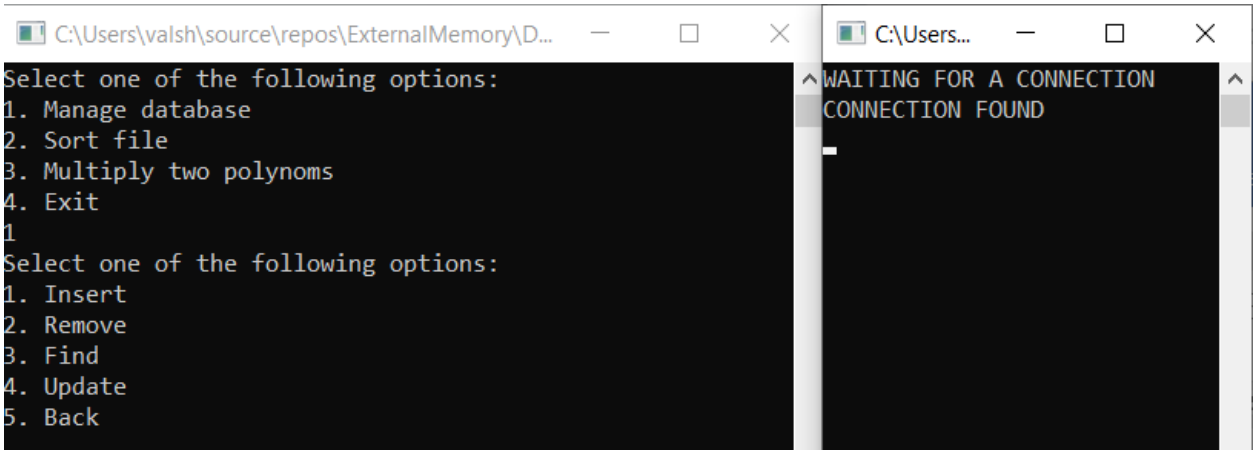

Рисунок 3.2 Режим работы с базой данных

Добавим в базу данных ключ C++ со значением bad с помощью операции Insert. Затем изменим значение этого ключа на good. Серверная часть выводит запросы в сокращенном формате в консоль, что продемонстрировано на рисунке 3.3.

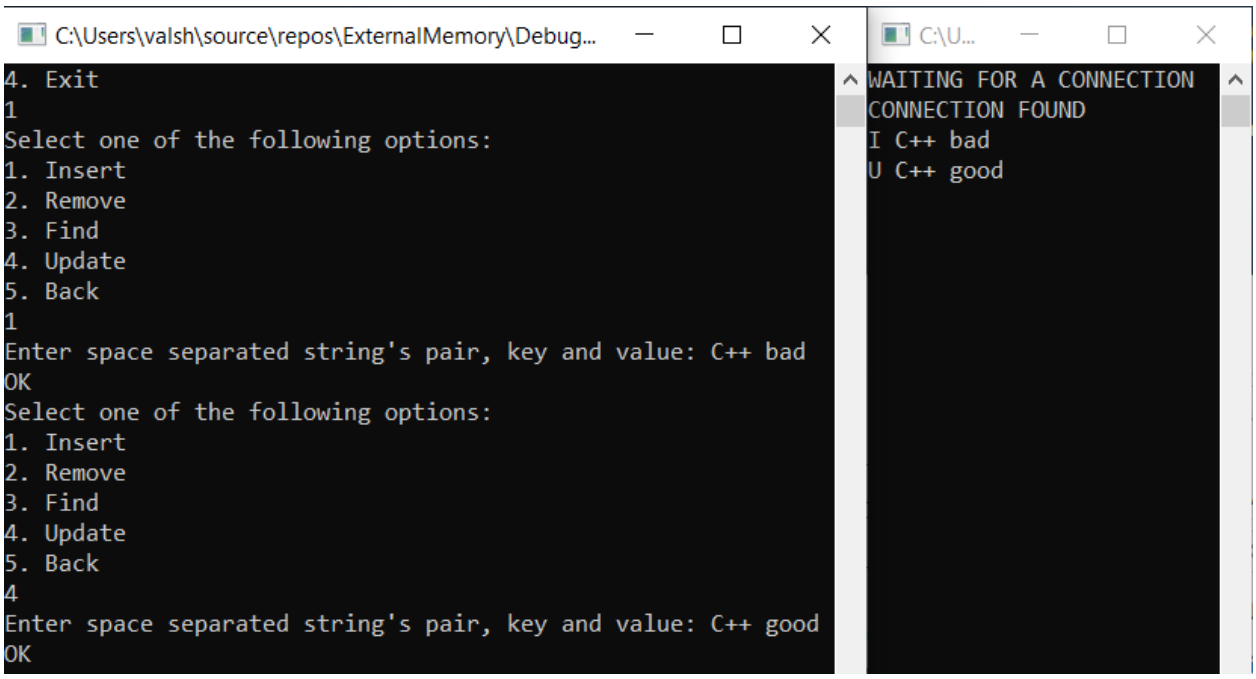

Рисунок 3.3 Добавление ключа и обновление значения

Сделаем поиск по ключу C++ и несуществующему ключу Java. Результаты продемонстрированы на рисунке 3.4.

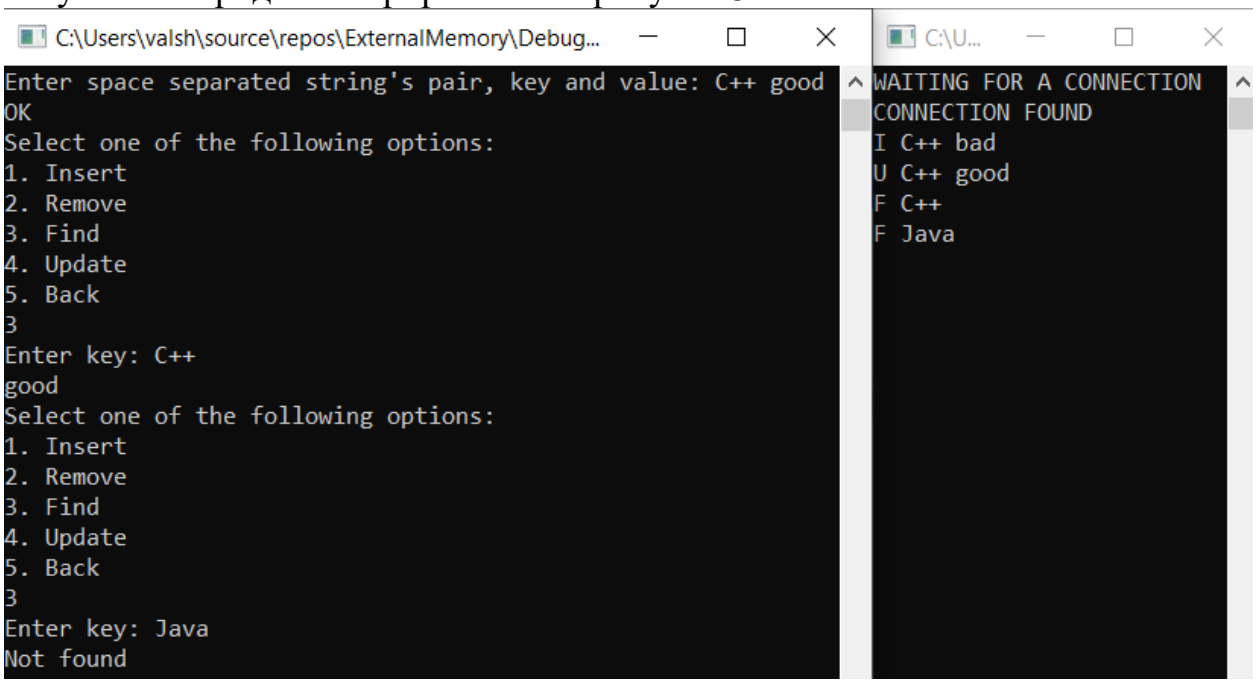

Рисунок 3.4 Поиск по ключам

Удалим ключ C++ и попробуем сделать поиск по нему. Результат показан на рисунке 3.5.

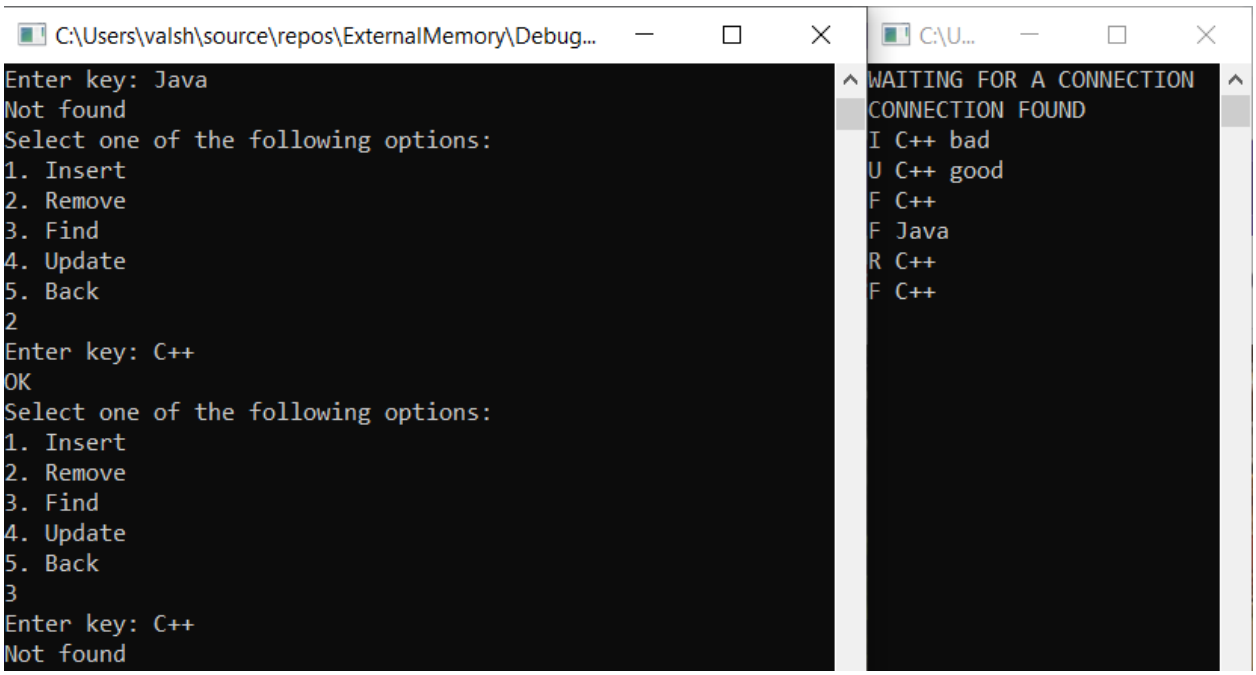

Рисунок 3.5 Удаление и поиск по ключу

Продемонстрируем работу сортировки. В качестве тестового примера используем маленький файл и разрешим использование большого количества памяти. Запуск сортировки и результат продемонстрированы на рисунках 3.6 и 3.7. Файлы бинарные.

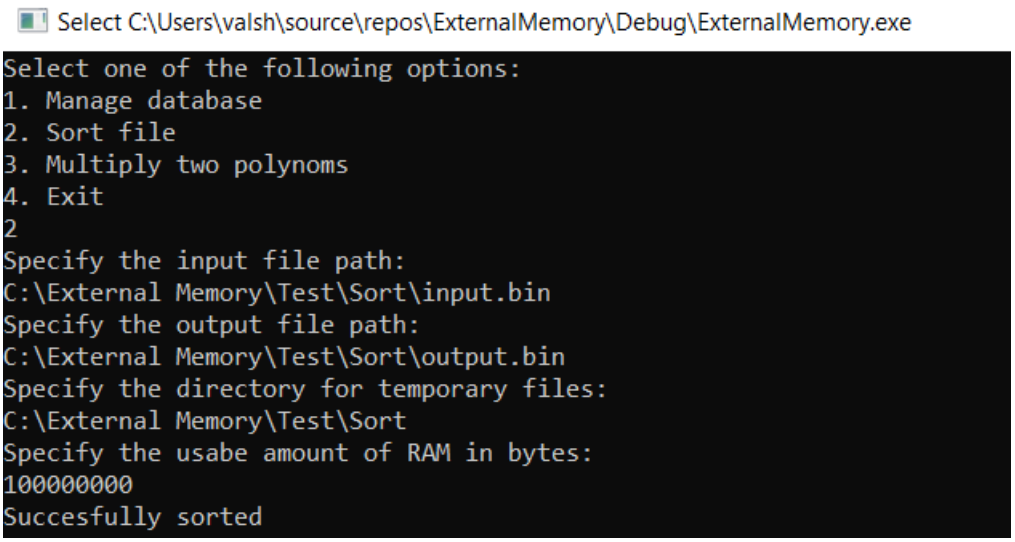

Рисунок 3.6 Запуск внешней сортировки

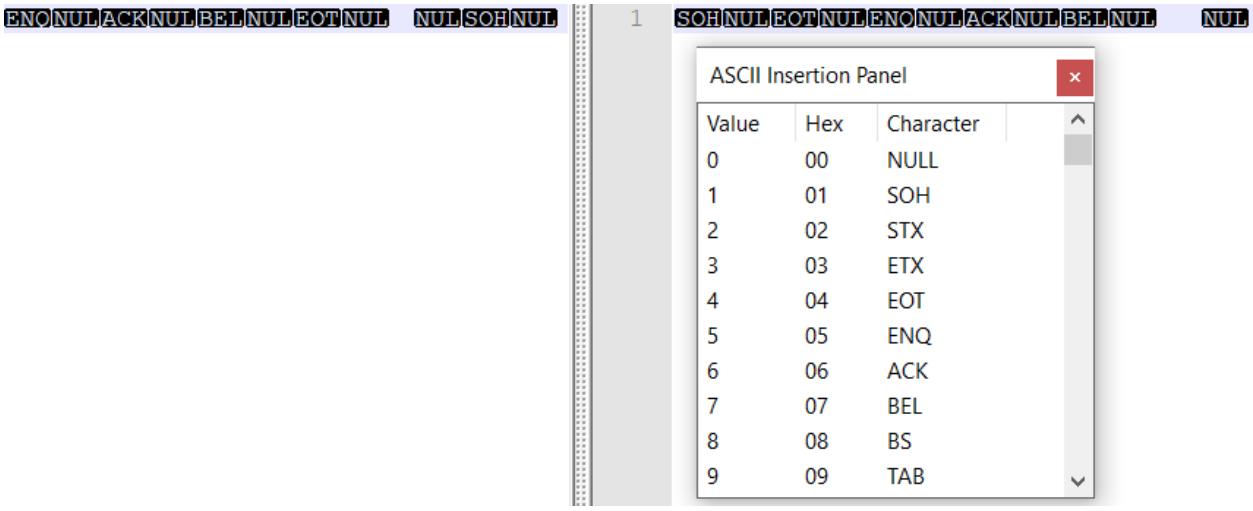

Рисунок 3.7 Результат сортировки

Продемонстрируем работу алгоритма Карацубы. В качестве тестового примера используем маленькие файлы. Запуск алгоритма и результат продемонстрированы на рисунках 3.8 и 3.9. Файлы бинарные. $(2 + 2x) * (2 + 3x + 2x^2) = 4 + 10x + 10x^2 + 4x^3$

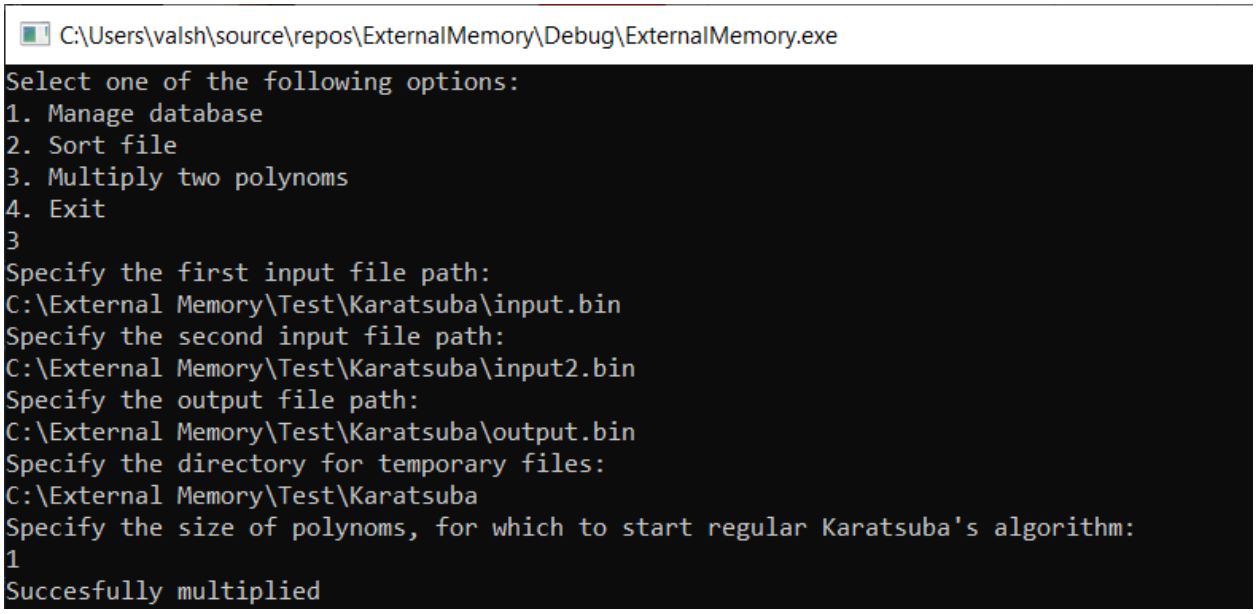

Рисунок 3.8 Запуск алгоритма Карацубы

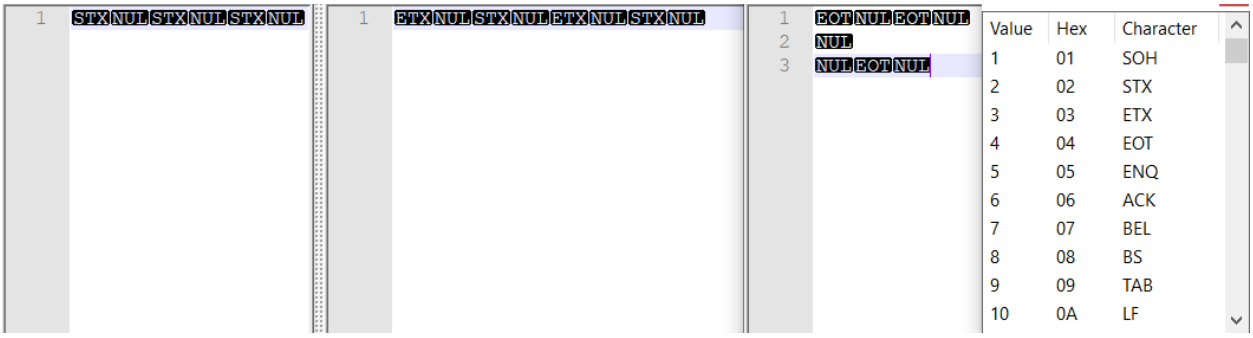

Рисунок 3.9 Результат алгоритма Карацубы

4. АНАЛИЗ ПРОИЗВОДИТЕЛЬНОСТИ АЛГОРИТМОВ

Все тестирование проводилось на ноутбуке с процессором Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz, 1800 Mhz, 4 ядра, 8 логических процессоров и 8 ГБ оперативной памяти. В качестве внешнего хранилища использовался встроенный SSD Intel Pro 6000p 256GB.

Тестирование B-дерева проводилось с параметром 160, так как при этом параметре вершина помещается на одну буферную страницу в памяти. В качестве ключей и значений использовались целые числа типа int, файлы вершин хранились в бинарном формате. Результаты приведены в таблице 3.1.

Таблица 3.1 Производительность B-дерева

	1000 запросов вставки	1000 запросов удаления	1000 запросов поиска	1000 запросов обновления
10^8 записей	8.83 с	8.25 с	4.01 с	5.40 с
10^9 записей	10.56 с	10.53 с	5.76 с	6.79 с

Из результатов видно, что операции поиска выполняются немного быстрее всех остальных. Это связано с тем, что при остальных операциях необходимо совершить перезапись вершин. Также следует отметить, что при многократном обращении к одним и тем же вершинам, то есть при использовании близких ключей, производительность сильно возрастает, т. к. файлы кэшируются операционной системой. При увеличении количества записей в 10 раз, затраченное время увеличивается несущественно, что и является главным преимуществом B-дерева.

Тестирование алгоритма сортировки проводилось на целых числах типа int, файлы хранились в бинарном формате. Результаты приведены в таблице 3.2.

Таблица 3.2 Производительность алгоритма внешней сортировки

	1 гб оперативной памяти	100 мб оперативной памяти	10 мб оперативной памяти	1 мб оперативной памяти
10^8 чисел	13.63 с	15.83 с	16.17 с	23.38 с
10^9 чисел	192.37 с	230.17 с	-	-

Как видно из результатов, при уменьшении количества доступной памяти в 10 раз, время выполнения увеличивается несильно. Поэтому данный алгоритм может справиться с огромными объемами данных, используя адекватное количество оперативной памяти.

Тестирование алгоритма Карацубы проводилось на целых числах типа int, файлы хранились в бинарном формате. Результаты приведены в таблице 3.3.

Таблица 3.3 Производительность алгоритма Карацубы во внешней памяти

	10^7 граница запуска во внутренней памяти	10^6 граница запуска во внутренней памяти	10^5 граница запуска во внутренней памяти	10^4 граница запуска во внутренней памяти
10^6 чисел	-	59.07 с	63.87 с	119.63 с
10^7 чисел	2082.03 с	2100.03 с	2243.63 с	-

Как видно из результатов, при уменьшении границы запуска во внутренней памяти затраченное время увеличивается несильно. Таким образом, на более мощных компьютерах возможно перемножение больших многочленов таким способом за приемлимое время.

ЗАКЛЮЧЕНИЕ

В ходе курсовой работы была разработана библиотека, содержащая реализации трех алгоритмов во внешней памяти: алгоритм внешней сортировки, алгоритм Карацубы, и B-дерево. Также были разработаны клиентское и серверное ПО, демострирующие работу этих алгоритмов. В ходе разработки были получены теоритические знания о алгоритмах во внешней памяти, а также изучены и применены возможности языка C++. Как уже отмечалось, реализация алгоритмов на этом языке не слишком сложна, но при этом сохраняет высокие показатели производительности. Отдельна была изучена библиотека WinSock2, позволяющая установить связь между клиентом и сервером.

На данный момент, обработка больших данных является задачей первостепенной важности. Не всегда имеется возможность разместить эти данные полностью в оперативной памяти. В таких случаях, применение алгоритмов во внешней памяти является необходимым. Например, в современных системах управления базами данных применяется улучшенная реализация B-дерева, называемая B^+-дерево. Быстрое перемножение многочленов позволяет совершать быстрое пермножение чисел, которое существенно ускоряет решение многих важных задач в теории чисел. Сортировка зачастую является частью других алгоритмов, например, она применяется при загрузке большого объема данных в базу данных за раз. Несмотря на то, что алгоритмы во внешней памяти позволяют обрабатывать большие объемы данных, следует помнить, что они часто медленнее своих классических аналогов и, если данные возможно разместить в оперативной памяти, то не следует использовать алгоритмы во внешней памяти.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. WIKIPEDIA [Электронный ресурс] – WIKIPEDIA: https://ru.wikipedia.org
2. ВИКИКОНСПЕКТЫ [Электронный ресурс] – B-дерево: https://neerc.ifmo.ru/wiki/index.php?title=B-дерево
3. C++ REFERENCE [Электронный ресурс] – C++ REFERENCE: https://en.cppreference.com/w/

ПРИЛОЖЕНИЕ. Исходный код алгоритма внешней сортировки

```
const int SYSTEM_BLOCK_SIZE = 1024;
const int MAX_BUFFER_SIZE = 4096;
const int ADDITIONAL_RAM_PER_FILE = 4 * sizeof(int) + sizeof(FILE*);
const int INDEPENDENT_RAM_USAGE = 200;
const int MINIMAL_FILE_NEED = 3;

std::string getTempFileNameSort(const std::string& tempFileDirectory, int index)
{
	return (tempFileDirectory + "/temp" + std::to_string(index) + ".bin");
}

int splitAndSort(FILE* input, const std::string& tempFileDirectory, int
availableRAM)
{
	int length = availableRAM / sizeof(int);
	int* buffer = new int[length];
	if (length == 0)
	{
		return 0;
	}
	int counter = 0;
	length = fread(buffer, sizeof(int), length, input);
	while (length > 0)
	{
		std::sort(buffer, buffer + length);
		FILE* current;
		fopen_s(&current, getTempFileNameSort(tempFileDirectory,
counter).c_str(), "wb");
		if (current == NULL)
		{
			return 0;
		}
		fwrite(buffer, sizeof(int), length, current);
		fclose(current);
		counter++;
		length = fread(buffer, sizeof(int), length, input);
	}
	fclose(input);
	delete[] buffer;
	return counter;
}
```

```
bool mergeFiles(FILE *mainOutput, const std::string& tempFileDirectory, int availableRAM, int numberOfFiles)
{
	int bufferSize = SYSTEM_BLOCK_SIZE;
	int wayOfMerging = std::min(int(availableRAM / (bufferSize * sizeof(int) + ADDITIONAL_RAM_PER_FILE) - 1), numberOfFiles);
	for (int coefficient = 2; coefficient <= availableRAM / SYSTEM_BLOCK_SIZE; coefficient++)
	{
		bufferSize = coefficient * SYSTEM_BLOCK_SIZE;
		wayOfMerging = std::min(int(availableRAM / (bufferSize * sizeof(int) + ADDITIONAL_RAM_PER_FILE) - 1), numberOfFiles);
		if (wayOfMerging < numberOfFiles)
		{
			bufferSize = (coefficient - 1) * SYSTEM_BLOCK_SIZE;
			wayOfMerging = std::min(int(availableRAM / (bufferSize * sizeof(int) + ADDITIONAL_RAM_PER_FILE) - 1), numberOfFiles);
			break;
		}
	}
	if (wayOfMerging < MINIMAL_FILE_NEED - 1)
	{
		return false;
	}
	bool lastMerge = false;
	int firstFileIndex = 0;
	std::priority_queue< std::pair<int, int> > heap;
	BufferedIntReader** readers = new BufferedIntReader*[wayOfMerging];
	for (int i = 0; i < wayOfMerging; i++)
	{
		readers[i] = new BufferedIntReader(nullptr, bufferSize);
	}
	BufferedIntWriter* writer = new BufferedIntWriter(nullptr, bufferSize);
	while (numberOfFiles > 1)
	{
		if (wayOfMerging >= numberOfFiles)
		{
			lastMerge = true;
		}
		for (int i = firstFileIndex; i < firstFileIndex + numberOfFiles; i += wayOfMerging)
		{
			int numberOfInputFiles = 0;
			if (lastMerge)
```

```
                writer->setOutput(mainOutput);
            else
            {
                FILE* currentOutput;
                fopen_s(&currentOutput,
getTempFileNameSort(tempFileDirectory, (firstFileIndex + numberOfFiles + i /
wayOfMerging)).c_str(), "wb");
                if (!currentOutput)
                {
                    for (int i = 0; i < wayOfMerging; i++)
                        delete readers[i];
                    delete[] readers;
                    delete writer;
                    return false;
                }
                writer->setOutput(currentOutput);
            }
            for (int j = i; j < std::min(i + wayOfMerging, firstFileIndex +
numberOfFiles); j++)
            {
                int index = j - i;
                FILE* currentInput;
                fopen_s(&currentInput,
getTempFileNameSort(tempFileDirectory, j).c_str(), "rb");
                numberOfInputFiles++;
                if (!currentInput)
                {
                    for (int i = 0; i < wayOfMerging; i++)
                        delete readers[i];
                    delete[] readers;
                    delete writer;
                    return false;
                }
                readers[index]->setInput(currentInput);
            }
            heap = std::priority_queue< std::pair<int, int> >();
            for (int j = 0; j < numberOfInputFiles; j++)
            {
                int current = readers[j]->next();
                if(current != INT_MAX)
                    heap.push({ -current, j });
            }
            int cnt = 0;
            while (!heap.empty())
```

```
                {
                        writer->writeNext(-heap.top().first);
                        int index = heap.top().second;
                        heap.pop();
                        int current = readers[index]->next();
                        if (current == 1)
                                cnt++;
                        if (current != INT_MAX)
                                heap.push({ -current, index });
                }
                for (int j = 0; j < numberOfInputFiles; j++)
                {
                        remove(getTempFileNameSort(tempFileDirectory, i +
j).c_str());
                }
        }
        firstFileIndex += numberOfFiles;
        numberOfFiles = (numberOfFiles + wayOfMerging - 1) /
wayOfMerging;
    }
    for (int i = 0; i < wayOfMerging; i++)
        delete readers[i];
    delete writer;
    delete[] readers;
    return true;
}

bool externalSort(std::string inputFilePath, std::string outputFilePath, std::string
tempFileDirectory, int availableRAM)
{
    availableRAM -= INDEPENDENT_RAM_USAGE;
    FILE* input;
    FILE* output;
    fopen_s(&input, inputFilePath.c_str(), "rb");
    if (!input)
    {
        return false;
    }
    int numberOfFiles = splitAndSort(input, tempFileDirectory, availableRAM);
    if (numberOfFiles == 0)
        return false;
    if (numberOfFiles == 1)
    {
        remove(outputFilePath.c_str());
```

```
		int result = rename(getTempFileNameSort(tempFileDirectory,
0).c_str(), outputFilePath.c_str());
		if (result == 0)
			return true;
		else
			return false;
	}
	fopen_s(&output, outputFilePath.c_str(), "wb");
	if (!output)
	{
		return false;
	}
	return mergeFiles(output, tempFileDirectory, availableRAM,
numberOfFiles);
}
```