МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение

высшего образования

«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ

ТОМСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

Институт кибернетики

Направление «Программная инженерия»

Кафедра прикладной математики

КУРСОВАЯ РАБОТА

по дисциплине

«Алгоритмы и структуры данных»

АБСТРАКТНЫЕ ТИПЫ ДАННЫХ

Задание №8 Заказ товаров почтой

Студент:

А.В. Нечеухин

Подпись: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

«\_\_\_»\_\_\_\_\_\_\_\_\_\_20\_\_\_г.

Руководитель:

доцент кафедры программной инженерии О.Б. Фофанов

Подпись: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

«\_\_\_»\_\_\_\_\_\_\_\_\_\_20\_\_\_г.

Томск 2016

Содержание

Введение

1 МОДЕЛИРОВАНИЕ АБСТРАКТНЫХ ТИПОВ ДАННЫХ

1.1 Общие теоретические положения и постановка задачи

1.2 Описания и алгоритмы

1.2.1 Алгоритм работы вычислительной системы

1.2.2 Алгоритм работы процессора

1.2.3 Описание и работа стека

1.2.3 Описание и работа очереди

1.3 Результат выполнения и анализ работы вычислительной системы

2 ПОИСК ИНФОРМАЦИИ В РАЗЛИЧНЫХ СТРУКТУРАХ ДАННЫХ

2.1 Общие теоретические положения и постановка задачи

2.2 Описания и алгоритмы

2.2.1 Алгоритм линейного поиска

2.2.2 Алгоритм быстрого линейного поиска

2.2.3 Алгоритм бинарного поиска

2.2.4 Алгоритм блочного поиска

2.2.5 Алгоритм Боуэра-Мура

2.3 Результат выполнения и сравнительный анализ

3 ИССЛЕДОАВНИЕ АЛГОРИТМОВ СОРТИРОВОК ДЛЯ РАЗЛИЧНЫХ СТРУКТУР И РАЗМЕРНОСТЕЙ ДАННЫХ

3.1 Общие теоретические положения и постановка задачи

3.2 Описания и алгоритмы

3.2.1 Алгоритм сортировки вставками

3.2.2 Алгоритм сортировки Шелла

3.2.3 Алгоритм сортировки Хоара

3.3 Результат выполнения и сравнительный анализ

4 РЕАЛИЗАЦИЯ СТРУКТУРЫ ДАННЫХ ТИПА ДЕРЕВО И ТИПОВЫЕ АЛГОРИТМЫ ИХ ОБРАБОТКИ

4.1 Общие теоретические положения и постановка задачи

4.2 Описания и алгоритмы

4.2.1 Алгоритм добавления элемента в дерево

4.2.2 Алгоритм удаления элемента из дерева

4.2.3 Алгоритм поиска элемента в дереве

4.3 Результат выполнения и сравнительный анализ

5 РЕАЛИЗАЦИЯ ФУНКЦИЙ ХЕШИРОВАНИЯ И РАЗЛИЧНЫХ МЕТОДОВ РАЗРЕШЕНИЯ КОЛЛИЗИЙ

5.1 Общие теоретические положения и постановка задачи

5.2 Описания и алгоритмы

5.3 Результат выполнения и сравнительный анализ

Заключение

Список используемой литературы

Приложение А Листинг программы для раздела 1

Приложение Б Листинг программы для раздела 2

Приложение В Листинг программы для раздела 3

Приложение Г Листинг программы для раздела 4

Приложение Д Листинг программы для раздела 5

Введение

Развитие информационных технологий привело к тому, что общество в двадцать первом веке принято называть «информационным». Довольно сложно представить сегодня организацию, предприятие или фирму в которых не задействованы автоматические или автоматизированные системы, управление которыми осуществляется при помощи компьютеризированных вычислительных систем. Каждая такая система состоит из программной и аппаратной частей. Многофункциональные вычислительные устройства (персональные компьютеры, смартфоны, смарт-телевизоры и др.) способны выполнять различные функции от выхода во всемирную сеть интернет и общение между другими устройствами на огромных расстояниях, до поддержания определенных условий, например, в ядерном реакторе или моделирование поведения системы и определение координат, времени и возможных разрушений, причиненных падением метеорита. Все это возможно не только благодаря большой вычислительной скорости и мощности современных персональных компьютеров (ПК). Большую роль играет рынок программного обеспечения (ПО). Различным фирмам и компаниям, специализирующихся в одинаковых сферах, требуется совершенно различное программное обеспечение, удовлетворяющее их потребностям и нуждам.

В связи с этим существует большой спрос на программное обеспечение, разрабатываемое для самых различных сфер жизнедеятельности и организаций, которые в этих сферах специализируются. Следовательно, умение разрабатывать качественное ПО, удовлетворяющее требованиям заказчика, является показателем профессионализма и мастерства разработчика. Однако такая разработка невозможна без знания основных алгоритмов.

Сегодня существует большое количество алгоритмов для выполнения одних и тех же задач. В качестве примера можно привести большое количество алгоритмов сортировки. Но ни один из их не является «устаревшим», все они используются для той или иной задачи. Более простые способы используются для обучения, для быстрой реализации, более сложные для более быстрого выполнения программы, или для экономии памяти, используемой программой в состав которой этот алгоритм входит.

В связи с вышесказанным прослеживается проблема выбора алгоритма для определенной задачи. И разработчик должен эту проблему постоянно решать, а для этого необходимо не только знать устройство алгоритма, а также потребление им ресурсов, таких как время и память. Разработчик должен уметь сравнить алгоритмы, выполняющие одинаковые задачи и проанализировав результаты выбрать подходящий для конкретной задачи.

*Целью* курсовой работы является приобретение практических навыков реализации различных алгоритмов, таких так алгоритмы поиска, сортировки, реализация динамических структур данных и др.

В качестве задач можно выделить следующие:

1. Моделирование абстрактных типов данных для различных реализаций.

2. Поиск информации в различных структурах данных.

3. Исследование эффективности алгоритмов сортировок для различных размерностей данных

4. Реализация структуры данных типа «дерево» и типовые алгоритмы их обработки.

5. Реализация функций хеширования и различных методов разрешения коллизий.

1 МОДЕЛИРОВАНИЕ АБСТРАКТНЫХ ТИПОВ ДАННЫХ

1.1 Общие теоретические положения и постановка задачи

Моделируемая вычислительная система состоит из процессора и трех очередей (Рисунок 1):

- очередь «Высокий приоритет»;

- очередь «Средний приоритет»;

- очередь «Низкий приоритет».

В систему поступают запросы, которые создаются генератором задач. Каждая зада может быть описана следующим образом:

Type process

String: Name; //Имя процесса

Integer: Time; //Время выполнения операции

Priorities: Priority; //Приоритет операции

Все поступающие в систему процессы (задачи) ставятся в соответствующую приоритету очередь. Сначала обрабатывается очередь с высокоприоритетными задачами. В случае пустой очереди управление передается на обработку очереди с задачами, имеющими средний приоритет. В случае пустой очереди с задачами, имеющими средний приоритет, управление передается низкоприоритетным задачам. Когда все задачи выполнены, то процессор переходит в состояние ожидания (процессор свободен).

Когда поступает задача с более высоким приоритетом, то процессор прерывает выполнение текущей задачи, помещает текущую задачу в стек, и задача с более высоким приоритетом начинает исполняться. В случае выполнения всех высокоприоритетных задач, вычислительная система начинает обработку следующую по приоритету очередь, либо помещается на выполнение задача из стека в случае если эта задача имеет более высокий приоритет.

Сценарий работы системы (с шаговым временем) можно описать следующим образом:

- генерируется одна новая задача, в формате, описанном выше;

- задача помещается в соответствующую приоритету очередь;

- моделируется обслуживание очередей вычислительной системы;

- генерация следующей задачи и встраивание в соответствующую очередь;

- произвести генерацию не менее 10000 задач.

Генератор задач

Очередь «высокий приоритет»

Очередь «средний приоритет»

Очередь «низкий приоритет»

Процессор

Стек

Рисунок 1 – Схема вычислительной системы

Система построена на стеке, организованном на односвязном линейном списке и кольцевых очередях на массиве в статической памяти.

1.2 Описания и алгоритмы

1.2.1 Алгоритм работы вычислительной системы

Функция generateProcess() случайным образом выбирает время работы задачи, и ее приоритет. Задает имя задачи, и сформированная задача возвращается в вызывающий метод. Стоит также отметить, что генератор задач также может не сгенерировать ни одной задачи.

Функция сompleteTask() поочередно выбирает задачу из очередей в соответствии с приоритетом и пытается поместить эту задачу на выполнение процессором. Если выполняемая в процессоре задача имеет меньший приоритет, чем текущая задача, помещаемая на выполнение процессором, то выполняемая задача прерывается, а текущая удаляется из очереди задач. В случае, если все очереди пусты, система выдает сообщение об ожидании поступления новой задачи.

Основной метод main(), запускаемый при работе программы создает три очереди с различными приоритетами. Создается также «процессор» - объект, обрабатывающий очереди [подраздел 1.2.2] и файл, в котором ведется статистика работы системы в каждый момент времени.

После предварительной подготовки системы запускается цикл генерации и выполнения задач. Данный цикл оканчивает свою работу после выполнения всех задач в очередях и только после того, как счетчик задач превысит установленное значение. Т.е. иными словами при превышении счетчика необходимых задач система продолжает выполнять оставшиеся в очередях задачи и только после этого завершает свою работу.

В каждый момент времени перед генерацией каждой задачи система записывает свое состояние в файл после чего запускается генерация задачи генерацию задачи, распределение сгенерированной задачи в соответствующую приоритету очередь. После чего вызывается метод completeTask(), описанный выше. В данном разделе.

1.2.2 Алгоритм работы процессора

Процессор не может работать без общей системы, поэтому в данном разделе будут описаны лишь методы, реализующие поведение процессора.

Метод getCurrentTask() возвращает задачу, которая выполняется на процессоре в данный момент, при этом проверяя не вышло ли время ее выполнения. Если время выполнения задачи вышло, то процессор освобождается.

Метод putTask() помещает задачу на выполнение процессором, при этом запускает счетчик времени для текущей задачи.

Метод isFree() проверяет свободен ли процессор в данный момент времени. Если процессор свободен, или есть возможность его освободить, то запускаются задачи из стека (если они существуют).

Метод interrupt() перед прерыванием проверяется состояние процессора и если он свободен, то задача помещается на выполнение процессором. В случае если процессор занят, то проверяется приоритет двух задач и если у текущей задачи приоритет выше той, что выполняется на процессоре в данный момент времени, то выполняемая задача помещается в стек, а претендующая задача ставится на выполнение процессором. В случае, если претендующая задача имеет меньший или одинаковый приоритет с выполняемой на процессоре задачей, то выдается сообщение о невозможности прерывания текущей задачи.

1.2.3 Описание и работа стека

Стек основан на односвязном линейном списке в качестве элемента стека выступает структура MyNode которая хранит ссылку на предыдущую такую же структуру и сами данные:

MyNode <Type>

MyNode PreviousElement;

Type Data;

Стек не содержит каких-либо особенностей, требующий комментирования или описания алгоритма. Кроме стандартных методов, с которыми работает стек реализован дополнительный метод getStrackElements(), возвращающий список всех элементов, находящихся в стеке. Данный метод необходим для распечатки текущего состояния стека в системе.

1.2.3 Описание и работа очереди

Очередь, организованная в данной системе построена на элементах в статической памяти с динамическим ее выделением при недостаточном объеме. Также данная очередь является кольцевой и, как и другие объекты типонезависимой, т.е. она позволяет хранить объекты любого типа. При организации кольцевой очереди был использован дополнительный индекс, хранящий начало текущей очереди.

Метод enqueue() добавляет элемент в очередь. Сначала проверяется достаточно ли выделено памяти для помещения нового элемента в очередь и в случае недостатка памяти выполняется проверка зацикленности очереди. В случае зацикливания очередь раскручивается, и все элементы очереди копируются в новый, более объемный массив данных. Если очередь была не зацикленной, то производится простое копирование в более объемный массив.

Память под новый элемент есть, однако далее необходима проверка в какое место вставить объект, т.к. очередь может быть уже зациклена. Тут помогает указатель, который хранит индекс начала очереди и если он указывает на последний элемент в памяти, то необходимо зациклить очередь.

Метод dequeue() удаляет элемент из очереди и возвращает этот элемент вызывающему методу. Перед удалением выполняется проверка и если очередь пуста вызывается исключение. Дальнейшее удаление изменяет размер очереди и смещает индекс начала. В случае если удаляется первый элемент из памяти, то очередь раскручивается и превращается в обычный массив с той лишь разницей, что начальные элементы у него могут быть не определены.

Метод front() возвращает элемент, на который ссылается указатель «начало очереди». В случае пустой очереди вызывается исключение.

Кроме основных методов, необходимых для работы очереди реализован дополнительный метод getQueueElements(), который возвращает список всех элементов, находящихся в очереди без изменения самих данных. Тут также, как и в случаях с добавлением элемента в очередь и удалением элемента из очереди необходимо решать вопросы с зацикливанием и корректным порядком обработки данных.

1.3 Результат выполнения и анализ работы вычислительной системы

В ходе работы вычислительная система постоянно прописывает каждое выполняемое действие в консоль [Рисунок 1], что облегчает отладку приложения, т.к. моделируемые данные случайны и предсказать поведение такой системы в случае ошибки невозможно.

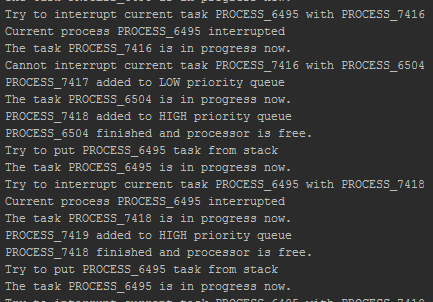


Рисунок 1 – Результат распечатки лога программы

Вычислительная система во время работы формирует файл .csv, который легко переводится в .xml. Распечатка файла приведена в таблице 1, а в таблице 2 приведена сводная статистика по всем объектам системы

Таблица 1. Результат моделирования вычислительной системы

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| *Время* | *Элемент* | *Размер* | *Имя, время* | *Имя, время* | *Имя, время* | *Имя, время* | *Имя, время* | *Имя, время* | *Имя, время* |
| 566 | LowPriorityQueue | 79 | PROCESS 121,1 | PROCESS 124,1 | PROCESS 126,1 | PROCESS 130,2 | *PROCESS 134,1* | PROCESS 137,2 | PROCESS 143,2 |
|  |  |  |  |  |  |  |  |  |  |
| 566 | MiddlePriorityQueue | 6 | PROCESS 364,1 | PROCESS 368,2 | PROCESS 370,1 | PROCESS 375,2 | *PROCESS 376,1* | PROCESS 382,2 |  |
| 566 | Stack | 1 | PROCESS 118,2 |  |  |  |  |  |  |
| 566 | Processor | 1 | PROCESS 360,1 |  |  |  |  |  |  |
| 566 | HighPriorityQueue | 0 |  |  |  |  |  |  |  |
| 567 | LowPriorityQueue | 80 | PROCESS 121,1 | PROCESS 124,1 | PROCESS 126,1 | PROCESS 130,2 | *PROCESS 134,1* | PROCESS 137,2 | PROCESS 143,2 |
| 567 | MiddlePriorityQueue | 5 | PROCESS 368,2 | PROCESS 370,1 | PROCESS 375,2 | PROCESS 376,1 | *PROCESS 382,2* |  |  |
| 567 | Stack | 1 | PROCESS 118,2 |  |  |  |  |  |  |
| 567 | Processor | 1 | PROCESS 364,1 |  |  |  |  |  |  |
| 567 | HighPriorityQueue | 0 |  |  |  |  |  |  |  |

Таблица 2. Сводная статистика объектов системы

|  |  |  |  |
| --- | --- | --- | --- |
| *Имя объекта* | *Количество машинного времени, когда объект был заполнен* | *Средний размер очереди* | *Время работы процессора (с)* |
| HighPrior | 112 | 1,007 | 36 |
| MiddlePrior | 2375 | 1,29 |  |
| LowPrior | 15226 | 136,26 |  |
| Stack | 4736 | 1,127 |  |

В первой части разработана система имитирующую однопроцессорную систему с различными приоритетами задач. Задачи генерируются каждый отдельный промежуток времени и после каждой генерации задач проверяется освобождение системы. Данный подход можно было изменить, используя принципы объектно-ориентированного программирования и паттерны (например, оповещатель), однако это усложнило бы понимание кода и его структуру.

2 ПОИСК ИНФОРМАЦИИ В РАЗЛИЧНЫХ СТРУКТУРАХ ДАННЫХ

2.1 Общие теоретические положения и постановка задачи

Поиск информации в различных структурах данных является одной из самых востребованных операций на сегодняшний день по обработке данных. Связано это, в первую очередь, с возрастающим объемом информации и с человеческой потребностью в получении этой информации.

В данном разделе будут рассмотрены алгоритмы поиска элемента в массиве и проведен сравнительный анализ следующих алгоритмов:

1. Алгоритмы поиска в неупорядоченном массиве

- линейный;

- быстрый линейный (с барьером).

2. В упорядоченном массиве:

- быстрый линейный (с барьером);

- бинарный;

- блочный.

Необходимо построить графики зависимости количества операций сравнения от количества элементов в массиве. Определить временную сложность алгоритмов.

Кроме поиска элемента в массиве данных необходимо также изучить алгоритм поиска контекста в больших объемах текстовой информации, построить графики зависимости количества операций сравнения от количества символов в массиве и определить временную сложность алгоритма Боуэра-Мура.

2.2 Описания и алгоритмы

2.2.1 Алгоритм линейного поиска

Данный алгоритм является самым простым алгоритмом из данной серии. Его работа заключается в том, что все элементы массива, начиная с первого, последовательно сравниваются с искомым элементом. Сравнение элементов продолжается до тех пор, пока не будут просмотрены все элементы или очередной элемент массива не равен искомому

2.2.2 Алгоритм быстрого линейного поиска

Данный алгоритм является улучшением алгоритма линейного поиска. Любой алгоритм из серии поиска содержит блок проверки на окончание массива. В алгоритме линейного поиска эта проверка осуществляется каждый раз перед обращением к очередному элементу. Однако проверка на окончание может осуществляться не при каждом сравнении. Для этого в конец массива включается (N+1) элемент, равный искомому, где N – размерность исходного массива. Т.е. другими словами добавляется искомый элемент в конец массива.

Достоинства данного подхода состоят в том, что проверка на окончание массива происходит единожды, лишь при совпадении очередного элемента с искомым. Если этот элемент находится внутри массива, то поиск заканчивается удачно и элемент считается найденным. Если же этот элемент оказался последним (N+1), то искомого элемента в массиве нет.

2.2.3 Алгоритм бинарного поиска

Принцип, лежащий в основе данного алгоритма поиска (и некоторых других алгоритмов), состоит в том, что иногда удается последовательно уменьшать объем задачи до такой степени, что ее решение в конце концов становится тривиальным. Другое название у данного принципа – принцип «Разделяй и властвуй». Главный шаг при бинарном поиске – получить элемент из середины массива и, если он не равен искомому, то в зависимости от его значения продолжить рассматривать подмассив, находящийся справа или слева от выбранного значения. Повторное выполнение этого шага быстро сокращает размер области поиска.

Последовательное описание алгоритма бинарного поиска:

1. Определить середину массива.

2. Если элемент, находящийся в середине массива совпадает с искомым, поиск завершен.

3. Если значение из середины массива больше искомого, применить бинарный поиск к первой половине массива.

4. Если ключ из середины массива меньше искомого, бинарный поиск необходимо применить ко второй половине массива.

5. Продолжить выполнение пунктов 1-4, пока размер области поиска не уменьшится до нуля. Если размер области поиска равен нулю – значения в массиве нет.

Стоит также отметить, что данный алгоритм может быть реализован как с использованием рекурсивной реализации, так и с развернутой при помощи циклов.

2.2.4 Алгоритм блочного поиска

Блочный поиск состоит в том, что массив, упорядоченный по возрастанию, разбивается на определённое число блоков. В процессе поиска искомый элемент последовательно сравнивается с последним элементом блоков. Если искомый элемент меньше последнего элемента очередного блока, то искомый элемент может находиться только внутри этого блока. Для поиска элемента в блоке можно применить линейный поиск.

2.2.5 Алгоритм Бойера-Мура

Данный алгоритм разработан двумя учеными – Бойером и Муром, считается что данный алгоритм является наиболее быстрым для поиска строки в подстроке. Основной особенностью данного алгоритма является то, что сравнение с шаблоном или паттерном выполняется не слева направо, как это происходит в других алгоритмах, а справа налево.

Данный алгоритм считается наиболее эффективный для поиска данных в текстовых редакторах, браузерах и других подобных приложениях, использующих поиск «Ctrl+F».

Алгоритм сравнивает символы шаблона с символами исходной строки. Сравнение производится справа налево. Если символы шаблона и исходной строки совпали, то сравнение продолжается со следующего символа шаблона (предпоследнего символа) и так далее. Если все символы совпали значит подстрока найдена и поиск окончен.

В случае несовпадения шаблона с исходной строкой алгоритм использует правило сдвига плохого символа. Суть его заключается в том, что таблица «плохих символов» содержит последнее вхождение символа в шаблон (за исключением последнего символа) для каждого символа из алфавита. Если символ не встречается, то для него ставится в соответствие длина шаблона. Если символ шаблона на текущем шаге не совпал с символом исходной строки, то шаблон сдвигается слева направо на столько символов, сколько указано в таблице «плохих символов». Если в таблице нет такого символа, то шаблон сдвигается полностью на свою длину. Пример алгоритма приведен ниже.

a-2; б-3 – значения таблицы «плохих символов»

вабавабаавваабавбаб

ааб

вабавабаавваабавбаб

ааб

вабавабаавваабавбаб

ааб

вабавабаавваабавбаб

ааб

вабавабаавваабавбаб

ааб

2.3 Результат выполнения и сравнительный анализ

Произведены практические замеры времени работы алгоритмов (таблица 3). Из таблицы видно, что сортированный линейный поиск работает хуже линейного поиска. Это связано с тем, что, найдя элемент, алгоритм линейного поиска прекращает работу, а шаблон поисковый (элемент) в сортированных данных может находится намного дальше чем в исходном, что и приводит к увеличению времени работы сортированного линейного поиска. Хорошие результаты показал блочный поиск с выбором размера блока = 10.

Результат выполнения работы программы представлен на рисунке 2. В качестве найденного элемента указывается индекс этого элемента. Разница в индексах, представленных на рисунке означает, что некоторые алгоритмы используют сортированные данные, другие исходные.

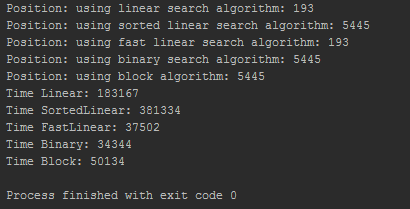


Рисунок 2 – Результат выполнения программы

Талица 3. Практические результаты времени работы различных алгоритмов

| *Размер-ность данных* | *Время линейного поиска (мкс)* | *Время линейного поиска в сортированных данных (мкс)* | *Время линейного поиска с барьером (мкс)* | *Время бинарного поиска (мкс)* | *Время блочного поиска (мкс)* |
| --- | --- | --- | --- | --- | --- |
| 0 | 0 | 0 | 0 | 0 | 0 |
| 1000 | 148,428 | 59,213 | 24,08 | 18,554 | 20,527 |
| 2000 | 151,586 | 54,476 | 25,265 | 18,159 | 19,343 |
| 3000 | 223,827 | 96,715 | 68,293 | 18,159 | 23,685 |
| 4000 | 160,271 | 45,002 | 32,765 | 18,553 | 19,738 |
| 5000 | 150,797 | 117,637 | 28,423 | 18,948 | 26,843 |
| 6000 | 283,04 | 350,938 | 134,217 | 18,554 | 44,608 |
| 7000 | 230,933 | 401,072 | 87,241 | 20,132 | 50,529 |
| 8000 | 155,534 | 66,713 | 29,607 | 18,948 | 20,528 |
| 9000 | 352,123 | 151,981 | 165,403 | 19,737 | 43,424 |
| 10000 | 367,913 | 318,568 | 202,905 | 19,343 | 54,476 |
| 20000 | 643,847 | 108,163 | 395,941 | 26,054 | 28,422 |
| 30000 | 204,878 | 1355,988 | 58,819 | 24,869 | 135,796 |
| 40000 | 275,145 | 649,374 | 140,139 | 55,265 | 121,98 |
| 50000 | 581,082 | 1435,333 | 350,544 | 24,475 | 165,008 |
| 60000 | 354,096 | 1655,608 | 179,219 | 32,765 | 147,244 |

На основании данных таблицы построен график в одних осях. На графике исключены результаты работы алгоритма сортированного линейного поиска, чтобы наглядность не терялась (рисунок 3).

Рисунок 3 – График зависимости времени работы различных алгоритмов поиска от размерности массива

Временная сложность алгоритмов:

- алгоритм линейного поиска - O(n);

- алгоритм линейного поиска в упорядоченном массиве - O(n);

- алгоритм линейного поиска с барьером- O(n);

- алгоритм бинарного поиска в упорядоченном массиве - O(log(2n));

- алгоритм блочного поиска в упорядоченном массиве - O(n).

Также тут стоит отметить то, что эта сложность приведена для наихудшего варианта.

Алгоритм Бойера-Мура исследовался отдельно от остальных. Практические данные для исходного числа массива (не сортированного), для отсортированных данных по убыванию и для отсортированных данных по возрастанию приведены в таблице 4. В худшем случае временная сложность алгоритма – O(n\*m), где n – размерность шаблона, а m – размерность строки в которой ищется совпадение шаблона.

Таблица 4. Практические результаты времени работы алгоритма Бойера-Мура

|  |  |  |  |
| --- | --- | --- | --- |
| *Размер массива* | *Время поиска в исходном массиве (мс)* | *Время поиска в отсортированном массиве по возрастанию (мс)* | *Время поиска в отсортированном массиве по убыванию (мс)* |
| 0 | 0 | 0 | 0 |
| 1000 | 2,231162 | 0,701087 | 0,130664 |
| 2000 | 2,507491 | 1,155057 | 0,197378 |
| 3000 | 1,863644 | 0,523841 | 0,545159 |
| 4000 | 1,924042 | 0,5345 | 1,426649 |
| 5000 | 1,369804 | 0,35607 | 0,211984 |
| 6000 | 2,782637 | 1,914173 | 0,166982 |
| 7000 | 2,379985 | 0,969127 | 0,346202 |
| 8000 | 2,915275 | 1,329144 | 0,101847 |
| 9000 | 3,651495 | 1,794956 | 0,045397 |
| 10000 | 2,156948 | 0,630425 | 0,791487 |
| 20000 | 4,537723 | 3,107126 | 0,285804 |
| 30000 | 6,971 | 3,056598 | 0,253433 |
| 40000 | 6,515846 | 2,646446 | 0,6474 |
| 50000 | 6,970605 | 4,436665 | 1,627975 |
| 60000 | 7,659849 | 2,800795 | 1,275063 |

Рисунок 4. График зависимости времени выполнения поиска алгоритмом Бойера-Мура от размерности исходных данных

Пару слов можно сказать по поводу графика. Из него отчетливо видно, что алгоритм работает быстрее для отсортированных данных по убыванию, однако это не так. Связано это со случайным генератором, который помещает случайное значение в массив случайных данных. В приложении можно найти каким образом происходит генерация данных предметной области. Тут же можно предположить, что сгенерированные датчиком случайных чисел данные не удовлетворяют нормальному распределению и это нужно исследовать, однако это не является темой данной работы. Интересующийся читатель может самостоятельно проверить данное предположение.

3 ИССЛЕДОАВНИЕ АЛГОРИТМОВ СОРТИРОВОК ДЛЯ РАЗЛИЧНЫХ СТРУКТУР И РАЗМЕРНОСТЕЙ ДАННЫХ

3.1 Общие теоретические положения и постановка задачи

Сортировка данных является основой для любого алгоритма. Практически ни одна программа не обходится без данного алгоритма, правда, не всегда видно каким образом это делается. Человек лучше воспринимает информацию ранжированную или отсортированную по какому-либо критерию, поэтому данный вид алгоритма очень важен. Однако их существует большое количество и каждый из них имеет свои преимущества и недостатки. Поэтому разработчику необходимо знать, как, и где лучше применять тот или иной алгоритм.

В данном разделе необходимо провести экспериментальный сравнительный анализ различных методов сортировки. Для чистоты эксперимента сортировка должна проводиться на одинаковых наборах входных данных, которые генерируются случайным образом. Для более полного анализа методов сортировка должна проводиться для различных размерностей данных, например, 500, 1000, 3000, 5000, 8000, 10000, 30000, 60000.

В качестве исходных наборов данных использовать массивы или файлы соответствующего типа.

Необходимо проследить динамику роста требуемого для сортировки времени.

Провести эксперименты для различных типов данных, упорядоченных в случайном порядке, по возрастанию или убыванию.

Необходимо сравнить теоретические оценки времени сортировки и числа требуемых операций с экспериментальными.

Построить соответствующие таблицы и графики сравнительного анализа различных методов сортировки.

Провести исследования для метода Шелла с собственным алгоритмом изменения числа серий и сортировки Хоара с выбором медианы на каждом шаге разделения.

3.2 Описания и алгоритмы

3.2.1 Алгоритм сортировки вставками

Идея данного алгоритма заключается в разделении массива на отсортированную и не отсортированную части. Из неотсортированной части выбирается последовательно текущий элемент и перемещается в отсортированную часть на то место, пока не нарушится условие сравнения: следующий элемент отсортированной части больше (меньше) перемещаемого элемента в случае сортировки по возрастанию (убыванию);

Как только текущему элементу было найдено место в отсортированной части массива, берется следующий элемент из неотсортированной части и действие алгоритма повторяется. Визуализация алгоритма проиллюстрирована на рисунке 5.

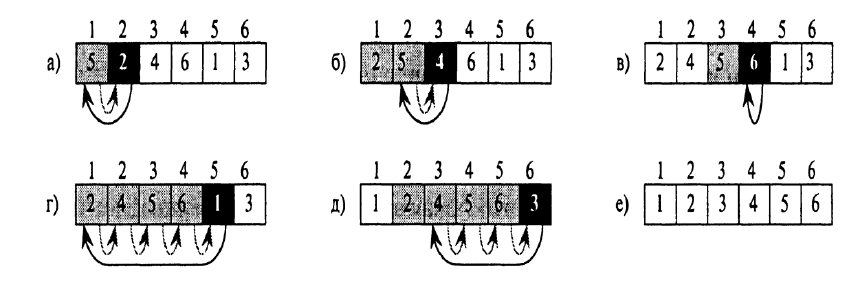


Рисунок 5 – Иллюстрация алгоритма сортировки вставками

3.2.3 Алгоритм сортировки Шелла

Алгоритм сортировки Шелла можно считать модификацией алгоритма вставками, т.к. сортируются данные в алгоритме Шелла точно также, как и в алгоритме вставками, однако, в алгоритме Шелла применена идея принципа «Разделяй и властвуй», где исходный массив разбивается на серии, и для каждой серии выполняется сортировка вставками. После этого алгоритм снова разбивается на серии, но уже на меньшее их количество и так до тех пор, пока количество серий не останется равным одному.

Данный алгоритм требует предварительного вычисления серий. Существуют разные подходы для вычисления, однако, самым оптимальным считаются серии, предложенные Седжвиком.

(1)

В данной работе алгоритм Шелла с вычислением серий, предложенных Седжвиком будет сравниваться с алгоритмом Шелла с вычислением серий по следующему закону:

(2)

3.2.2 Алгоритм сортировки Хоара

Данный вид сортировки (называемый еще Quick sort или быстрая сортировка) считается одним из самых быстрых алгоритмов сортировки. Он является улучшением алгоритма пузырьковой сортировки. Основной принцип данного алгоритма заключается в разделении после каждого прохода на группы и в перестановках на наиболее возможном расстоянии. Данный алгоритм также использует принцип «Разделяй и властвуй».

Сам алгоритм можно описать следующими шагами:

1. выбор одного элемента массива в качестве опорного. Есть различные методики выбора, начиная от выбора срединного элемента до выбора элемента со случайным индексом. Хорошие данные показывается алгоритм с выбором среднеарифметического значения между максимальным и минимальным значениями массива в качестве опорного.

2. Производится разделение массива таким образом, чтобы слева от опорного элемента остались элементы меньше (или равны) опорному элементу, а справа, соответственно больше.

- производится установка индекса i на минимальный индекс массива;

- производится смещение индекса по элементам вправо до тех пор, пока значение элемента с индексом i не будет больше значения опорного элемента;

- производится установка индекса j на максимальный индекс массива;

- производится смещение индекса j по элементам влево до тех пор, пока значение элемента с индексом j не будет меньше (или равно) значения опорного элемента;

- значения элементов с индексами i и j меняются местами;

- если i=j то операция разделения закончена.

3. Рекурсивно упорядочиваются правая и левая части, находящиеся соответственно справа и слева от опорного массива.

Иллюстрация работы алгоритма представлена на рисунке 6.

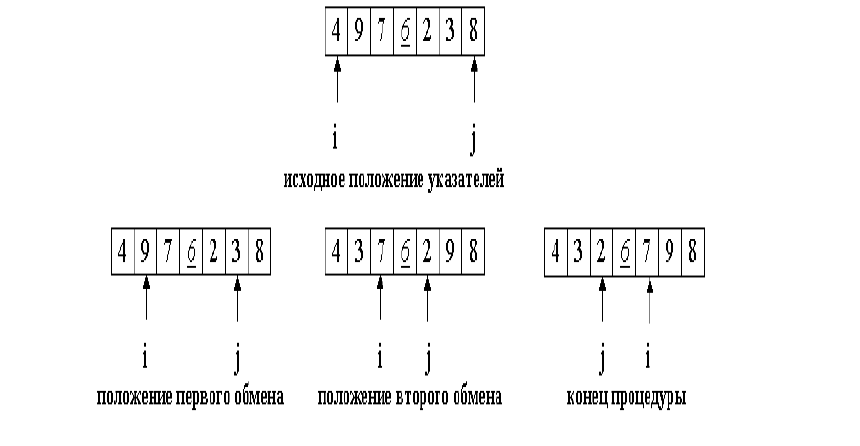


Рисунок 6 – Иллюстрация алгоритма быстрой сортировки

3.3 Результат выполнения и сравнительный анализ

В связи с тем, что сгенерировать последовательность данных для сортировки в 60 тысяч элементов из словаря предметной области в 30 слов не представляется возможным (повторяемость элементов сильно повлияет на алгоритмы сортировки) принято решение о генерации дробных значений и их сортировке. Пример выполнения программы представлен на рисунке 7.

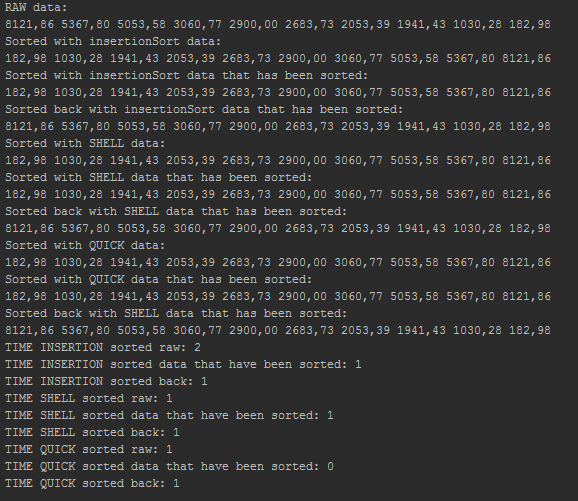


Рисунок 7 – Пример выполнения программы

Произведены практические замеры времени сортировок Шелла, вставками и быстрой сортировки. Результаты все отражены в таблице 5. На основе этих данных построены графики для сравнительного анализа относительной скорости выполнения сортировки (Рисунок 8, 9, 10)

Таблица 5. Практические результаты времени работы алгоритмов сортировки.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| *Размерность данных* | *Сортировка вставками не упорядоченные данные* | *Сортировка вставками, упорядоченные данные* | *Сортировка вставками упорядоченные в обратном направлении данные* | *Сортировка Шелла не упорядоченные данные* | *Сортировка Шелла упорядоченные данные* | *Сортировка Шелла упорядоченные данные в обратном направлении* | *Быстрая сортировка не упорядоченные данные* | *Быстрая сортировка упорядоченные данные* | *Быстрая сортировка упорядоченные в обратном направлении данные* |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1000 | 6 | 0 | 4 | 6 | 0 | 5 | 0 | 0 | 0 |
| 2000 | 17 | 0 | 15 | 17 | 1 | 17 | 0 | 0 | 0 |
| 3000 | 38 | 0 | 33 | 38 | 2 | 36 | 0 | 0 | 0 |
| 4000 | 63 | 0 | 58 | 68 | 2 | 64 | 0 | 0 | 1 |
| 5000 | 101 | 0 | 90 | 115 | 2 | 98 | 1 | 0 | 1 |
| 6000 | 141 | 1 | 133 | 149 | 2 | 143 | 1 | 0 | 1 |
| 7000 | 193 | 1 | 180 | 205 | 1 | 191 | 1 | 1 | 0 |
| 8000 | 253 | 1 | 230 | 275 | 2 | 252 | 1 | 0 | 1 |
| 9000 | 321 | 0 | 295 | 339 | 2 | 315 | 1 | 1 | 0 |
| 10000 | 393 | 1 | 362 | 418 | 2 | 389 | 1 | 1 | 1 |
| 20000 | 1691 | 1 | 1580 | 1792 | 4 | 1678 | 2 | 2 | 3 |
| 30000 | 4001 | 2 | 3751 | 4234 | 4 | 3978 | 3 | 3 | 4 |
| 40000 | 7311 | 1 | 6907 | 9658 | 5 | 7248 | 4 | 5 | 5 |
| 50000 | 11798 | 2 | 10904 | 12168 | 7 | 11434 | 5 | 6 | 6 |
| 60000 | 16810 | 2 | 15896 | 18075 | 8 | 16655 | 7 | 7 | 7 |

Анализируя графики, полученные на основе данных таблицы 5 можно прийти к выводу, что самым эффективным алгоритмом является алгоритм быстрой сортировки время работы, которого не превышает 7 миллисекунд для данных размерностью 60 тысяч элементов (Таблица 5).

Также стоит сказать пару слов о сортировке Шелла и сортировке вставками. На графиках видно, что сортировка вставками происходит быстрее сортировки Шелла, однако на самом деле это не совсем так. Замер времени проводился совместно с вычислением серий (серии вычислялись по формуле Седжвика) для каждого замера производится вычисление серий, что увеличивает время работы алгоритма. Однако, при разработке реального программного обеспечения последовательность серий вычисляется единожды при запуске программы и затем она используется на протяжении всего времени ее работы.

Рисунок 8 – Время сортировки неупорядоченных данных для различных размерностей данных

Рисунок 9 – Время сортировки упорядоченных данных для различных размерностей данных

Рисунок 10 – Время сортировки упорядоченных в обратном порядке данных для различных размерностей данных

Таблица 6. Практические результаты времени работы алгоритма Шелла для серий, вычисленных по Седжвику и собственного алгоритма серий.

| *Размерность данных* | *Серии по Седжвику для не сортированных данных* | *Серии по Седжвику для сортированных данных* | *Серии по Седжвику для сортированных в обратном порядке данных* | *Собственный алгоритм сесий для не сортированных данных* | *Собственный алгоритм серий для сортированных данных* | *Собственный алгоритм серий для сортированных данных в обратной последовательности* |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1000 | 5 | 1 | 5 | 3 | 0 | 5 |
| 2000 | 18 | 1 | 17 | 14 | 0 | 15 |
| 3000 | 38 | 2 | 36 | 33 | 0 | 38 |
| 4000 | 67 | 1 | 63 | 58 | 0 | 63 |
| 5000 | 104 | 2 | 99 | 92 | 0 | 97 |
| 6000 | 151 | 2 | 140 | 132 | 0 | 141 |
| 7000 | 205 | 2 | 191 | 180 | 1 | 192 |
| 8000 | 269 | 2 | 250 | 236 | 0 | 251 |
| 9000 | 338 | 2 | 318 | 301 | 0 | 317 |
| 10000 | 418 | 3 | 391 | 371 | 0 | 390 |
| 20000 | 1788 | 3 | 1687 | 1622 | 0 | 1688 |
| 30000 | 4218 | 5 | 4016 | 3835 | 1 | 4018 |
| 40000 | 7733 | 5 | 7241 | 6944 | 1 | 7199 |
| 50000 | 12417 | 7 | 11607 | 11031 | 2 | 11414 |
| 60000 | 17682 | 8 | 17101 | 16087 | 1 | 16514 |

В таблице 6 представлены практические результаты работы алгоритма Шелла для разной упорядоченности исходных данных. Серии в алгоритме вычислялись с помощью формулы Седжвика (1) и с помощью собственной формулы (2). На основании данных таблицы построены графики функций (Рисунок 11, 12, 13).

Рисунок 11 - Время сортировки не упорядоченных для последовательности серий, вычисленных различными способами

Рисунок 12 - Время сортировки упорядоченных данных для последовательности серий, вычисленных различными способами

Рисунок 13 - Время сортировки упорядоченных в обратном направлении данных для последовательности серий, вычисленных различными способами

Анализируя полученные данные, и графики можно сделать вывод, что алгоритм Шелла использующий вычисление последовательности по формуле Седжвика (1) работает медленнее алгоритма, использующего вычисление серий по формуле (2). В связи с полученными данными было проведено еще одно сравнение алгоритмов сортировки (алгоритма Шелла с использованием формулы вычисления последовательности серий (2) и алгоритма сортировки вставками). Полученные данные в ходе эксперимента объединены в таблицу 7.

Таблица 7. Практические результаты времени работы алгоритма Шелла для собственного алгоритма серий и алгоритма сортировки вставками.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| *Размерность данных* | *Не сортированные данные, алгоритм сортировки вставками* | *Сортированные данные, алгоритм сортировки вставками* | *Сортированные в обратном порядке данные, алгоритм сортировки вставками* | *Не сортированные данные, алгоритм Шелла на собственном алгоритме серий* | *Сортированные данные, алгоритм Шелла на собственном алгоритме серий* | *Сортированные в обратной последовательности данные, алгоритм Шелла на собственном алгоритме серий* |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1000 | 6 | 0 | 4 | 6 | 0 | 6 |
| 2000 | 16 | 0 | 15 | 18 | 0 | 17 |
| 3000 | 35 | 1 | 33 | 38 | 0 | 36 |
| 4000 | 62 | 0 | 59 | 68 | 1 | 64 |
| 5000 | 99 | 0 | 91 | 105 | 1 | 98 |
| 6000 | 144 | 0 | 131 | 149 | 1 | 140 |
| 7000 | 192 | 1 | 177 | 204 | 2 | 191 |
| 8000 | 252 | 0 | 231 | 267 | 2 | 250 |
| 9000 | 325 | 0 | 293 | 339 | 2 | 316 |
| 10000 | 399 | 0 | 368 | 416 | 1 | 391 |
| 20000 | 1708 | 1 | 1580 | 1803 | 2 | 1688 |
| 30000 | 4007 | 2 | 3798 | 4265 | 2 | 4047 |
| 40000 | 7546 | 2 | 6818 | 7741 | 2 | 7275 |
| 50000 | 11550 | 2 | 11056 | 12049 | 3 | 11399 |
| 60000 | 16818 | 2 | 15810 | 17725 | 3 | 16787 |

Как видно из таблицы алгоритм Шелла с вычислением последовательности серий по собственной формуле (2) выполняется также чуть медленнее алгоритма сортировки вставками, и связано это также с расходованием времени на вычисление последовательности. Точно сказать какой алгоритм лучше сказать сложно из-за не идеальных условий проведения экспериментов, а также большое влияние на работу оказывает количество использования в программе алгоритма.

4 РЕАЛИЗАЦИЯ СТРУКТУРЫ ДАННЫХ ТИПА ДЕРЕВО И ТИПОВЫЕ АЛГОРИТМЫ ИХ ОБРАБОТКИ

4.1 Общие теоретические положения и постановка задачи

Сильноветвящееся дерево – дерево, у которого большое (больше двух) количество ветвей, исходящих из одного узла (Рисунок 14). Такие деревья также называются Б-деревьями. Как и любое другое дерево Б-дерево имеет один общий узел, который называется корнем (root). Данного типа деревья является полностью сбалансированными.

Каждое дерево определяется минимальной степенью дерева *T*≥ 2, которая определяет минимальное и максимальное количество элементов, которые допустимо хранить в узле. Минимальное количество элементов не может быть менее чем *T*- 1, а максимальное количество элементов не должно превышать 2·*T*- 1. Например, для *T* = 2 в узле допустимо нахождения элементов в диапазоне [ 1 ; 3 ]. У каждого узла может быть либо m+1 потомков (m – количество элементов в узле), либо потомков может не быть. Узлы, не имеющие потомков, называются листьями. Все листья расположены на одном уровне.

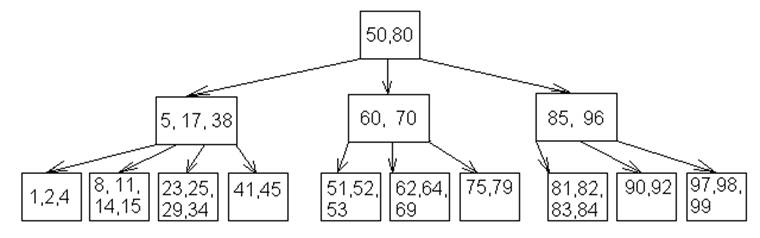


Рисунок 14 – Сильноветвящееся дерево

Б-деревья имеют большое практическое значение. Большое отличие данного дерева от других является то, что каждая ветвь – это адрес во внешней памяти или на диске. Т.к. данный тип памяти является медленным, то необходимо сократить количество операций обращения к этой памяти. За счет хранения большого количества элементов в узле (странице жесткого диска) при одном обращении к диску или внешней памяти считывается сразу большое количество информации

В данном разделе будут описаны основные операции, применяемые к структурам данных, а именно: поиск элемента в дереве, добавление нового элемента, удаление элемента из дерева.

Структура каждого узла может быть описана следующим образом:

class BTreeNode<Type> {  
 public boolean Leaf; //Is the node leaf  
 public int Size; //Values count  
 public Comparable<Type>[] Values; //Values (keys)  
 public BTreeNode[] Children; //Pointers  
}

4.2 Описания и алгоритмы

4.2.1 Алгоритм добавления элемента в дерево

Для вставки значения в дерево необходимо пройти дерево от корня к необходимому узлу. Если узел, куда вставляется новое значение не является полным, то производится вставка в этот узел в соответствующую позицию, а остальные элементы смещаются вправо от вставляемого значения.

Если узел, куда помещается вставляемое значение заполнен, то производится разбиение этого узла (Рисунок 15).

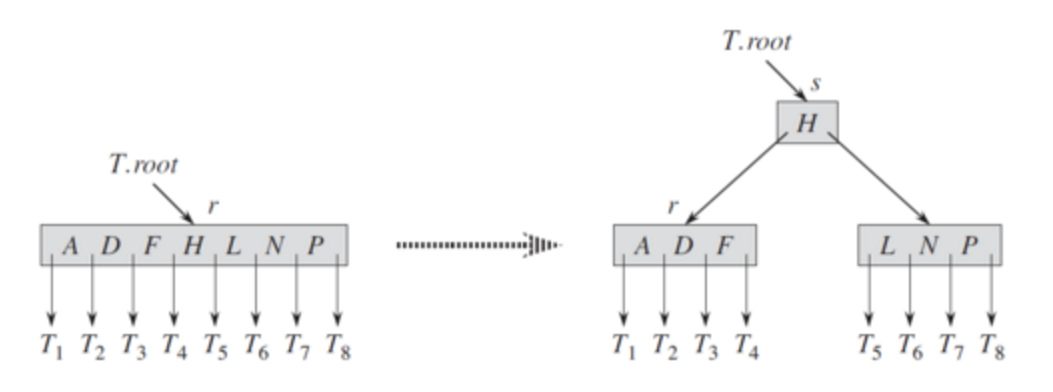


Рисунок 15 – Процесс разбиения узла b-дерева

При разбиении центральный элемент выносится в качестве нового узла, левые элементы помещаются в левое поддерево, а правые – в правое. Также во время разбиения перемещаются соответствующие ссылки на поддеревья.

Если после разбиения корневой элемент оказался полным, то продолжается разбиение для следующего узла и так до тех пор, пока не будет найден не заполненный узел, либо пока не встретится корень. Тогда в этом случае увеличивается высота дерева. Стоит заметить, что операция вставки выполняется рекурсивно.

4.2.2 Алгоритм удаления элемента из дерева

Операция удаления на порядок сложнее операции вставки.

Если удаляемое значение находится в узле, который является листом, тогда производится обычное удаление из этого узла (Рисунок 17).

Если удаляемое значение находится в узле, не являющемся листом, выполняются следующие действия:

- Если у следующего или предыдущего потомка количество элементов составляет больше *T* - 1 то крайний правый дочерний элемент (левый потомок) ставится вместо удаляемого элемента и удаление этого (крайнего правого элемента у левого потомка) элемента продолжается рекурсивно в левом поддереве, аналогично для правого потомка, т.е. крайний левый элемент дочернего узла, находящегося справа от удаления помещается вместо удаляемого элемента и продолжается удаление крайнего левого элемента дочернего узла, находящегося справа (Рисунок 18).

- Если в правом и левом потомке количество элементов *T*– 1, то происходит объединение потомков в один узел и элемент из родительского узла удаляется (Рисунок 19).

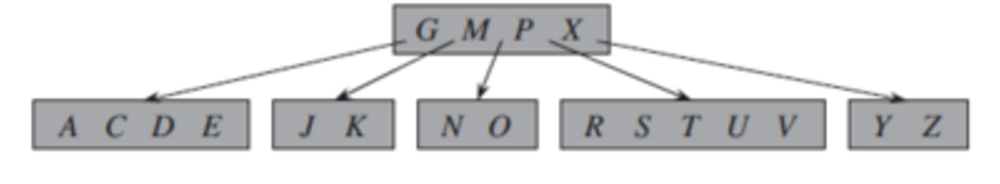


Рисунок 16 – Начальное состояние дерева

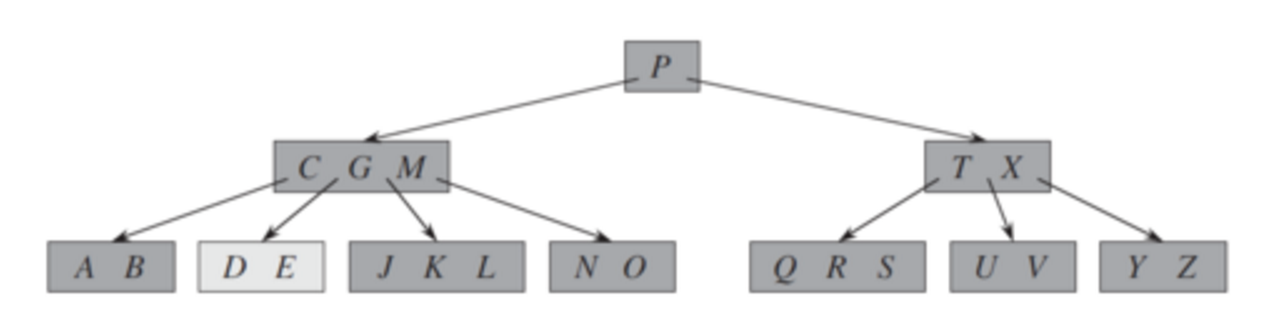


Рисунок 17 – Удаление элемента из листа дерева

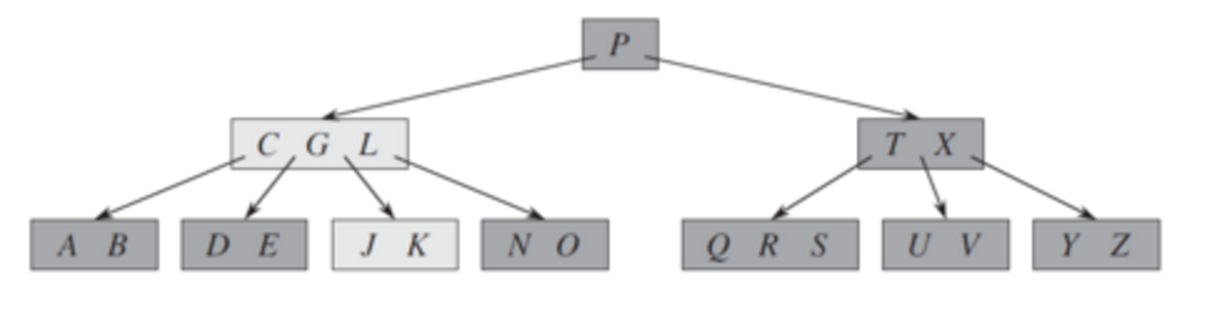


Рисунок 18 – удаление элемента «M» из узла без объединения

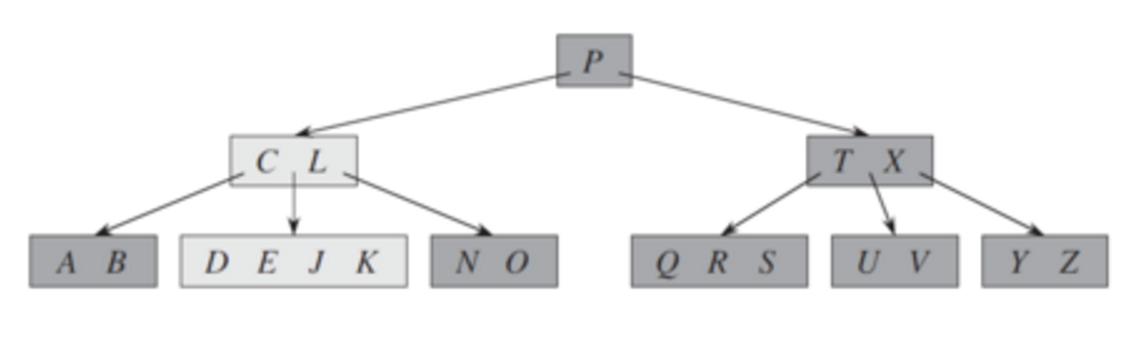


Рисунок 19 – Удаление элемента из узла с объединением

4.2.3 Алгоритм поиска элемента в дереве

Процедура поиска элемента в дереве является также рекурсивной процедурой. Сначала необходимо проверить все элементы родительского узла, если значения совпадают, то возвращается поддерево.

Если в родительском узле не содержится элемент, то определяется дальнейшая ветка для следования вниз по дереву и рекурсивно запускается поиск в поддереве.

Если родительский элемент является одновременно листом, и не содержит искомого элемента, то в дереве нет искомого значения.

4.3 Результат выполнения и сравнительный анализ

Построение дерева производится в одинаковой последовательности. Последовательность добавления элементов в дерево:

1; 3; -2; 8; 4;

6; -8; 12; -4; -3;

2; 3; -1; 4; 1;

5; 10; 3; -1; 0;

-4; -6; -8; -2; 3;

4; 9; 11; 12; 5.

Построение деревьев производится с рангом 3, 4, 10. Удаление элементов также производится в одинаковой последовательности для всех деревьев:

8; 1; 3; -2; -4;

-8; 4; 10; 11; 12;

5; -1; 0; 2.

Результаты выполнения для деревьев с рангом 3 (Рисунок 20), рангом 4 (Рисунок 21) и ранком 5 (Рисунок 22) представлены на рисунках ниже.

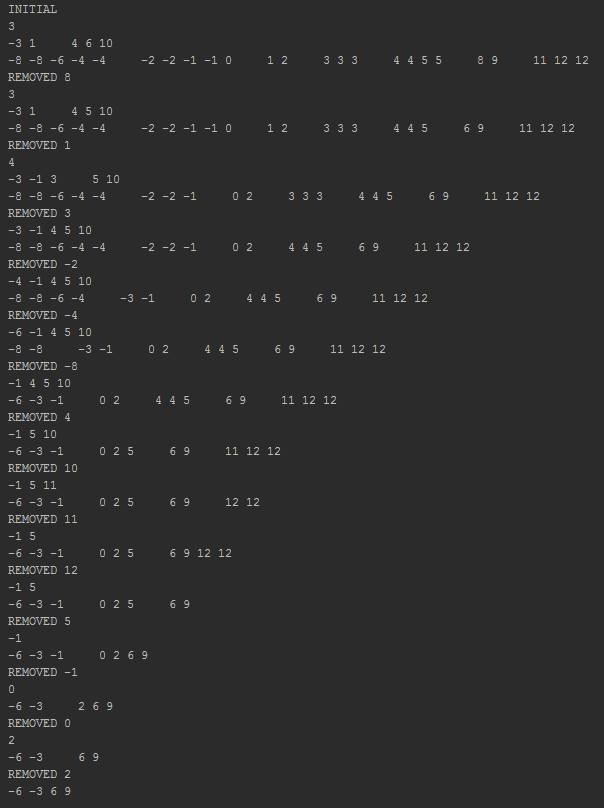


Рисунок 20 – Удаление элементов из дерева с рангом 3

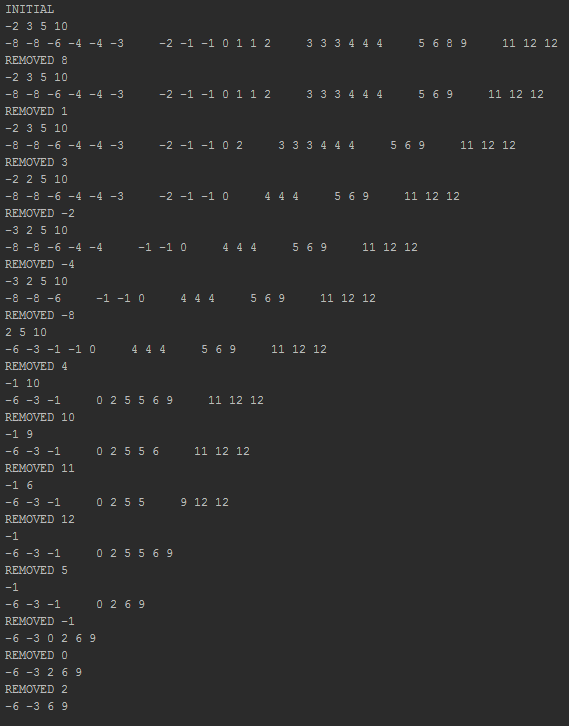


Рисунок 21 – Последовательность удаления из дерева с рангом 4

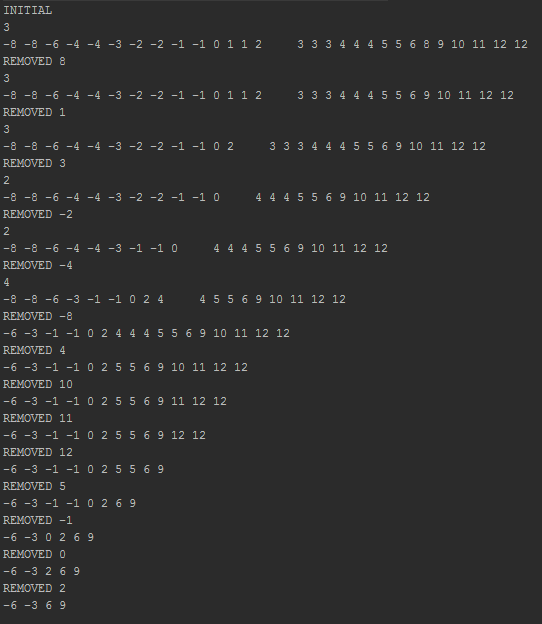


Рисунок 22 – Последовательность удаления из дерева с рангом 10

Результаты построения деревьев также отражены на рисунках 20, 21, 22 в разделе «INITIAL»

Можно убедиться, что все деревья после перестроения остаются сильноветвящимися и удовлетворяют всем свойствам, перечисленным в разделе 4.1.

Каждый узел на рисунках 20, 21, 22 отделен от других узлов табуляцией, а каждый элемент в узле отделен пробелом.

5 РЕАЛИЗАЦИЯ ФУНКЦИЙ ХЕШИРОВАНИЯ И РАЗЛИЧНЫХ МЕТОДОВ РАЗРЕШЕНИЯ КОЛЛИЗИЙ

5.1 Общие теоретические положения и постановка задачи

Хеширование – процесс преобразования входных данных в выходную последовательность битов фиксированной длины при помощи какого-либо алгоритма – хеш-функции. Хеш-функция должна быть непредсказуемая, т.е. она должна выдавать данные (хеш-код) для близких по значению входных данных совершенно различные и далекие выходные значения хеш-функции, приближенные к случайным.

Во время хеширования могут возникнуть проблемы, называемые коллизиями. Для различных входных данных хеш-функция может выдать одинаковую выходную последовательность. Для решения данной проблемы существует несколько подходов – методы разрешения коллизий:

- линейное опробование;

- квадратичное опробование;

- цепочка;

- двойное хеширование.

Самым распространенным является метод разрешения коллизий «цепочка», представляющий из себя массив элементов, где в качестве каждого элемента хранится динамический двунаправленный список. При возникновении коллизии элемент добавляется в ту же самую ячейку, имеющую одинаковый хеш-код (ключ), в конец динамического массива (Рисунок 23).

Хеш таблица – структура данных, которая представляет собой данные, хранящиеся парами (ключ – значение). Чаще всего такие таблицы называются ассоциативными массивами, т.к. поиск осуществляется по ключу, в качестве которого может быть любой тип данных, в том числе и символьный.

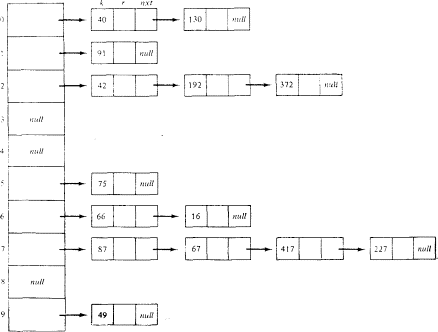


Рисунок 23 – Графическая интерпретация разрешения коллизий методом цепочек

В данном разделе будут рассмотрены 3 различных функции хеширования, заполнены хеш-таблицы. Реализованы функции добавления нового элемента в таблицу, удаление элемента из таблицы и поиск элемента в таблице. Также будет реализован метод разрешения коллизий в виде цепочки.

Будет произведена оценка качества каждой функции хеширования.

5.2 Описания и алгоритмы

Для добавления нового элемента в хеш-таблицу вычисляется хеш-код (ключ) от входных данных. Если в хеш-таблице значение с данным ключом является NULL, то создается новый объект типа «список» и в этот список добавляется единственный элемент.

Если в таблице с ключом уже существуют данные, то перебирается весь список элементов с текущим значением ключа. Если список не содержит добавляемое значение, то оно добавляется в конец списка и увеличивается число коллизий. Иначе (если добавляемый элемент существует в хеш-таблице), функция завершает работу без каких-либо изменений.

Процесс удаления значения из таблицы схож с процессом добавления. Сначала вычисляется хеш-код (ключ) от входных данных.

Перебираются все элементы списка с ключом, вычисленным в предыдущем пункте. Если элемент существует в списке – он удаляется.

Если после удаления список стал пустым, то значение в таблице по ключу записывается NULL.

Процесс поиска элемента в хеш-таблице также не представляет особых трудностей. Перед поиском также вычисляется хеш-код (ключ) от искомого значения. Затем по ключу просматривается список, и, если элемент существует в списке, функция возвращает ключевое значение.

Первые две хеш-функции принимают в качестве параметра объект, вычисляют на основе его строковое значение, на основе этого значения вычисляется число, которое используется дальше в формулах (3, 4). Третья функция использует встроенный в Java метод hashCode(), который возвращает целочисленное значение на основе объекта.

В первой хеш-функции используется следующее выражение:

где *n* – число, полученное из строкового значения;

*Simple*1, Simple2 – простые случайные числа, т.е. числа делящиеся на единицу и сами на себя;

MOD – операция остатка от деления левой части выражения на правую.

где *n* – число, полученное из строкового значения;

*Simple –* простое случайное число, т.е. число делящееся на единицу и на само себя.

5.3 Результат выполнения и сравнительный анализ

Сгенерирована случайная последовательность данных из тысячи элементов. Созданы три таблицы для каждой хеш-функции размерностью также 1000 элементов. Проведя несколько испытаний построен график зависимости (Рисунок 25) количества коллизий от метода хеширования. Результат выполнения программы приведен на рисунке 24.

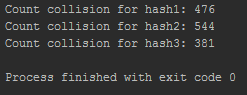


Рисунок 24 – Результат выполнения программы

Рисунок 25 – График зависимости количества коллизий от метода хеширования

Из графика (Рисунок 25) видно, что наилучшие результаты показала хеш-функция 3, с минимальным количеством коллизий, также выдаваемый результат является стабильным. Хеш-функция 2 показала также неплохой стабильный и средний результат. Третья функция выдает нестабильное количество коллизий, однако распределение коллизий равномерное (Рисунок 26-28).

Рисунок 26 – Распределение коллизий по индексам для хеш-функции 1

Рисунок 27– Распределение коллизий по индексам для хеш-функции 2

Рисунок 28– Распределение коллизий по индексам для хеш-функции 3

На основании гистограмм (Рисунок 26-28) можно сделать вывод, что все три хеш-функции являются хорошими функциями, т.е. распределение по хеш-таблицам производится равномерно.

Также из графиков видно, что для первой и второй функций максимальное количество коллизий для одного ключа составляет 8, а для третьей функции 5.

На основе предыдущего утверждения можно сделать вывод, что хеш-таблица для третьей хеш-функции будет иметь меньший размер в оперативной памяти, т.к. хранится меньшее количество ссылок в списках.

Наиболее эффективной для использования является третья хеш-функция, т.е. встроенная в Java функция hashCode(), возвращающая хеш-код для каждого объекта.

Заключение

В ходе проделанной курсовой работы были изучены многие алгоритмы и структуры данных, реализованы структуры типа дерево, абстрактные типы данных типа списки, очереди, стеки и другие.

Произведено моделирование вычислительной системы с дискретным генератором задач, рассмотрены различные функции хеширования и многое другое.

В результате выполнения работы произведен анализ эффективности различных алгоритмов, сравнение скоростей их выполнения для различных типов данных и различных размерностей, сделаны выводы об эффективности использования определенных алгоритмов в определенных ситуациях. Так, например, сортировку Шелла целесообразно использовать при многократных сортировках, а сортировку вставками лучше использовать при единовременной сортировке данных.

Одной из наиболее сложных задач оказалась отладка вычислительной системы, т.к. ошибки появлялись случайным образом и закономерность их появления выявить было невозможным. Связано это было с постоянной генерацией случайных новых задач, и удалением старых, что приводило практически к не контролируемому изменению структур данных, отследить изменения которых было сложно.

Другой сложностью оказалась реализация структуры сильноветвящегося дерева (в особенности операция удаления элемента из дерева). Подробного описания алгоритма в сети Интернет нет и отладка такой структуры также сложна из-за большой вложенности элементов и постоянных рекурсивных вызовов.

Список используемой литературы

1. Алгоритм Бойера-Мура // Викиконспекты [Электронный ресурс]. – 2016. – Режим доступа: https://neerc.ifmo.ru/wiki/index.php?title=Алгоритм\_Бойера-Мура. – Дата доступа: 15.11.2016.

2. Алгоритм Бойера-Мура // Википедия [Электронный ресурс]. – 2016. – Режим доступа: https://ru.wikipedia.org/wiki/Алгоритм\_Бойера\_—\_Мура. – Дата доступа: 18.11.2016

3. Руководство по языку программирования Java // Metaint сайт о программировании [Электронный ресурс]. – 2016. – Режим доступа: http://metanit.com/java/tutorial. – Дата доступа: 21.12.2016.

4. Алгоритмы, методы, исходники // Algolist.manual.ru [Электронный ресурс]. – 2017. – Режим доступа: http://algolist.ru. – Дата доступа: 02.01. 2017.

5. Хеширование // Википедия [Электронный ресурс]. – 2017. – Режим доступа: https://ru.wikipedia.org/wiki/Хеширование. – Дата доступа: 05.01.2017.

6. Б-деревья и их свойства. Включение в Б-дерево. Исключение из Б-дерева // ИВТ-вики [Электронный ресурс]. – 2016. – Режим доступа: http://ивтб.рф/exams/саод/27.htm. – Дата доступа: 21.12.2016.

7. B-дерево // Викиконспекты [Электронный ресурс]. – 2016. –Режим доступа: https://neerc.ifmo.ru/wiki/index.php?title=B-дерево. – Дата доступа: 21.12. 2016.

8. Фофанов О.Б. Алгоритмы и структуры данных – Томск: «Издательство Томского Политехнического Университета», 2014. – 127 с.

Приложение А

Листинг программы для раздела 1

Файл Main

package com.company.Task1;  
  
import java.io.FileWriter;  
import java.util.List;  
import java.util.Random;  
  
public class Main  
{  
 static final int COUNT\_TASKS = 10000;  
 static final int MAX\_TASK\_TIME = 2;  
 static int processID = 0;  
 //Create 3 queues  
 static MyQueue<MyProcess> myLowPriorityQueue;  
 static MyQueue<MyProcess> myMiddlePriorityQueue;  
 static MyQueue<MyProcess> myHighPriorityQueue;  
 //Create new processor  
 static MyProcessor processor;  
  
 ////////////////  
 //main function  
 ////////////////  
 public static void main(String[] args) throws Exception  
 {  
 //Create 3 queues  
 myLowPriorityQueue = new MyQueue();  
 myMiddlePriorityQueue = new MyQueue();  
 myHighPriorityQueue = new MyQueue();  
 //Create new processor  
 processor = new MyProcessor();  
 //Create output file  
 FileWriter file = new FileWriter("D:\\result.csv");  
 int currentCycle = 0; //Current cycle iteration  
 while(true) { //endless cycle  
 //Save data to the file  
 file.write(generateOutputFileString(currentCycle++));  
 Thread.sleep(1); //Sleep 1 ms  
 MyProcess process; //Create new process/task  
 if( (process = generateProcess()) == null) {  
 completeTask();  
 continue;  
 }  
 //Complete all tasks before exit  
 if(processID >= COUNT\_TASKS) //close after 10000 tasks  
 {  
 if(myHighPriorityQueue.isEmpty() && myMiddlePriorityQueue.isEmpty() && myLowPriorityQueue.isEmpty()) {  
 file.close();  
 break;  
 }  
 else  
 completeTask();  
 continue;  
 }  
 //Add the process to a queue  
 switch (process.Priority) {  
 case HIGHEST: {  
 myHighPriorityQueue.enqueue(process);  
 System.out.println(process.Name + " added to HIGH priority queue");  
 break;  
 }  
 case MIDDLE: {  
 myMiddlePriorityQueue.enqueue(process);  
 System.out.println(process.Name + " added to MIDDLE priority queue");  
 break;  
 }  
 case LOWEST: {  
 myLowPriorityQueue.enqueue(process);  
 System.out.println(process.Name + " added to LOW priority queue");  
 break;  
 }  
 }  
 completeTask();  
 }  
 }  
 ///////////////////  
 //Process generator  
 ///////////////////  
 public static MyProcess generateProcess()  
 {  
 Random random = new Random(System.currentTimeMillis());  
 int time = random.nextInt(MAX\_TASK\_TIME+1);  
 if(time == 0) //there's no task  
 return null;  
 processID ++;  
 MyProcess process = new MyProcess();  
 //Generate parameters  
 process.Name = "PROCESS\_"+ processID;  
 process.Time = time;  
 //Generate priority  
 int temp = random.nextInt(3); //random number 0..2  
 switch (temp)  
 {  
 case 0:  
 process.Priority = Priorities.HIGHEST;  
 break;  
 case 1:  
 process.Priority = Priorities.MIDDLE;  
 break;  
 case 2:  
 process.Priority = Priorities.LOWEST;  
 break;  
 }  
 return process;  
 }  
 ///////////////////////////  
 //Execute task on processor  
 ///////////////////////////  
 public static void completeTask()  
 {  
 //Checking new task for breaking the current task in a processor  
 if(!myHighPriorityQueue.isEmpty()) {  
 try {  
 //Can we put task in processor  
 if (processor.interrupt(myHighPriorityQueue.front())) {  
 myHighPriorityQueue.dequeue(); //remove task from queue  
 }  
 }  
 catch (Exception e) {  
 System.out.println(e.getMessage());  
 }  
 }  
 else if(!myMiddlePriorityQueue.isEmpty()) {  
 try {  
 //Can we put task in processor  
 if (processor.interrupt(myMiddlePriorityQueue.front())) {  
 myMiddlePriorityQueue.dequeue(); //remove task from queue  
 }  
 }  
 catch (Exception e) {  
 System.out.println(e.getMessage());  
 }  
 }  
 else if(!myLowPriorityQueue.isEmpty()) {  
 try {  
 //Can we put task in processor  
 if (processor.interrupt(myLowPriorityQueue.front())) {  
 myLowPriorityQueue.dequeue(); //remove task from queue  
 }  
 }  
 catch(Exception e) {  
 System.out.println(e.getMessage());  
 }  
 }  
 else  
 System.out.println("PROCESSOR IS WAITING NEW TASK");  
 }  
  
 //////////////////////////////////////  
 //Generate new string to write to file  
 //////////////////////////////////////  
 public static String generateOutputFileString(int time) throws Exception  
 {  
 String outputString = "";  
 List<MyProcess> list;  
 /\*Block highQueue\*/  
 outputString = time + "; ";  
 outputString += "HighPriorityQueue; ";  
 outputString += myHighPriorityQueue.size() + "; ";  
 if(!myHighPriorityQueue.isEmpty()) {  
 list = myHighPriorityQueue.getQueueElements();  
 //Get all elements from queue  
 for (MyProcess current : list)  
 outputString += current.Name + "," + current.Time + ";";  
 }  
 outputString += "\n";  
  
 /\*Block middleQueue\*/  
 outputString += time + "; ";  
 outputString += "MiddlePriorityQueue; ";  
 outputString += myMiddlePriorityQueue.size() + "; ";  
 if(!myMiddlePriorityQueue.isEmpty()) {  
 list = myMiddlePriorityQueue.getQueueElements();  
 //Get all elements from queue  
 for (MyProcess current : list)  
 outputString += current.Name + "," + current.Time + ";";  
 }  
 outputString += "\n";  
  
 /\*Block lowQueue\*/  
 outputString += time + "; ";  
 outputString += "LowPriorityQueue; ";  
 outputString += myLowPriorityQueue.size() + "; ";  
 if(!myLowPriorityQueue.isEmpty()) {  
 list = myLowPriorityQueue.getQueueElements();  
 //Get all elements from queue  
 for (MyProcess current : list)  
 outputString += current.Name + "," + current.Time + ";";  
 }  
 outputString += "\n";  
  
 /\*Block stack\*/  
 outputString += time + "; ";  
 outputString += "Stack; ";  
 outputString += processor.getStack().count() + "; ";  
 if(!processor.getStack().isEmpty()) {  
 list = processor.getStack().getStackElements();  
 //Get all elements from queue  
 for (MyProcess current : list)  
 outputString += current.Name + "," + current.Time + ";";  
 }  
 outputString += "\n";  
  
 /\*Block processor\*/  
 outputString += time + "; ";  
 outputString += "Processor; ";  
 MyProcess currentTask = processor.getCurrentTask();  
 if(currentTask == null)  
 outputString += 0 + "; ";  
 else  
 {  
 outputString += 1 + "; ";  
 outputString += currentTask.Name +","+ currentTask.Time + ";";  
 }  
 outputString +="\n";  
 return outputString;  
 }  
}

Файл MyNode

package com.company.Task1;  
//Node's structure  
public class MyNode <Type>  
{  
 public MyNode PreviousElement;  
 public Type Data;  
}

Файл MyProcess

package com.company.Task1;  
public class MyProcess  
{  
  
 public String Name;  
 public Integer Time; //Time that need to finish the query  
 public Priorities Priority; //Priority  
  
}

Файл MyProcessor

public class MyProcessor {  
 private boolean IsFree;  
 private MyProcess ProcessInProgress; //Current task in progress  
 private long StartTime; //Start time for current task  
 private MyStack<MyProcess> Stack; //Stack for delayed tasks  
 ///////////////  
 //Constructor  
 //////////////  
 public MyProcessor()  
 {  
 this.IsFree = true;  
 this.Stack = new MyStack<>();  
 }  
 /////////////////////////////  
 //Get stack for delayed tasks  
 /////////////////////////////  
 public MyStack<MyProcess> getStack()  
 {  
 return this.Stack;  
 }  
 ///////////////////////////////////////////  
 //Put new task in processor  
 //return true if task put in the processor  
 ///////////////////////////////////////////  
 private void putTask(MyProcess task) throws Exception  
 {  
 System.out.println("The task " + task.Name + " is in progress now.");  
 this.ProcessInProgress = task;  
 this.StartTime = System.currentTimeMillis();  
 this.IsFree = false;  
 }  
 ///////////////////////////////////////////////  
 //Get current task in a processor. If processor  
 //free, it returns null  
 ///////////////////////////////////////////////  
 public MyProcess getCurrentTask() throws Exception  
 {  
 if(!this.IsFree) //Check if processor free. If not, try to free  
 {  
 long endTime = System.currentTimeMillis();  
 if( (endTime-this.StartTime) > this.ProcessInProgress.Time)  
 {  
 this.StartTime = 0;  
 this.IsFree = true; //Free processor  
 this.ProcessInProgress = null;  
 }  
 }  
 return ProcessInProgress;  
 }  
  
 //////////////////////////////////////////////  
 //Interrupt current task  
 //Return true if interruption process succeed  
 //////////////////////////////////////////////  
 public boolean interrupt(MyProcess applicant) throws Exception  
 {  
  
 if(this.isFree())  
 {  
 this.putTask(applicant);  
 return true;  
 }  
 if(Priorities.hasHighestPriority(applicant.Priority, this.ProcessInProgress.Priority)) //swap tasks  
 {  
 System.out.println("Try to interrupt current task " + this.ProcessInProgress.Name + " with " + applicant.Name);  
 this.Stack.push(ProcessInProgress);  
 System.out.println("Current process " + this.ProcessInProgress.Name + " interrupted");  
 putTask(applicant);  
 return true;  
 }  
 else  
 {  
 System.out.println("Cannot interrupt current task " + this.ProcessInProgress.Name + " with "+ applicant.Name);  
 return false;  
 }  
 }  
 /////////////////////////  
 //Check is processor free  
 /////////////////////////  
 public boolean isFree() throws Exception  
 {  
 if(!this.IsFree) //Check was the task in processor and have it finished  
 {  
 long endTime = System.currentTimeMillis();  
 if( (endTime-this.StartTime) > this.ProcessInProgress.Time)  
 {  
 System.out.println(this.ProcessInProgress.Name + " finished and processor is free.");  
 this.StartTime = 0;  
 this.IsFree = true; //Free processor  
 this.ProcessInProgress = null;  
 if(!this.Stack.isEmpty()) //Try to run task from stack  
 {  
 System.out.println("Try to put " + this.Stack.peek().Name + " task from stack ");  
 putTask(this.Stack.pop());  
 }  
 }  
 }  
 return this.IsFree;  
 }  
}

Файл MyQueue

public class MyQueue <Type>{  
 private int Size;  
 private Type [] Data;  
 private int IndexStart; //The first element to be dequeued (its index)  
 //////////////  
 //Constructor  
 //////////////  
 public MyQueue()  
 {  
 resetQueue();  
 }  
 ////////////////////////////////////  
 //Set all elements in initial state  
 ///////////////////////////////////  
 private void resetQueue()  
 {  
 this.Size = 0;  
 this.Data = (Type[]) new Object[0];  
 this.IndexStart = 0;  
 }  
 //////////////////////////  
 //Add element to the queue  
 //////////////////////////  
 public void enqueue(Type element) throws Exception  
 {  
 if(this.isEmpty())  
 resetQueue();  
 if(this.Data.length == this.Size) //Not enough memory  
 {  
 if(this.IndexStart != 0) //Uncycle queue  
 {  
 Type [] temp = (Type[]) new Object[this.Size\*2]; //Add some memory  
 int indexTemp = 0;  
  
 for(int i=this.IndexStart; i<this.Data.length; i++) //Copy start queue  
 temp[indexTemp++] = this.Data[i];  
 for(int i=0; i<this.IndexStart; i++) //Copy end queue  
 temp[indexTemp++] = this.Data[i];  
 this.Data = temp;  
 this.IndexStart = 0; //Reset index start  
  
  
 }  
 else  
 {  
 this.Data = Arrays.copyOf(this.Data,  
 this.isEmpty() ? 1 : this.Size \* 2); //Just copy uncycled queue  
 }  
 }  
 //Add element to array  
 if( (this.IndexStart + this.Size) >= this.Data.length)  
 this.Data[(this.IndexStart + this.Size) - this.Data.length] = element; //Cycle queue  
 else  
 this.Data[this.IndexStart + this.Size] = element;  
  
 this.Size++;  
  
 }  
 ///////////////////////////////  
 //Delete element from the queue  
 ///////////////////////////////  
 public Type dequeue() throws Exception  
 {  
 if(this.isEmpty()) //Can we dequeue?  
 {  
 resetQueue();  
 throw new Exception("Cannot dequeue. The queue is empty.");  
 }  
 this.Size--;  
 Type returnedValue = this.Data[this.IndexStart];  
 this.Data[this.IndexStart] = null;  
 if( (this.IndexStart + 1) == this.Data.length) //Get next element from cycled queue  
 this.IndexStart = 0;  
 else  
 this.IndexStart++;  
 return returnedValue;  
  
 }  
 /////////////////////  
 ///Return queue's size  
 //////////////////////  
 public int size()  
 {  
 return this.Size;  
 }  
 /////////////////  
 //Is queue empty?  
 /////////////////  
 public boolean isEmpty()  
 {  
 if(this.Size == 0)  
 return true;  
 else  
 return false;  
 }  
 ////////////////////////////////////  
 //Return the first element in queue  
 ////////////////////////////////////  
 public Type front() throws Exception  
 {  
 if(this.isEmpty())  
 {  
 resetQueue();  
 throw new Exception("Cannot get element. Stack is empty");  
 }  
 else  
 return this.Data[this.IndexStart];  
 }  
  
 ////////////////////////////  
 //Get List of queue elements  
 ////////////////////////////  
 public List<Type> getQueueElements() throws Exception {  
 List<Type> list = new ArrayList<>();  
 int start = this.IndexStart;  
 int size = this.Size;  
 //Full list  
 while (size != 0) {  
 size--;  
 list.add(this.Data[start]);  
 //Get next element from cycled queue  
 if ((start + 1) == this.Data.length)  
 start = 0;  
 else  
 start++;  
 }  
 return list;  
 }  
}

Файл MyStack

//class Stack  
public class MyStack <Type> {  
 private MyNode<Type> Top; //Top pointer on TOP  
 private int Count; //Number of stack elements  
  
 /////////////  
 //Constructor  
 /////////////  
 public MyStack()  
 {  
 this.Count = 0;  
 this.Top = null;  
 }  
 //////////////////////  
 //Add element to stack  
 //////////////////////  
 public void push(Type data)  
 {  
 MyNode<Type> element = new MyNode<Type>();  
 if(this.Count == 0)  
 element.PreviousElement = null;  
 else  
 element.PreviousElement = this.Top;  
 element.Data = data;  
 this.Top = element;  
 this.Count++;  
 }  
 ///////////////////////////  
 //Remove element from stack  
 ///////////////////////////  
 public Type pop() throws Exception  
 {  
 if(this.Count == 0)  
 throw new Exception("Can't pop element. Stack is empty");  
 MyNode<Type> element = new MyNode<Type>();  
 element = this.Top;  
 this.Top = this.Top.PreviousElement;  
 this.Count--;  
 return element.Data;  
 }  
 /////////////////////////////////////////  
 //Get element from stack without removing  
 /////////////////////////////////////////  
 public Type peek() throws Exception  
 {  
 if(this.Count == 0)  
 throw new Exception("Can't peak element. Stack is empty");  
 return this.Top.Data;  
 }  
 //////////////////////  
 //Check is stack empty  
 //////////////////////  
 public boolean isEmpty()  
 {  
 if(this.Count == 0)  
 return true;  
 else  
 return false;  
 }  
 //Return count stack elements  
 public int count()  
 {  
 return this.Count;  
 }  
 ////////////////////  
 //Get stack elements  
 ////////////////////  
 public List<Type> getStackElements()  
 {  
 ArrayList<Type> list = new ArrayList<>();  
 //Create copy of top  
 MyNode<Type> top = new MyNode<>();  
 top.PreviousElement = this.Top.PreviousElement;  
 top.Data = this.Top.Data;  
 Type element;  
 while ((element = getElement(top)) != null) {  
 list.add(element);  
 top = top.PreviousElement;  
 }  
 return list;  
 }  
 /////////////////////////////  
 //Get data from "TOP" element  
 /////////////////////////////  
 private Type getElement(MyNode<Type> top)  
 {  
 MyNode<Type> element;  
 element = top;  
 if(top == null)  
 return null;  
 else  
 return element.Data;  
 }  
  
}

Файл Priorities

public enum Priorities  
{  
 HIGHEST(3),  
 MIDDLE(2),  
 LOWEST(1);  
 private Priorities(int value)  
 {  
 this.Value = value;  
 }  
 public static boolean hasHighestPriority(Priorities first, Priorities second)  
 {  
 if(first.Value > second.Value)  
 return true;  
 else  
 return false;  
 }  
 private int Value;  
}

Приложение Б

Листинг программы для раздела 2

Файл BinarySearch

public class BinarySearch<Type> {  
 Comparable<Type>[] Data;  
 Type SearchPattern;  
  
 /////////////  
 //Constructor  
 /////////////  
 BinarySearch(Comparable<Type>[] data) {  
 this.Data = data;  
 }  
  
 ///////////////////////////  
 //Wrapper for binary search  
 ///////////////////////////  
 public int search(Type pattern) throws Exception {  
 this.SearchPattern = pattern;  
 return searchProcess(0, this.Data.length - 1);  
 }  
  
 /////////////////////////  
 //Binary search algorithm  
 /////////////////////////  
 private int searchProcess(int start, int end) throws Exception {  
 if (start == end && !this.Data[start].equals(this.SearchPattern))  
 throw new Exception("Couldn't find the element");  
 int middleIndex = (start + end) / 2;  
 //Split array  
 if (this.Data[middleIndex].compareTo(this.SearchPattern) > 0)  
 return searchProcess(start, middleIndex);  
 else if (this.Data[middleIndex].compareTo(this.SearchPattern) < 0)  
 return searchProcess(middleIndex + 1, end);  
 else  
 return middleIndex;  
 }  
}

Файл BlockSearch

public class BlockSearch<Type> {  
 Comparable<Type>[] Data;  
 int SizeBlock;  
  
 /////////////  
 //Constructor  
 /////////////  
 BlockSearch(Comparable<Type>[] data) {  
 this.SizeBlock = 10;  
 this.Data = data;  
 }  
  
 /////////////  
 //Constructor  
 /////////////  
 BlockSearch(Comparable<Type>[] data, int sizeBlock) {  
 this.SizeBlock = sizeBlock;  
 this.Data = data;  
 }  
  
  
 public int search(Type pattern) throws Exception {  
 int startBlock = 0;  
 while (true) {  
 int endBlock = startBlock + this.SizeBlock;  
 if (startBlock + this.SizeBlock >= this.Data.length)  
 return searchInBlock(pattern, startBlock, this.Data.length-1);  
 if (this.Data[endBlock].compareTo(pattern) >= 0)  
 return searchInBlock(pattern, startBlock, endBlock);  
 startBlock += this.SizeBlock;  
 }  
 }  
  
  
 private int searchInBlock(Type pattern, int start, int end) throws Exception {  
 for (int i = start; i < end; i++)  
 if (pattern.equals(Data[i]))  
 return i;  
 throw new Exception("Couldn't find the element");  
 }  
}

Файл BoyerMoore

public class BoyerMoore {  
 String Data;  
 //Table stopSymbols:  
 // 'a' - 5;  
 // 'b' - 2;  
 // 'c' - 7;  
 //Integer - last character position in pattern (except last symbol)  
 Map<Character, Integer> tableStopSymbols;  
  
 /////////////  
 //Constructor  
 /////////////  
 BoyerMoore(String text) {  
 this.Data = text;  
 }  
  
  
 ///////////////////////////  
 //Searching pattern in text  
 ///////////////////////////  
 public int search(String pattern) throws Exception {  
  
 if(pattern.length() > this.Data.length())  
 throw new Exception("Couldn't find the pattern");  
  
 fillTableStopSymbols(pattern);  
 int indexFinishSearch = this.Data.length()-pattern.length();  
 int indexFinishPattern = pattern.length()-1;  
 //Cycle by text  
 for (int i = 0; i <= indexFinishSearch ; ) {  
 for(int j=indexFinishPattern; j >= 0; j--) {  
 //There's no Data symbol in pattern  
 if(this.Data.charAt(i+j) != pattern.charAt(j))  
 {  
 //How much must we move the pattern  
 if(tableStopSymbols.containsKey(this.Data.charAt(i+j)))  
 i += tableStopSymbols.get(this.Data.charAt(i+j));  
 else  
 i += pattern.length()-1;  
 break;  
 }  
 //Return index when we've found the pattern  
 if(j == 0)  
 return i;  
 }  
 }  
 throw new Exception("Couldn't find the pattern");  
 }  
  
 /////////////////////////////  
 //Fill the stop symbols table  
 /////////////////////////////  
 private void fillTableStopSymbols(String pattern) {  
 tableStopSymbols = new HashMap<>();  
 //negative 1 is used cause indexation from 0  
 //and algorithm doesn't use the last pattern symbol  
 for (int i = 1; i < pattern.length(); i++) {  
 tableStopSymbols.put(pattern.charAt(i-1), pattern.length()-i);  
 }  
 //Add last char in pattern  
 tableStopSymbols.put(pattern.charAt(pattern.length()-1), pattern.length()-1);  
 }  
}

Файл FastLinearSearch

public class FastLinearSearch<Type> {  
 private Comparable<Type>[] Data;  
 /////////////  
 //Constructor  
 /////////////  
 public FastLinearSearch(final Comparable<Type>[] data) {  
 Data = Arrays.copyOf(data, data.length + 1);  
 }  
 ///////////////////////////////////////////////////////  
 //Returns the index of the element that has been found  
 // or emit exception  
 //////////////////////////////////////////////////////  
 public int search(Comparable<Type> pattern) throws Exception {  
 Data[Data.length - 1] = pattern;  
 int index = 0;  
 while (true) {  
 if (pattern.equals(Data[index++]))  
 break;  
 }  
 //Check if the cycle was finished in the end  
 if (index == Data.length)  
 throw new Exception("Couldn't find the element");  
 else  
 return --index;  
 }  
}

Файл LinearSearch

public class LinearSearch<Type> {  
 private Type[] Data;  
 /////////////  
 //Constructor  
 /////////////  
 public LinearSearch(final Type[] data) {  
 Data = data;  
 }  
  
 //////////////////////////////////////////////////////  
 //Returns the index of the element that has been found  
 // or emit exception  
 //////////////////////////////////////////////////////  
 public int search(Type pattern) throws Exception {  
 int index = 0;  
 while (true) {  
 if (this.Data.length == index)  
 throw new Exception("Couldn't find the element");  
 if (pattern.equals(Data[index++]))  
 return --index;  
 }  
 }  
}

Файл SortedLinearSearch

public class SortedLinearSearch<Type> {  
 Type[] Data;  
  
 /////////////  
 //Constructor  
 /////////////  
 SortedLinearSearch(Type[] data) {  
 this.Data = (Type[]) new Object[data.length + 1];  
 Data = Arrays.copyOf(data, data.length + 1);  
 }  
  
 //////////////////////////////////////////////////////  
 //Returns the index of the element that has been found  
 // or emit exception  
 //////////////////////////////////////////////////////  
 public int search(Type pattern) throws Exception {  
 Data[Data.length - 1] = pattern;  
 int index = 0;  
 while (true) {  
 if (pattern.toString().compareTo(Data[index].toString()) <= 0)  
 break;  
 index++;  
 }  
 //Check if the cycle was finished in the end  
 if (Data[index].equals(pattern))  
 return index;  
 else  
 throw new Exception("Couldn't find the element");  
 }  
}

Файл Main

public class Main {  
  
 public static void main(String[] argc) throws Exception{  
 //Generate Data  
 String pattern = getWordToFind();  
 String[] strings = generateDomainInformation(pattern, 9000);  
  
 //Sorted copy  
 String[] sortedStrings = strings.clone();  
  
 ShellSort shellSort = new ShellSort(strings.clone(), SortType.DESCENDING);  
 String[] descendingSortedStrings = (String[])shellSort.sort();  
  
 Arrays.sort(sortedStrings);  
  
 LinearSearch<String> linearSearch = new LinearSearch<>(strings);  
 SortedLinearSearch<String> sortedLinearSearch = new SortedLinearSearch<>(sortedStrings);  
 FastLinearSearch<String> fastLinearSearch = new FastLinearSearch<>(strings);  
 BinarySearch<String> binarySearch = new BinarySearch<>(sortedStrings);  
 BlockSearch<String> blockSearch = new BlockSearch<>(sortedStrings);  
  
 long startLinear = System.nanoTime();  
 //Try to search using linear algorithm  
 try {  
 System.out.println("Position: using linear search algorithm: " + linearSearch.search(pattern));  
 } catch (Exception e) {  
 System.out.println(e.getMessage());  
 }  
 long endLinear = System.nanoTime();  
  
  
 long startSortedLinear = System.nanoTime();  
 //Try to search using fast sorted linear algorithm  
 try {  
 System.out.println("Position: using sorted linear search algorithm: " + sortedLinearSearch.search(pattern));  
 } catch (Exception e) {  
 System.out.println(e.getMessage());  
 }  
 long endSortedLinear = System.nanoTime();  
  
 long startFastLinear = System.nanoTime();  
 //Try to search using fast sorted linear algorithm  
 try {  
 System.out.println("Position: using fast linear search algorithm: " + fastLinearSearch.search(pattern));  
 } catch (Exception e) {  
 System.out.println(e.getMessage());  
 }  
 long endFastLinear = System.nanoTime();  
  
  
 long startBinary = System.nanoTime();  
 //Try to search using binary algorithm  
 try {  
 System.out.println("Position: using binary search algorithm: " + binarySearch.search(pattern));  
 } catch (Exception e) {  
 System.out.println(e.getMessage());  
 }  
 long endBinary = System.nanoTime();  
  
  
 long startBlock = System.nanoTime();  
 //Try to search using block algorithm  
 try {  
 System.out.println("Position: using block algorithm: " + blockSearch.search(pattern));  
 } catch (Exception e) {  
 System.out.println(e.getMessage());  
 }  
 long endBlock = System.nanoTime();  
  
  
 String sortedText = stringFromArray(strings);  
 String descendingSortedText = stringFromArray(descendingSortedStrings);  
 String textRaw = stringFromArray(strings);  
 long startBoyerMooreRawData = System.nanoTime();  
 //Run cycle to find out approximate time  
 BoyerMoore boyerMoore = new BoyerMoore(textRaw);  
 try {  
 System.out.println(boyerMoore.search(pattern));  
 } catch (Exception e) {  
 System.out.println(e.getMessage());  
 }  
 long endBoyerMooreRawData = System.nanoTime();  
  
  
 long startBoyerMooreSorted = System.nanoTime();  
 //Run cycle to find out approximate time  
 boyerMoore = new BoyerMoore(sortedText);  
 try {  
 System.out.println(boyerMoore.search(pattern));  
 } catch (Exception e) {  
 System.out.println(e.getMessage());  
 }  
 long endBoyerMooreSorted = System.nanoTime();  
  
  
 long startBoyerMooreDescSorted= System.nanoTime();  
 //Run cycle to find out approximate time  
 boyerMoore = new BoyerMoore(descendingSortedText);  
 try {  
 System.out.println(boyerMoore.search(pattern));  
 } catch (Exception e) {  
 System.out.println(e.getMessage());  
 }  
 long endBoyerMooreDescSorted = System.nanoTime();  
  
 System.out.println("Time BoyerMoore raw data: " + (endBoyerMooreRawData - startBoyerMooreRawData));  
 System.out.println("Time BoyerMoore sorted data: " + (endBoyerMooreSorted - startBoyerMooreSorted));  
 System.out.println("Time BoyerMoore descending sorted data: " + (endBoyerMooreDescSorted - startBoyerMooreDescSorted));  
 System.out.println("Time Linear: " + (endLinear - startLinear));  
 System.out.println("Time SortedLinear: " + (endSortedLinear - startSortedLinear));  
 System.out.println("Time FastLinear: "+ (endFastLinear - startFastLinear));  
 System.out.println("Time Binary: " + (endBinary - startBinary));  
 System.out.println("Time Block: " + (endBlock - startBlock));  
 }  
 //////////////////////////////////////////////////////////////////////////  
 //Generates a sequence with only one occurrence in expression "wordToFind"  
 /////////////////////////////////////////////////////////////////////////  
 private static String[] generateDomainInformation(String wordToFind, int countElements) throws Exception{  
 //Generate random pattern position  
 Random random = new Random(System.currentTimeMillis());  
 int patternPosition = random.nextInt(countElements-1);  
 //To form result array  
 String [] result = new String[countElements];  
 for(int i=0; i<countElements; i++) {  
 if(i==patternPosition) {  
 result[i] = wordToFind;  
 continue;  
 }  
 String currentWord = getWordToFind();  
 if (currentWord.compareTo(wordToFind) == 0) {  
 i--;  
 continue;  
 }  
 result[i] = currentWord;  
 }  
 return result;  
 }  
 ///////////////////////////////////////////  
 //Method selects the random word from array  
 ///////////////////////////////////////////  
 private static String getWordToFind() throws Exception{  
 String[] arrayDomainInf = {"адрес", "почтовый", "отделение", "адресат","отправитель", "индекс", "накладная", "абонент",  
 "возврат", "знак", "марка", "оплата", "входящий", "исходящий", "служба", "сторона", "вес", "пакет", "письмо", "посылка",  
 "штемпель", "класс", "вручение", "секограмма"};  
 Random random = new Random(System.currentTimeMillis());  
 Thread.sleep(1);  
 int index = random.nextInt(arrayDomainInf.length-1);  
 return arrayDomainInf[index];  
 }  
 /////////////////////////////  
 //Generates string from array  
 /////////////////////////////  
 private static String stringFromArray(String[] array)  
 {  
 String result = "";  
 for(int i=0; i<array.length; i++) {  
 result += array[i] + " ";  
 }  
 return result;  
 }  
}

Приложение В

Листинг программы для раздела 3

Файл InsertionSort

public class InsertionSort<Type> {  
  
 private Comparable<Type>[] Data;  
 private SortType SortType;  
  
 public InsertionSort(Comparable<Type>[] array, SortType sortType) {  
 Data = Arrays.copyOf(array, array.length);  
 SortType = sortType;  
 }  
  
 public Type[] sort() throws Exception {  
 if (SortType == SortType.ASCENDING)  
 return ascendingSort();  
 else  
 return descendingSort();  
 }  
  
 ///////////////////////  
 //From low to high sort  
 ///////////////////////  
 private Type[] ascendingSort() throws Exception {  
 if (this.Data.length == 0)  
 throw new Exception("The array has 0 length");  
 int j;  
 for (int i = 0; i < Data.length; i++) {  
 Type elementToSort = (Type) Data[i];  
 //Search place in sorted part  
 for (j = i - 1; j >= 0 && (Data[j].compareTo(elementToSort) > 0); j--) {  
 Data[j + 1] = Data[j]; //Move current element to right while place isn't right  
 }  
 //The right place has been found  
 Data[j + 1] = (Comparable<Type>) elementToSort;  
 }  
 return (Type[]) Data;  
 }  
  
 ///////////////////////  
 //From high to low sort  
 ///////////////////////  
 private Type[] descendingSort() throws Exception {  
 if (this.Data.length == 0)  
 throw new Exception("The array has 0 length");  
 int j;  
 for (int i = 0; i < Data.length; i++) {  
 Type elementToSort = (Type) Data[i];  
 //Search place in sorted part  
 for (j = i-1; j>=0 && (Data[j].compareTo(elementToSort) < 0); j--) {  
 Data[j + 1] = Data[j]; //Move current element to right while place isn't right  
 }  
 //The right place has been found  
 Data[j + 1] = (Comparable<Type>) elementToSort;  
 }  
 return (Type[]) Data;  
 }  
}

Файл ShellSort

public class ShellSort<Type> {  
 private Comparable<Type>[] Data;  
 private SortType SortType;  
 private List<Integer> Incrementation;  
  
 //Using default Sedjvik's series  
 public ShellSort(final Comparable<Type>[] array, SortType type) {  
 this.Data = array;  
 this.SortType = type;  
 Incrementation = new ArrayList<>();  
 calculateSedjvikSeries();  
 }  
  
 //Using another series  
 public ShellSort(final Comparable<Type>[] array, SortType type, boolean useMySeries) {  
 this.Data = array;  
 this.SortType = type;  
 Incrementation = new ArrayList<>();  
 if (useMySeries)  
 calculateMySeries();  
 else  
 calculateSedjvikSeries();  
 }  
  
  
 public Type[] sort() throws Exception {  
 if (SortType == SortType.ASCENDING)  
 return ascendingSort();  
 else  
 return descendingSort();  
 }  
  
 ///////////////////////  
 //From low to high sort  
 ///////////////////////  
 private Type[] ascendingSort() throws Exception {  
 if (this.Data.length == 0)  
 throw new Exception("The array has 0 length");  
 int countCycles = Incrementation.size() - 1;  
 while (countCycles >= 0) {  
 int dividePart = Incrementation.get(countCycles--);  
 //Using Insertion sort  
 int j;  
 for (int i = dividePart; i < Data.length; i++) {  
 Type elementToSort = (Type) Data[i];  
 //Search place in sorted part  
 for (j = i - 1; j >= 0 && (Data[j].compareTo(elementToSort) > 0); j--) {  
 Data[j + 1] = Data[j]; //Move current element to right while place isn't right  
 }  
 //The right place has been found  
 Data[j + 1] = (Comparable<Type>) elementToSort;  
 }  
 }  
 return (Type[]) Data;  
 }  
  
 ///////////////////////  
 //From high to low sort  
 ///////////////////////  
 private Type[] descendingSort() throws Exception {  
 if (this.Data.length == 0)  
 throw new Exception("The array has 0 length");  
 int countCycles = Incrementation.size() - 1;  
 while (countCycles >= 0) {  
 int dividePart = Incrementation.get(countCycles--);  
 //Using Insertion sort  
 int j;  
 for (int i = dividePart; i < Data.length; i++) {  
 Type elementToSort = (Type) Data[i];  
 //Search place in sorted part  
 for (j = i - 1; j >= 0 && (Data[j].compareTo(elementToSort) < 0); j--) {  
 Data[j + 1] = Data[j]; //Move current element to right while place isn't right  
 }  
 //The right place has been found  
 Data[j + 1] = (Comparable<Type>) elementToSort;  
 }  
 }  
 return (Type[]) Data;  
 }  
  
  
 /////////////////////////////  
 //Calculate series by Sedjvik  
 /////////////////////////////  
 private void calculateSedjvikSeries() {  
 Incrementation.add(1);  
 int i = 1;  
 do {  
 if (i % 2 == 0)  
 Incrementation.add((int) (9 \* Math.pow(2, i) - 9 \* Math.pow(2, i / 2)));  
 else  
 Incrementation.add((int) (8 \* Math.pow(2, i) - 6 \* Math.pow(2, (i + 1) / 2) + 1));  
 i++;  
  
 } while (Incrementation.get(i - 1) < this.Data.length && 3 \* Incrementation.get(i - 1) < Data.length);  
 }  
  
 /////////////////////////////////////////  
 //Calculate series using my own algorithm  
 /////////////////////////////////////////  
 private void calculateMySeries() {  
 Incrementation.add(1);  
 int i = 1;  
 do {  
 Incrementation.add((int)Math.pow(i+1, Incrementation.get(i-1)));  
 i++;  
  
 } while (Incrementation.get(i - 1) < this.Data.length );  
 }  
  
}

Файл SortType

public enum SortType {  
 ASCENDING(0), //FROM LOW TO HIGH  
 DESCENDING(1); //FROM HIGH TO LOW  
  
 SortType(int value) {  
 this.Value = value;  
 }  
  
 private int Value;  
}

Файл QuickSort

public class QuickSort<Type> {  
 Comparable<Type>[] Data;  
 SortType SortType;  
  
 QuickSort(final Comparable<Type>[] array, SortType type) {  
 this.SortType = type;  
 this.Data = array;  
 }  
  
 public Type[] sort() throws Exception {  
 if (SortType == SortType.ASCENDING)  
 ascendingSort(0, this.Data.length - 1);  
 else  
 descendingSort(0, this.Data.length - 1);  
 return (Type[]) Data;  
 }  
  
 ///////////////////////  
 //From low to high sort  
 ///////////////////////  
 private void ascendingSort(int left, int right) throws Exception {  
 if (this.Data.length == 0)  
 throw new Exception("The array has 0 length");  
 if (left < right) {  
 //TODO: невозможно вычислять медиану, т.к. для ее вычисления необходима сортировка. Можно использовать среднее арифметическое  
 Type middle = (Type) this.Data[(left + right) / 2];  
 int i = left;  
 int j = right;  
 while (i <= j) {  
 while (this.Data[i].compareTo(middle) < 0) i++;  
 while (this.Data[j].compareTo(middle) > 0) j--;  
 //Swap elements  
 if (i <= j) {  
 Comparable<Type> temp = this.Data[i];  
 this.Data[i] = this.Data[j];  
 this.Data[j] = temp;  
 i++;  
 j--;  
 }  
 }  
 //Call recursive if there is something to sort  
 if (j > 0) ascendingSort(left, j);  
 if (right > i) ascendingSort(j + 1, right);  
 }  
 }  
  
 ///////////////////////  
 //From high to low sort  
 ///////////////////////  
 private void descendingSort(int left, int right) throws Exception {  
 if (this.Data.length == 0)  
 throw new Exception("The array has 0 length");  
 if (left < right) {  
 //TODO: невозможно вычислять медиану, т.к. для ее вычисления необходима сортировка. Можно использовать среднее арифметическое  
 Type middle = (Type) this.Data[(left + right) / 2];  
 int i = left;  
 int j = right;  
 while (i <= j) {  
 while (this.Data[i].compareTo(middle) > 0) i++;  
 while (this.Data[j].compareTo(middle) < 0) j--;  
 //Swap elements  
 if (i <= j) {  
 Comparable<Type> temp = this.Data[i];  
 this.Data[i] = this.Data[j];  
 this.Data[j] = temp;  
 i++;  
 j--;  
 }  
 }  
 //Call recursive if there is something to sort  
 if (j > 0) descendingSort(left, j);  
 if (right > i) descendingSort(j + 1, right);  
 }  
 }  
}

Файл Main

public class Main {  
  
 public static void main(String[] argc) throws Exception {  
 Double[] RawArray = generateDomainInformation(60000);  
  
  
 /\*System.out.println("RAW data: ");  
 for (Double current : RawArray) {  
 System.out.format(String.format("%.2f ", current));  
 }\*/  
 System.out.println();  
  
 long timeStartInsertion = System.currentTimeMillis();  
 InsertionSort insertionSort = new InsertionSort(RawArray, SortType.ASCENDING);  
 Object[] sorted = insertionSort.sort();  
 /\*System.out.println("Sorted with insertionSort data: ");  
 for (Object current : sorted) {  
 System.out.print(String.format("%.2f ", current));  
 }  
 System.out.println();\*/  
 long timeEndInsertion = System.currentTimeMillis();  
  
 long timeStartInsertionSorted = System.currentTimeMillis();  
 insertionSort = new InsertionSort((Double[])sorted, SortType.ASCENDING);  
 sorted = insertionSort.sort();  
 /\*System.out.println("Sorted with insertionSort data that has been sorted: ");  
 for (Object current : sorted) {  
 System.out.print(String.format("%.2f ", current));  
 }  
 System.out.println();\*/  
 long timeEndInsertionSorted = System.currentTimeMillis();  
  
 long timeStartInsertionSortedBack = System.currentTimeMillis();  
 insertionSort = new InsertionSort((Double[])sorted, SortType.DESCENDING);  
 sorted = insertionSort.sort();  
 /\*System.out.println("Sorted back with insertionSort data that has been sorted: ");  
 for (Object current : sorted) {  
 System.out.print(String.format("%.2f ", current));  
 }  
 System.out.println();\*/  
 long timeEndInsertionSortedBack = System.currentTimeMillis();  
  
 long timeStartShell = System.currentTimeMillis();  
 ShellSort shellSort = new ShellSort(RawArray, SortType.ASCENDING);  
 sorted = shellSort.sort();  
 /\*System.out.println("Sorted with SHELL data: ");  
 for (Object current : sorted) {  
 System.out.print(String.format("%.2f ", current));  
 }  
 System.out.println();\*/  
 long timeEndShell = System.currentTimeMillis();  
  
 long timeStartShellSorted = System.currentTimeMillis();  
 shellSort = new ShellSort((Double[])sorted, SortType.ASCENDING);  
 sorted = shellSort.sort();  
 /\*System.out.println("Sorted with SHELL data that has been sorted: ");  
 for (Object current : sorted) {  
 System.out.print(String.format("%.2f ", current));  
 }\*/  
 System.out.println();  
 long timeEndShellSorted = System.currentTimeMillis();  
  
 long timeStartShellSortedBack = System.currentTimeMillis();  
 shellSort = new ShellSort((Double[])sorted, SortType.DESCENDING);  
 sorted = shellSort.sort();  
 /\*System.out.println("Sorted back with SHELL data that has been sorted: ");  
 for (Object current : sorted) {  
 System.out.print(String.format("%.2f ", current));  
 }  
 System.out.println();\*/  
 long timeEndShellSortedBack = System.currentTimeMillis();  
  
 long timeStartQuick = System.currentTimeMillis();  
 QuickSort quickSort = new QuickSort(RawArray, SortType.ASCENDING);  
 sorted = quickSort.sort();  
 /\*System.out.println("Sorted with QUICK data: ");  
 for (Object current : sorted) {  
 System.out.print(String.format("%.2f ", current));  
 }  
 System.out.println();\*/  
 long timeEndQuick = System.currentTimeMillis();  
  
 long timeStartQuickSorted = System.currentTimeMillis();  
 quickSort = new QuickSort((Double[])sorted, SortType.ASCENDING);  
 sorted = quickSort.sort();  
 /\*System.out.println("Sorted with QUICK data that has been sorted: ");  
 for (Object current : sorted) {  
 System.out.print(String.format("%.2f ", current));  
 }  
 System.out.println();\*/  
 long timeEndQuickSorted = System.currentTimeMillis();  
  
 long timeStartQuickSortedBack = System.currentTimeMillis();  
 quickSort = new QuickSort((Double[])sorted, SortType.DESCENDING);  
 sorted = quickSort.sort();  
 /\*System.out.println("Sorted back with SHELL data that has been sorted: ");  
 for (Object current : sorted) {  
 System.out.print(String.format("%.2f ", current));  
 }  
 System.out.println();\*/  
 long timeEndQuickSortedBack = System.currentTimeMillis();  
  
 System.out.println("TIME INSERTION sorted raw: " + (timeEndInsertion-timeStartInsertion));  
 System.out.println("TIME INSERTION sorted data that have been sorted: " + (timeEndInsertionSorted-timeStartInsertionSorted));  
 System.out.println("TIME INSERTION sorted back: "+(timeEndInsertionSortedBack-timeStartInsertionSortedBack));  
  
 System.out.println("TIME SHELL sorted raw: " + (timeEndShell-timeStartShell));  
 System.out.println("TIME SHELL sorted data that have been sorted: " + (timeEndShellSorted-timeStartShellSorted));  
 System.out.println("TIME SHELL sorted back: "+(timeEndShellSortedBack-timeStartShellSortedBack));  
  
 System.out.println("TIME QUICK sorted raw: " + (timeEndQuick-timeStartQuick));  
 System.out.println("TIME QUICK sorted data that have been sorted: " + (timeEndQuickSorted-timeStartQuickSorted));  
 System.out.println("TIME QUICK sorted back: "+(timeEndQuickSortedBack-timeStartQuickSortedBack));  
  
  
 }  
 //////////////////////////////////////////////////////////////////////////  
 //Generates a sequence with only one occurrence in expression "wordToFind"  
 /////////////////////////////////////////////////////////////////////////  
 private static Double[] generateDomainInformation (int countElements) throws Exception {  
 //Generate random  
 Random random = new Random(System.currentTimeMillis());  
 //To form result array  
 Double [] result = new Double[countElements];  
 for(int i=0; i<countElements; i++) {  
 Thread.sleep(1);  
 result[i] = 0 + random.nextDouble() \* (10000-0 + 1); //min + rand\*(max-min+1)  
 }  
 return result;  
 }  
}

Приложение Г

Листинг программы для раздела 4

Файл BTreeNode

class BTreeNode<Type> {  
 public boolean Leaf; //Is the node leaf  
 public int Size; //Current values count  
 public Comparable<Type>[] Values; //Values (keys)  
 public BTreeNode[] Children; //Pointers  
  
 /////////////////////////////////////////////  
 //Constructor  
 //parameter "t" is the minimum degree of tree  
 //t=1 - binary TREE  
 //t=2 - 2-3 TREE  
 //etc.  
 ////////////////////////////////////////////  
 BTreeNode(int t) {  
 int Size = 0;  
  
 //one extra place in case of sharing value, pretending overflow  
 this.Children = new BTreeNode[2 \* t + 1];  
 this.Leaf = true;  
 this.Values = new Comparable[2 \* t];  
 }  
  
 ///////////////////////////////  
 //Returns true if the node full  
 ///////////////////////////////  
 boolean isFull() {  
 if (this.Size == Values.length - 1)//-1 because one is extra (see constructor)  
 return true;  
 else  
 return false;  
 }  
}

Файл BTree

class BTree<Type> {  
 private BTreeNode Root;  
 private Integer T;  
  
 /////////////////////////////////////////////  
 //Constructor  
 //parameter "t" is the minimum degree of tree  
 //t=1 - binary TREE  
 //t=2 - 2-3 TREE  
 //etc.  
 /////////////////////////////////////////////  
 BTree(int t) throws Exception {  
 if (t < 2)  
 throw new Exception("Degree must be more than 1");  
 this.T = t;  
 this.Root = new BTreeNode(t);  
 }  
  
 ////////////////////////////////////////////////  
 //Delete element value from the tree if possible  
 ////////////////////////////////////////////////  
 public void delete(Comparable<Type> value) {  
 while (contains(value)) {  
 delete(value, this.Root);  
 if (this.Root.Size == 0)  
 this.Root = this.Root.Children[0];  
  
 }  
 }  
  
 //////////////////////////////////////////////  
 //Recursive deletion with rearranging the tree  
 ///////////////////////////////////////////////  
 private void delete(Comparable<Type> value, BTreeNode parent) {  
  
 ///Check all Values in parent  
 for (int i = 0; i < parent.Size; i++) {  
 //Just remove the value in case it's leaf  
 if (parent.Leaf && parent.Values[i].compareTo(value) == 0) {  
 //Move all elements to the left  
 for (int j = i; j < parent.Size - 1; j++)  
 parent.Values[j] = parent.Values[j + 1];  
 parent.Size--;  
 //Set removed value as null  
 parent.Values[parent.Size] = null;  
 return; //finish  
 }  
 //parent isn't leaf but the value in parent  
 else if (!parent.Leaf && parent.Values[i].compareTo(value) == 0) {  
 BTreeNode leftChild = parent.Children[i];  
 BTreeNode rightChild = parent.Children[i + 1];  
 //If the value from leftChild can be move to the parent  
 if (leftChild.Size >= T) {  
 shareValue(parent, i, i + 1);  
 delete(value, rightChild);  
 return;  
 }  
 //If the Value from rightChild can be moved to the parent  
 else if (rightChild.Size >= T) {  
 //Get newValue from rightChild  
 shareValue(parent, i + 1, i);  
 delete(value, leftChild);  
 return;  
 }  
 //Not good, we have to merge leftChild, rightChild and Value  
 //Value will be median  
 else {  
 parent = mergeNodes(parent, i);  
 delete(value, parent);  
 return;  
 }  
 }  
 }  
 if (!parent.Leaf) {  
 ///SEE NEXT LEVEL  
 ///Check children first before going deeper inside the tree  
 ///Determine the appropriate branch to go further  
 int parentIndex = parent.Size - 1;  
 int childIndex = parent.Size;  
 for (int i = 0; i < parent.Size; i++) {  
 if (parent.Values[i].compareTo(value) > 0) {  
 parentIndex = i;  
 childIndex = i;  
 break; //not increase index, because it could lead to wrong branch  
 }  
 }  
 //Make sure that we can go deeper  
 BTreeNode nextTree = parent.Children[childIndex];  
 if (nextTree.Size == T - 1) {  
 //If the value from leftChild can be move to the parent  
 if (childIndex == 0) {  
 if (parent.Children[childIndex + 1].Size >= T)  
 shareValue(parent, childIndex + 1, childIndex);  
 else  
 nextTree = mergeNodes(parent, parentIndex);  
 }  
 //If the Value from rightChild can be moved to the parent  
 else if (childIndex == parent.Size) {  
 if (parent.Children[childIndex - 1].Size >= T)  
 shareValue(parent, childIndex - 1, childIndex);  
 else  
 nextTree = mergeNodes(parent, parentIndex);  
 } else if (parent.Children[childIndex - 1].Size >= T || parent.Children[childIndex + 1].Size >= T) {  
 //Get Value from left or from right  
 if (parent.Children[childIndex - 1].Size >= T)  
 shareValue(parent, childIndex - 1, childIndex);  
 else  
 shareValue(parent, childIndex + 1, childIndex);  
 }  
 //Not good, we have to merge leftChild, rightChild and Value  
 //Value will be median  
 else {  
 nextTree = mergeNodes(parent, parentIndex);  
 }  
 }  
 //Now we are sure that subtree has enough values to delete  
 delete(value, nextTree);  
 }  
 }  
  
 //////////////////////////////////////////  
 //Function to share value between children  
 ///////////////////////////////////////////  
 public void shareValue(BTreeNode parent, int indexSource, int indexDest) {  
 int indexInParent = (indexDest + indexSource) / 2;  
 BTreeNode source;  
 BTreeNode dest;  
 dest = parent.Children[indexDest];  
 source = parent.Children[indexSource];  
 //Get value from subtree that is located more right  
 if (indexSource > indexDest) {  
 //Set new value from parent to dest  
 dest.Values[dest.Size] = parent.Values[indexInParent];  
 //Set value from source to parent  
 parent.Values[indexInParent] = source.Values[0];  
 //Copy child to the dest  
 if (!dest.Leaf) {  
 dest.Children[dest.Size + 1] = source.Children[0];  
 //Move all children in source to the beginning  
 for (int i = 0; i < source.Size; i++)  
 source.Children[i] = source.Children[i + 1];  
 }  
 //Move all children in the source to the beginning  
 for (int i = 0; i < source.Size - 1; i++)  
 source.Values[i] = source.Values[i + 1];  
 dest.Size++;  
 source.Size--;  
 //Free old child and value in source  
 source.Values[source.Size] = null;  
 source.Children[source.Size + 1] = null;  
 }  
 //Get value from subtree that is located more left  
 else {  
 //Free one position from the beginning to put there new value  
 for (int i = dest.Size; i > 0; i--)  
 dest.Values[i] = dest.Values[i - 1];  
 //Set new value from parent to dest  
 dest.Values[0] = parent.Values[indexInParent];  
 //Set value from source to parent  
 parent.Values[indexInParent] = source.Values[source.Size - 1];  
 //Copy child to the dest  
 if (!dest.Leaf) {  
 //Free one position from the beginning to put there new child  
 for (int i = dest.Size + 1; i > 0; i--)  
 dest.Children[i] = dest.Children[i - 1];  
 //Copy child from source  
 dest.Children[0] = source.Children[source.Size];  
 }  
 dest.Size++;  
 source.Size--;  
 //Free old child and value in source  
 source.Children[source.Size + 1] = null;  
 source.Values[source.Size] = null;  
 }  
 }  
  
 ////////////////////////////////////  
 //Function merges two children nodes  
 ////////////////////////////////////  
 public BTreeNode mergeNodes(BTreeNode parent, int indexChildToMerge) {  
 int i = indexChildToMerge;  
 BTreeNode leftChild = parent.Children[i];  
 BTreeNode rightChild = parent.Children[i + 1];  
 //Copy parent value to leftChild  
 leftChild.Values[leftChild.Size] = parent.Values[i];  
 leftChild.Size++;  
 //Copy rightChild to leftChild  
 for (int j = 0; j < rightChild.Size; j++) {  
 leftChild.Values[leftChild.Size + j] = rightChild.Values[j];  
 }  
 //Copy all rightChild children to leftChild  
 for (int j = 0; j <= rightChild.Size; j++) {  
 leftChild.Children[leftChild.Size + j] = rightChild.Children[j];  
 }  
 //Set new leftChildSize  
 leftChild.Size += rightChild.Size;  
  
 //clear "value" from parent  
 //Put "value" from parent to the layer below  
 for (int j = i; j < parent.Size - 1; j++) {  
 parent.Values[j] = parent.Values[j + 1];  
 parent.Children[j + 1] = parent.Children[j + 2];  
 }  
 parent.Size--;  
 //free value and child from parent that have been removed  
 parent.Values[parent.Size] = null;  
 parent.Children[parent.Size + 1] = null;  
 //Return merged node  
 return leftChild;  
 }  
  
 ///////////////////////////////////////////  
 //Returns true if the tree contains "value"  
 ///////////////////////////////////////////  
 public boolean contains(Comparable<Type> value) {  
 try {  
 BTreeNode found = search(value, this.Root);  
 return true;  
 } catch (Exception err) {  
 return false;  
 }  
 }  
  
 ///////////////////////////////////////////  
 //Method returns subTree or emit Exception  
 //////////////////////////////////////////  
 public BTreeNode search(Comparable<Type> value, BTreeNode parent) throws Exception {  
 //Check all parent values  
 for (int i = 0; i < parent.Size; i++) {  
 if (parent.Values[i].compareTo(value) == 0)  
 return parent;  
 }  
 //Determine the appropriate branch to go further  
 for (int i = 0; i < parent.Size; i++) {  
 if (parent.Values[i].compareTo(value) > 0 && !parent.Leaf)  
 return search(value, parent.Children[i]);  
 }  
 //Try to find value in the last branch if possible  
 if (!parent.Leaf)  
 return search(value, parent.Children[parent.Size]);  
 throw new Exception("Couldn't find value");  
 }  
  
 ////////////////////////////  
 //Add new value to the tree  
 ///////////////////////////  
 public void add(Comparable<Type> value) {  
 //if we have to create new root  
 if (Root.isFull()) {  
 BTreeNode newRoot = new BTreeNode(T);  
 //Set the old root as a child newRoot  
 newRoot.Children[0] = this.Root;  
 newRoot.Leaf = false;  
 //Set newRoot as a root  
 this.Root = newRoot;  
 //The old root is full, so we have to split it  
 splitChildNode(this.Root, 0);  
 }  
 add(value, this.Root);  
 }  
  
 //////////////////////////////////////  
 //Private method to use recursive call  
 //////////////////////////////////////  
 private void add(Comparable<Type> value, BTreeNode node) {  
 int i = node.Size - 1; //Insertion position  
 if (node.Leaf) {  
 //Find the place to insert the value into leaf  
 while (i >= 0 && value.compareTo((Type) node.Values[i]) < 0) {  
 node.Values[i + 1] = node.Values[i];  
 i--;  
 }  
 //Do insertion  
 node.Values[i + 1] = value;  
 node.Size++;  
 //TODO: disk write "node"  
 } else {  
 //Find the right subTree to insert the value into  
 while (i >= 0 && value.compareTo((Type) node.Values[i]) < 0)  
 i--;  
 i++;  
 //TODO: disk read "child" instead of next string  
 BTreeNode child = node.Children[i];  
 //Split child if need  
 if (child.isFull()) {  
 splitChildNode(node, i);  
 //Correct insertion index after split because method "split" adds one element to  
 //parent node from child  
 if (value.compareTo((Type) node.Values[i]) > 0)  
 i++;  
 child = node.Children[i];  
 }  
 //recursive call to insert value into the child  
 add(value, child);  
 }  
 }  
  
 //////////////////////////////////////////////////////  
 //Split child node with index i and update parent node  
 //////////////////////////////////////////////////////  
 private void splitChildNode(BTreeNode parent, int childIndexToSplit) {  
 int i = childIndexToSplit; //index which child should be split  
 //Create children  
 BTreeNode rightChild = new BTreeNode(T);  
 BTreeNode leftChild = parent.Children[i];  
  
 //Customize rightChild  
 rightChild.Leaf = leftChild.Leaf; //Set the same status as splitting node(it can be leaf or not)  
 rightChild.Size = leftChild.Size / 2; //WARNING not to use binary tree  
  
 //Resize leftChild  
 leftChild.Size = rightChild.Size;  
  
 //Move splitting value to parent  
 //Some elements can already exist in parent  
 for (int j = parent.Size; j > i; j--)  
 parent.Values[j] = parent.Values[j - 1];  
 parent.Values[i] = leftChild.Values[leftChild.Size];  
 leftChild.Values[leftChild.Size] = null; //remove the value that has been moved  
  
 //Fill right node with values  
 for (int j = 0; j < rightChild.Size; j++) {  
 rightChild.Values[j] = leftChild.Values[leftChild.Size + 1 + j];  
 //Set null all values from the leftChild that have been moved  
 leftChild.Values[leftChild.Size + 1 + j] = null;  
 }  
  
 //Copy if need all children  
 if (!leftChild.Leaf) {  
 for (int j = 0; j <= rightChild.Size; j++) {  
 rightChild.Children[j] = leftChild.Children[leftChild.Size + 1 + j];  
 //Set null all children that have been moved  
 leftChild.Children[leftChild.Size + 1 + j] = null;  
 }  
 }  
  
 //Set customized rightChild to parent  
 for (int j = parent.Size; j > i; j--)  
 parent.Children[j + 1] = parent.Children[j];  
 parent.Children[i + 1] = rightChild;  
  
 //increment parent size  
 parent.Size++;  
 //TODO: disk write "leftChild", "rightChild", "parent"  
 }  
  
 ////////////////  
 //Print the tree  
 /////////////////  
 public void printByLayer() {  
  
 Queue<BTreeNode> queue = new LinkedList<>(); //queue to add  
 int currentCount = 1; //Count elements on current layer  
 int nextCount = 0; //Count elements on the next layer  
 queue.add(this.Root);  
 while (!queue.isEmpty()) {  
 BTreeNode currentNode = queue.poll();  
  
 //print node  
 for (int i = 0; i < currentNode.Size; i++)  
 System.out.print(currentNode.Values[i] + " ");  
 System.out.print(" ");  
 currentCount--;  
  
 //Add to the queue all children  
 for (int i = 0; i <= currentNode.Size; i++) {  
 if (currentNode.Children[i] != null) {  
 queue.add(currentNode.Children[i]);  
 nextCount++;  
 }  
 }  
  
 //Next layer?  
 if (currentCount == 0) {  
 System.out.println();  
 currentCount = nextCount;  
 nextCount = 0;  
 }  
  
 }  
 }  
  
}

Файл Main

public class Main {  
 public static void main(String[] argv) throws Exception {  
 //Create tree  
 BTree bTree = new BTree(10); //Create 2-3-4-... tree  
  
 bTree.add(1);  
 bTree.add(3);  
 bTree.add(-2);  
 bTree.add(8);  
 bTree.add(4);  
 bTree.add(6);  
 bTree.add(-8);  
 bTree.add(12);  
 bTree.add(-4);  
 bTree.add(-3);  
 bTree.add(2);  
 bTree.add(3);  
 bTree.add(-1);  
 bTree.add(4);  
 bTree.add(1);  
 bTree.add(5);  
 bTree.add(10);  
 bTree.add(3);  
 bTree.add(-1);  
 bTree.add(0);  
 bTree.add(-4);  
 bTree.add(-6);  
 bTree.add(-8);  
 bTree.add(-2);  
 bTree.add(3);  
 bTree.add(4);  
 bTree.add(9);  
 bTree.add(11);  
 bTree.add(12);  
 bTree.add(5);  
  
 System.out.println("INITIAL");  
 bTree.printByLayer();  
  
 bTree.delete(8);  
 System.out.println("REMOVED 8");  
 bTree.printByLayer();  
  
 bTree.delete(1);  
 System.out.println("REMOVED 1");  
 bTree.printByLayer();  
  
 bTree.delete(3);  
 System.out.println("REMOVED 3");  
 bTree.printByLayer();  
  
 bTree.delete(-2);  
 System.out.println("REMOVED -2");  
 bTree.printByLayer();  
  
 bTree.delete(-4);  
 System.out.println("REMOVED -4");  
 bTree.printByLayer();  
  
 bTree.delete(-8);  
 System.out.println("REMOVED -8");  
 bTree.printByLayer();  
  
 bTree.delete(4);  
 System.out.println("REMOVED 4");  
 bTree.printByLayer();  
  
 bTree.delete(10);  
 System.out.println("REMOVED 10");  
 bTree.printByLayer();  
  
 bTree.delete(11);  
 System.out.println("REMOVED 11");  
 bTree.printByLayer();  
  
  
 bTree.delete(12);  
 System.out.println("REMOVED 12");  
 bTree.printByLayer();  
  
 bTree.delete(5);  
 System.out.println("REMOVED 5");  
 bTree.printByLayer();  
  
 bTree.delete(-1);  
 System.out.println("REMOVED -1");  
 bTree.printByLayer();  
  
 bTree.delete(0);  
 System.out.println("REMOVED 0");  
 bTree.printByLayer();  
  
  
 bTree.delete(2);  
 System.out.println("REMOVED 2");  
 bTree.printByLayer();  
  
 }  
}

Приложение Д

Листинг программы для раздела 5

Файл AbstractHash

public abstract class AbstractHash {  
 private int TableSize;  
 private Object[] Data;  
 private Integer[] ListCollisions;  
 private int CollisionCount;  
  
 AbstractHash(int size) {  
 this.TableSize = size;  
 this.Data = new Object[size];  
 this.CollisionCount = 0;  
 this.ListCollisions = new Integer[this.TableSize];  
 for(int i=0; i<ListCollisions.length; i++)  
 ListCollisions[i] = 0;  
 }  
 public int getCollisionCount() {  
 return CollisionCount;  
 }  
 public Integer[] getCollisions() {  
 return ListCollisions;  
 }  
  
 ////////////////////////////////  
 //Method generates simple number  
 ////////////////////////////////  
 public Integer randSimpleNumber() {  
 Random random = new Random(System.currentTimeMillis());  
 ArrayList<Integer> list = new ArrayList<Integer>();  
 int max = random.nextInt(10000);  
 for (int i = 2; i < max; i++) {  
 int k = 0;  
 for (int j = 2; j < i / 2; j++) {  
 if (i % j == 0)  
 k++;  
 }  
 if (k == 0)  
 list.add(i);  
  
 }  
 if (list.size() == 0)  
 return randSimpleNumber();  
 return list.get(list.size() - 1);  
 }  
  
 ///////////////////////////  
 //Add new data to hash table  
 ///////////////////////////  
 final void add(Object object) {  
 int index = doHash(object);//% TableSize;  
 index = Math.abs(index) % TableSize;  
 //List was not created  
 if (Data[index] == null) {  
 List list = new ArrayList();  
 list.add(object);  
 this.Data[index] = list;  
 } else {  
 //Searching list. If object exists then return index  
 List list = (List) this.Data[index];  
 //Object has already been added before  
 if (list.contains(object))  
 return;  
 //Add object to the end of list  
 list.add(object);  
 CollisionCount++;  
 ListCollisions[index] ++;  
 }  
 }  
  
 public int getCountCollisions() {  
 return CollisionCount;  
 }  
  
 ////////////////////////////////////////  
 //Returns object from hash table or null  
 ////////////////////////////////////////  
 final public Object search(Object object) {  
 int index = Math.abs(doHash(object)) % TableSize;  
 if (Data[index] != null) {  
 //transform data to list  
 List list = (List) Data[index];  
 if (list.contains(object))  
 return list.get(list.indexOf(object));  
 else  
 return null;  
 }  
 return null;  
 }  
  
 ///////////////////////////////  
 //Delete object from hash table  
 ///////////////////////////////  
 final public void delete(Object object) {  
 if (search(object) != null) {  
 int index = Math.abs(doHash(object)) % TableSize;  
 List list = (List) this.Data[index];  
 if (list.size() == 1)  
 this.Data[index] = null; //Remove the whole list  
 else  
 list.remove(list.indexOf(object)); //Remove only element  
 }  
 }  
  
 //Abstract method for different hash algorithms  
 protected abstract Integer doHash(Object object);  
  
}

Файл Hash1

public class Hash1 extends AbstractHash {  
 private Integer SimpleNumber1;  
 private Integer SimpleNumber2;  
 private Integer Size;  
  
 /////////////  
 //Constructor  
 /////////////  
 Hash1(int size) throws Exception {  
 super(size);  
 this.Size = size;  
 SimpleNumber1 = super.randSimpleNumber();  
 Thread.sleep(1);  
 SimpleNumber2 = super.randSimpleNumber();  
 }  
  
 /////////  
 //Hashing  
 /////////  
 protected Integer doHash(Object object) {  
 String objectString = object.toString();  
 //Generate number from object  
 double numberFromObject = 0;  
 for (int i = 0; i < objectString.length(); i++) {  
 numberFromObject += objectString.charAt(i)\*i;  
 }  
 return (int) ((numberFromObject\*SimpleNumber1) % SimpleNumber2 ) \* SimpleNumber2;  
 }  
}

Файл Hash2

public class Hash2 extends AbstractHash {  
 private Integer SimpleNumber;  
  
 Hash2(int size) {  
 super(size);  
 SimpleNumber = super.randSimpleNumber();  
 }  
  
 protected Integer doHash(Object object) {  
 String objectString = object.toString();  
 //Generate number from object  
 double numberFromObject = 0;  
 for (int i = 0; i < objectString.length(); i++) {  
 numberFromObject += objectString.charAt(i)\*i;  
 }  
 //numberFromObject \*= SimpleNumber;  
 return (int) numberFromObject \* SimpleNumber;  
 }  
}

Файл Hash3

public class Hash3 extends AbstractHash {  
 Hash3(int size) throws Exception {  
 super(size);  
 }  
  
 protected Integer doHash(Object object) {  
 return object.hashCode();  
 }  
}

Файл Main

public class Main {  
 public static void main(String[] argv) throws Exception {  
  
 Double[] array = generateDomainInformation(1000);  
 Hash1 hash1 = new Hash1(array.length);  
 Hash2 hash2 = new Hash2(array.length);  
 Hash3 hash3 = new Hash3(array.length);  
 for(int i=0; i<array.length; i++) {  
 hash1.add(array[i]);  
 hash2.add(array[i]);  
 hash3.add(array[i]);  
 }  
 //Create output file  
 FileWriter file = new FileWriter("D:\\task5.csv");  
  
  
 Integer [] collisions1 = hash1.getCollisions();  
 Integer [] collisions2 = hash2.getCollisions();  
 Integer [] collisions3 = hash3.getCollisions();  
  
 for(int i = 0; i<array.length; i++) {  
 file.write("Hash1;" + i + ";" + collisions1[i] + "\n");  
 file.write("Hash2;" + i + ";" + collisions2[i] + "\n");  
 file.write("Hash3;" + i + ";" + collisions3[i] + "\n");  
  
 }  
 System.out.println("Count collision for hash1: " +hash1.getCollisionCount());  
 System.out.println("Count collision for hash2: " +hash2.getCollisionCount());  
 System.out.println("Count collision for hash3: " +hash3.getCollisionCount());  
  
 file.close();  
 }  
  
 //////////////////////////////////////////////////////////////////////////  
 //Generates a sequence with only one occurrence in expression "wordToFind"  
 /////////////////////////////////////////////////////////////////////////  
 private static Double[] generateDomainInformation (int countElements) throws Exception {  
 //Generate random  
 Random random = new Random(System.currentTimeMillis());  
 //To form result array  
 Double [] result = new Double[countElements];  
 for(int i=0; i<countElements; i++) {  
 Thread.sleep(1);  
 result[i] = 0 + random.nextDouble() \* (10000-0 + 1); //min + rand\*(max-min+1)  
 }  
 return result;  
 }  
  
}