**МИНОБРНАУКИ РОССИИ**

**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ**

**УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ**

**«ТУЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»**

Институт прикладной математики и компьютерных наук

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(тема курсовой работы)

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

к курсовой работе

по дисциплине

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(полное наименование учебной дисциплины)

|  |  |  |  |
| --- | --- | --- | --- |
| Студент гр. | \_\_\_\_\_\_\_\_\_\_\_\_\_\_ | \_\_\_\_\_\_\_\_\_\_\_\_\_\_ | \_\_\_\_\_\_\_\_\_\_\_\_\_\_ |
|  | (индекс группы) | (подпись и дата) | (инициалы и  фамилия) |

|  |  |  |  |
| --- | --- | --- | --- |
| Руководитель | \_\_\_\_\_\_\_\_\_\_\_\_\_\_ | \_\_\_\_\_\_\_\_\_\_\_\_\_\_ | \_\_\_\_\_\_\_\_\_\_\_\_\_\_ |
|  | (должность и  ученая степень) | (подпись и дата) | (инициалы и  фамилия) |

ТУЛА 2025

**СОДЕРЖАНИЕ**

[ВВЕДЕНИЕ 2](#_Toc200943987)

[ОБЩИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ 3](#_Toc200943988)

[ЗАДАЧА 1 8](#_Toc200943989)

[1. Постановка задачи 8](#_Toc200943990)

[2. Описание входной и выходной информации 9](#_Toc200943991)

[3. Алгоритм решения задачи 10](#_Toc200943992)

[4. Общие требования к программе 20](#_Toc200943993)

[5. Описание структуры программы для решения задачи 21](#_Toc200943994)

[6. Инструкция по эксплуатации программы 23](#_Toc200943995)

[7. Описание контрольного примера 24](#_Toc200943996)

[ЗАДАЧА 2 27](#_Toc200943997)

[1. Постановка задачи 27](#_Toc200943998)

[2. Описание входной и выходной информации 28](#_Toc200943999)

[3. Алгоритм решения задачи 29](#_Toc200944000)

[4. Общие требования к программе 36](#_Toc200944001)

[5. Описание структуры программы для решения задачи 37](#_Toc200944002)

[6. Инструкция по эксплуатации программы 39](#_Toc200944003)

[7. Описание контрольного примера 40](#_Toc200944004)

[ЗАКЛЮЧЕНИЕ 42](#_Toc200944005)

[Библиографический список 43](#_Toc200944006)

[Приложения 44](#_Toc200944007)

# **ВВЕДЕНИЕ**

В программировании структуры данных необходимы для эффективной работы с информацией. От того, как организованы данные, напрямую зависит, насколько быстро программа будет искать элементы, добавлять новые или удалять старые. Особенно это становится критичным, когда объём данных велик или доступ к ним требуется по сложным условиям.

Хеш-таблицы и двоичные кучи представляют собой две очень популярные и полезные структуры данных. Двоичные кучи незаменимы, когда требуется быстро находить и удалять элемент с наивысшим приоритетом (например, с максимальным значением). Хеш-таблицы, в свою очередь, являются лидерами по скорости, если необходимо искать, добавлять или удалять данные по уникальному ключу.

Целью данной курсовой работы является детальное исследование хеш-таблиц и двоичных куч, анализ их внутреннего устройства, алгоритмов работы и областей применения. В работе будут рассмотрены их сильные и слабые стороны, а также проведено сравнение их производительности для различных задач, таких как реализация словарей, кеширование данных и организация очередей с приоритетом.

# **ОБЩИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ**

1. Хеш-таблица

Хеш-таблица - это структура данных, реализующая интерфейс ассоциативного массива, которая позволяет хранить пары «ключ-значение». Её ключевая особенность заключается в том, что по ключу можно получить доступ к значению за время, близкое к константному. Для этого используется хеш-функция, которая преобразует ключ в индекс в базовом массиве, куда и помещается элемент.

В рамках данной работы рассматривается хеш-таблица для хранения пар, где ключ - это строка, а значение - целое число. Такой подход часто используется для подсчета частоты слов в тексте, реализации телефонных справочников или создания ассоциативных массивов.

* 1. Хеш-функция и вычисление индекса

Центральным элементом хеш-таблицы является хеш-функция. Она принимает на вход ключ и преобразует его в числовое значение (хеш), на основе которого вычисляется индекс ячейки в массиве. В реализации будет использована стандартная хеш-функция из C++ - std::hash. Она поддерживает множество типов данных, включая строки, что избавляет от необходимости разрабатывать и тестировать собственную функцию.

Чтобы получить индекс ячейки, вычисляется остаток от деления хеш-значения на размер таблицы. Правильный подбор размера таблицы способствует равномерному распределению элементов и снижает вероятность коллизий.

* 1. Коллизии и метод цепочек

Коллизия - это ситуация, когда два разных ключа после хеширования получают один и тот же индекс. Полностью избежать коллизий практически невозможно, так как количество потенциальных ключей может значительно превышать размер таблицы. В данной работе для разрешения коллизий применяется метод цепочек (chaining).

Суть метода проста: каждая ячейка таблицы является не одним элементом, а указателем на начало связанного списка (или другого контейнера). Если несколько элементов отображаются в одну и ту же ячейку, они последовательно добавляются в этот список. Таким образом, добавление нового элемента сводится к его вставке в список соответствующей ячейки, а поиск или удаление - к перебору элементов в этом списке.

* 1. Анализ производительности

Основное преимущество хеш-таблицы - это её скорость. В среднем операции поиска, вставки и удаления выполняются за константное время - O(1). Это означает, что время выполнения операции практически не зависит от количества элементов в таблице.

Однако это средний показатель. В худшем случае, при неудачной хеш-функции или специфическом наборе данных, все элементы могут попасть в одну цепочку. Тогда производительность операций деградирует до линейной - O(n), где n - число элементов. Для предотвращения таких ситуаций используется рехеширование: при превышении определённого коэффициента заполнения (load factor) таблица увеличивается в размере, и все элементы перераспределяются заново.

1. Двоичная куча (Max-Heap)

Двоичная куча - это древовидная структура данных, которая для удобства хранится в виде массива. В данной работе рассматривается максимальная двоичная куча (max-heap). Её основное свойство заключается в том, что значение в любом узле не меньше, чем значения в его дочерних узлах. Благодаря этому свойству самый большой элемент всегда находится на вершине кучи - в корне дерева.

* 1. Свойства и представление в массиве

Двоичную кучу удобно хранить в обычном динамическом массиве, где каждый элемент массива соответствует узлу дерева. Индексы родителя и потомков для любого узла с индексом i легко вычисляются по формулам (применяется целочисленное деление):

* Левый потомок: 2 × i + 1
* Правый потомок: 2 × i + 2
* Родитель: (i - 1) ÷ 2
  1. Основные операции

1. Вставка элемента (Insert): Новый элемент добавляется в конец массива. Затем выполняется операция «просеивания вверх» (sift-up): элемент сравнивается со своим родителем и, если он больше, меняется с ним местами. Процесс повторяется, пока элемент не займёт корректную позицию, восстанавливая свойство кучи.

2. Извлечение максимума (Extract-Max): Максимальный элемент (корень) сохраняется для возврата. На его место помещается последний элемент из массива, после чего размер кучи уменьшается. Для восстановления свойства кучи запускается операция «просеивания вниз» (sift-down) для нового корня: он сравнивается со своими потомками и меняется местами с наибольшим из них, если он меньше. Процесс продолжается, пока элемент не "опустится" на своё место.

3. Просмотр максимума (Peek-Max): Самая быстрая операция. Максимальный элемент всегда находится в корне (первый элемент массива), поэтому он просто возвращается без каких-либо изменений в структуре.

* 1. Анализ производительности и применение

Производительность основных операций двоичной кучи зависит от её высоты, которая для n элементов составляет O(log n).

* Вставка: O(log n)
* Извлечение максимального элемента: O(log n)
* Получение максимального элемента: O(1)

Двоичные кучи являются идеальным выбором для реализации очередей с приоритетом. Они также лежат в основе эффективного алгоритма сортировки (Heapsort), который сортирует массив на месте за время O(n log n), и применяются в алгоритмах на графах (например, в алгоритме Дейкстры).

1. Сравнительный анализ

Хеш-таблица и двоичная куча - эффективные структуры данных, но предназначенные для решения разных задач.

* Хеш-таблица обеспечивает сверхбыстрый доступ по ключу (O(1) в среднем). Это лучший выбор, если требуется часто искать, добавлять или удалять элементы по уникальному идентификатору. Однако она не поддерживает упорядоченное хранение данных.
* Двоичная куча создана для работы с приоритетами. Она позволяет быстро находить и извлекать самый "важный" (максимальный или минимальный) элемент за логарифмическое время (O(log n)), а просматривать его - мгновенно (O(1)).

## **ЗАДАЧА 1**

## **1. Постановка задачи**

Сделать программу-демонстратор, которая позволит пользователю взаимодействовать с двоичной кучей max-heap и выполнять с ней основные действия, а именно вставку элемента, извлечение максимального элемента, просмотр макс. элемента.

Разобраться с двоичными кучами: понять их основное свойство, как делаются вставка и извлечение элемента, как построить кучу.

Написать на C++ классы для двоичной кучи.

## **2. Описание входной и выходной информации**

Программа принимает на вход команды пользователя: строки (insert, extractMax, peekMax, display, size). Команда insert принимает значения для добавления в кучу - число.

На экран выводятся:

* Текст о том, что команда выполнена успешно или об ошибке.
* Следующие команды выводят:
  + insert: сообщение об успешном добавлении.
  + extractMax: извлечённое макс. Значение.
  + peekMax: значение макс. элемента.
  + size: количество элементов.
  + display: содержимое кучи.

## **3. Алгоритм решения задачи**

В этой работе делаем максимальную кучу, где родитель всегда не меньше своих детей. Хранить будем в массиве std::vector. Индексы родителя и потомков вычисляются по формулам, описанным в теоретической части.

Для работы с двоичной кучей реализуем несколько основных методов.

Метод insert: добавляет новый элемент в кучу. Сначала он добавляет элемент в конец массива. Затем, чтобы восстановить свойство максимальной кучи, которое могло быть нарушено, он выполняет операцию siftUp (просеивание вверх) для добавленного элемента, меняя его местами с родителями, пока он не встанет на свое место.

Изображение выглядит как текст, диаграмма, снимок экрана, дизайн

Содержимое, созданное искусственным интеллектом, может быть неверным.

Рисунок 1 – Блок-схема функции insert для двоичной кучи

Метод extractMax: забирает самый большой элемент (он всегда в корне, индекс 0). Сначала запоминает значение корня. Потом ставит на его место последний элемент кучи и удаляет этот последний элемент. Чтобы восстановить свойство кучи, запускает siftDown (просеивание вниз) для нового корня, меняя его местами с наибольшим из потомков, пока он не опустится на правильную позицию. Возвращает сохраненное максимальное значение. Если куча пуста, программа выведет сообщение об этом как об ошибке, а extractMax вернёт -1.

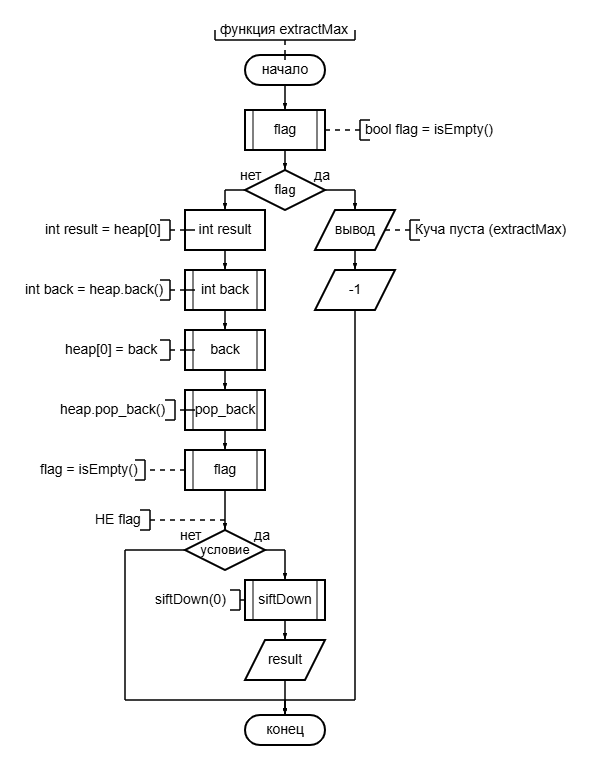


Рисунок 2 – Блок-схема функции extractMax

Метод peekMax: просто посмотреть максимальный элемент, не удаляя его. Возвращает значение из корня (индекс 0). Если куча пуста, программа выведет сообщение об этом как об ошибке, а peekMax вернёт -1.

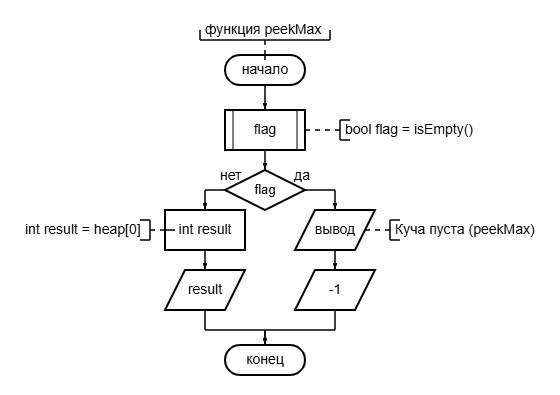


Рисунок 3 – Блок-схема функции peekMax

Метод siftUp нужен после вставки. Он сравнивает элемент с родителем. Если элемент больше родителя (нарушение свойства max-heap), меняет их местами и повторяет то же самое для новой позиции элемента, двигаясь вверх.

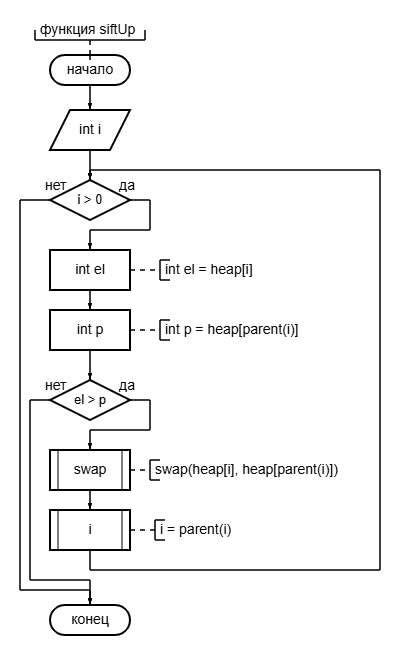


Рисунок 4 – Блок-схема функции siftUp

Метод siftDown нужен после извлечения максимума. Он сравнивает элемент с потомками. Если элемент меньше какого-то из потомков (нарушение свойства max-heap), он меняет его местами с наибольшим из потомков и повторяет процесс для новой позиции, двигаясь вниз.

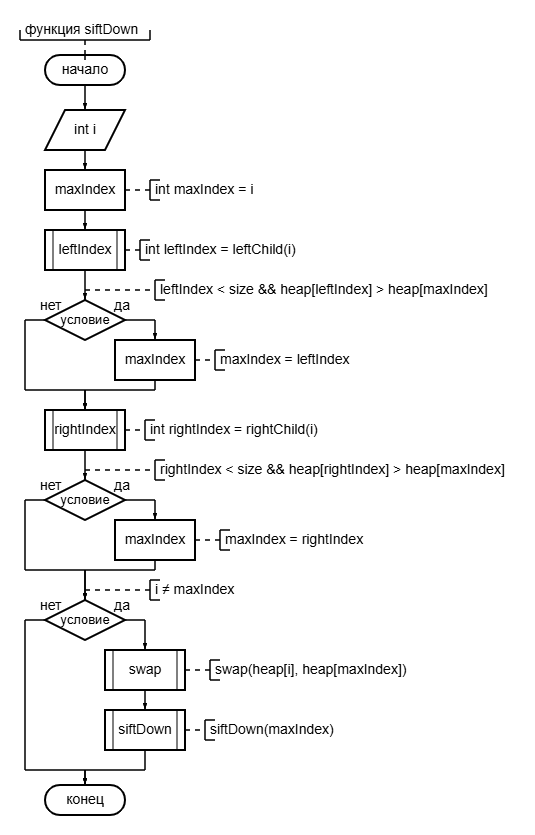


Рисунок 5 – Блок-схема функции siftDown

Функция main запускает runBinaryHeap уже в которой происходит диалог с пользователем, создание кучи и вызов её методов по командам пользователя.

Сначала создаётся объект BinaryHeap. После этого запускается внутренний цикл, уже для работы с ней. Программа просит пользователя ввести команду.

Если команде нужны данные, программа их запрашивает. Потом вызывается нужный метод у созданного объекта. Результат выводится на экран. Этот внутренний цикл крутится, пока пользователь не завершит процесс программы.

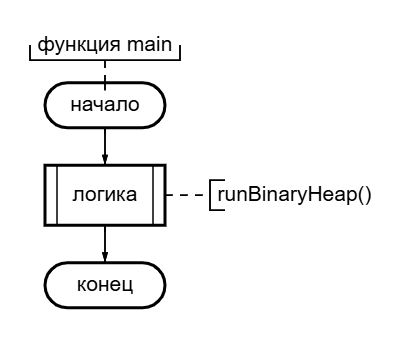


Рисунок 6 – Блок-схема функции main

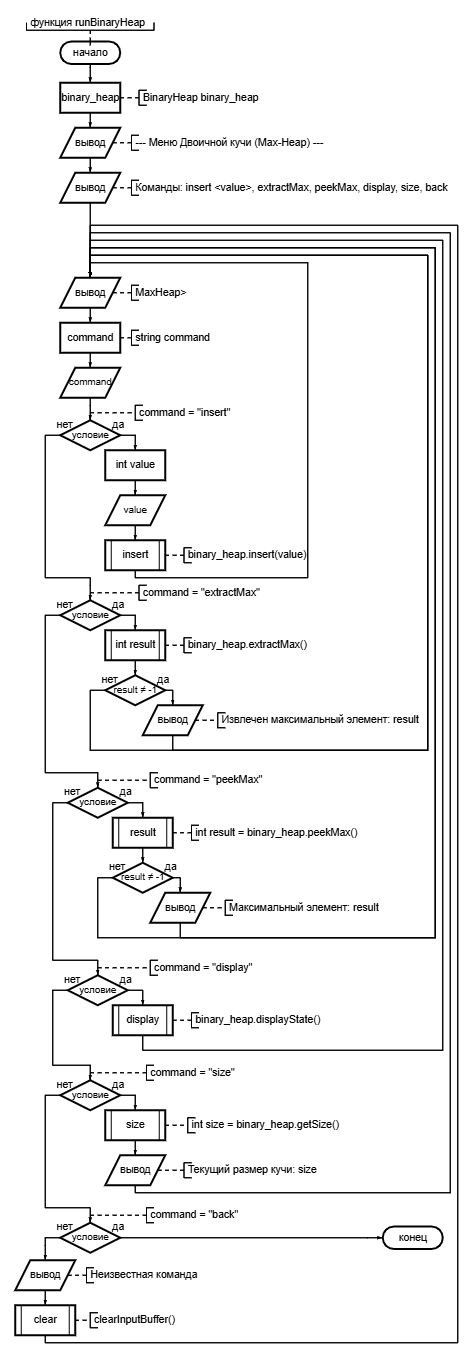


Рисунок 7 – Блок-схема функции runBinaryHeap

Блок схемы вспомогательных функций:

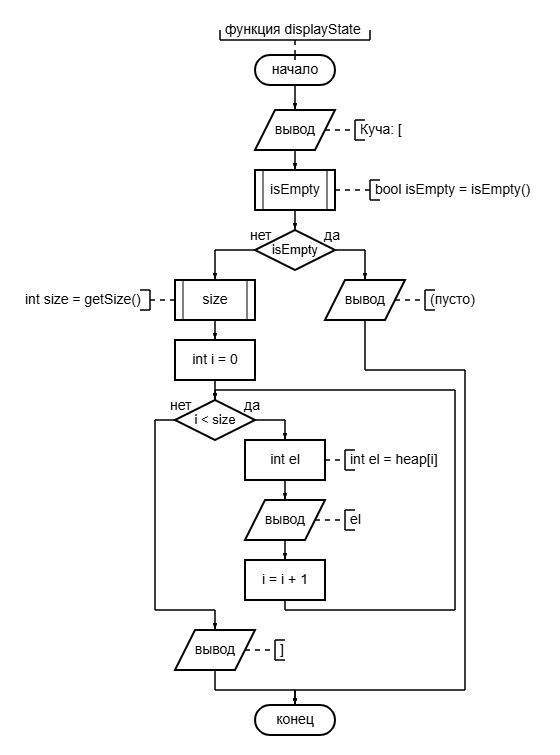


Рисунок 8 – Блок-схема функции displayState

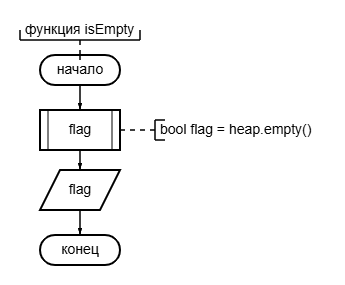


Рисунок 9 – Блок-схема функции isEmpty

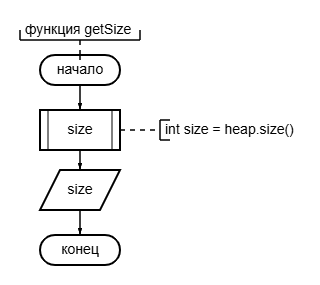


Рисунок 10 – Блок-схема функции getSize

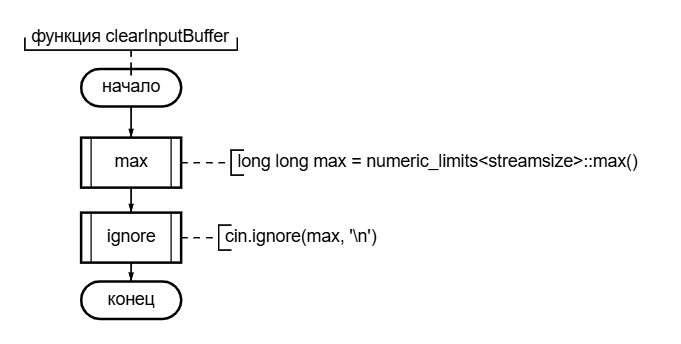


Рисунок 11 – Блок-схема функции clearInputBuffer

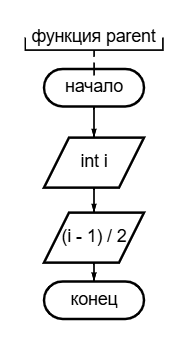


Рисунок 12 – Блок-схема функции parent

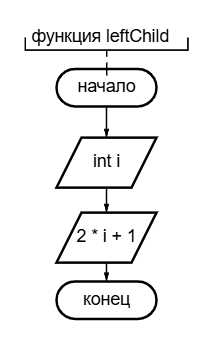


Рисунок 13– Блок-схема функции leftChild

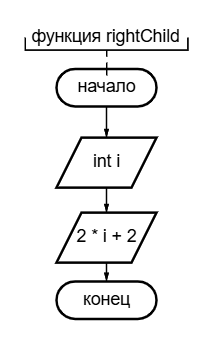


Рисунок 14 – Блок-схема функции rightChild

## **4. Общие требования к программе**

Что должна уметь программа:

* Правильно реализовывать двоичную кучу.
* Выполнять основные операции для бинарной кучи через команды пользователя.
* обработка ошибочных ситуаций.

Язык программирования C++ был выбран для данной курсовой работы по ряду причин.

Скорость. C++ позволяет писать быстрые программы. Это важно, когда мы работаем со структурами данных и хотим оценить их производительность. Он дает контроль над памятью и близок к "железу".

Гибкость. На C++ можно легко расширять программу. Мы сделали структуры данных в виде классов – это значит, что их можно будет потом доработать, добавить новые методы, если понадобится.

Кроссплатформенность. Код на C++ можно собрать и запустить на разных системах (Windows, Linux, macOS) почти без изменений. Это удобно.

Готовые инструменты (STL). В C++ есть стандартная библиотека с кучей полезных вещей: готовые контейнеры (vector, list), алгоритмы (swap), хеш-функция (std::hash). Это сильно упрощает разработку.

Популярность и поддержка. C++ – известный язык, по нему много документации, примеров и готовых решений в интернете. Если что-то непонятно, легко найти помощь.

Это значит, что легко найти информацию, примеры и помощь, если что-то пойдет не так при изучении или разработке.

## **5. Описание структуры программы для решения задачи**

Из чего состоит программа:

1. Заголовочные файлы и пространства имен:
   * #include: Включение необходимых библиотек C++:
     + <iostream>: для ввода/вывода данных в консоль.
     + <vector>: для реализации основы кучи и массива списков в хеш-таблице.
     + <string>: для работы со строковыми ключами/значениями.
     + <limits>: для очистки буфера ввода.
     + <algorithm>: содержит std::swap для обмена элементов в двоичной куче.
   * using namespace std: упрощает код, позволяя использовать элементы пространства имен std без необходимости писать std:: перед каждым из них.
2. Класс BinaryHeap:

Реализует двоичную кучу максимальных элементов.

Приватные члены:

* heap: std::vector<T> - вектор, хранящий элементы кучи.
* parent(i), leftChild(i), rightChild(i): Вспомогательные методы для вычисления индексов родителя и потомков.
* siftUp(i): Реализует операцию просеивания вверх для восстановления свойства max-heap после вставки.
* siftDown(i): Реализует операцию просеивания вниз для восстановления свойства max-heap после извлечения максимума.

Публичные методы:

* Конструктор: Создает пустую кучу.
* isEmpty(): Проверяет, пуста ли куча.
* insert(value): Добавляет новый элемент в кучу.
* peekMax(): Возвращает максимальный элемент без его удаления. Генерирует исключение, если куча пуста.
* extractMax(): Удаляет и возвращает максимальный элемент. Генерирует исключение, если куча пуста.
* displayState(): Выводит текущее содержимое кучи (в виде массива) в консоль.
* getSize(): Возвращает текущее количество элементов в куче.

1. Вспомогательные функции:

* clearInputBuffer(): Очищает буфер стандартного ввода std::cin для предотвращения ошибок при последующем вводе после некорректного ввода.
* runBinaryHeap(): Реализует цикл пользовательского интерфейса для работы с двоичной кучей.

1. Функция main:

Точка старта программы. Запускает runBinaryHeap() для работы с бинарной кучей.

## **6. Инструкция по эксплуатации программы**

1. Запустите программу.
2. Вводите команды для взаимодействия со структурой данных (“insert значение”, “extractMax”, “peekMax”, “display”, “size”).
3. Программа выведет результат каждой команды.
4. Для выхода используйте команду “back”

## **7. Описание контрольного примера**

Для проверки корректности работы программы запустим её и выполним последовательность команд, которые протестируют все основные функции двоичной кучи: добавление элементов, извлечение максимального, просмотр максимального и отображение состояния. Этот сценарий позволит оценить, как программа справляется с построением кучи, поддержанием её свойств и обработкой граничных случаев.

Вначале мы последовательно добавим в кучу пять чисел разной величины: 10, 20, 5, 30, 15. После этого мы запросим текущее состояние кучи и её размер, чтобы убедиться в корректности операции вставки. Затем мы проверим работу функций просмотра и извлечения максимального элемента. Несколько последовательных извлечений продемонстрируют, что куча правильно перестраивается после удаления корня. Наконец, мы попытаемся извлечь элемент из уже пустой кучи, чтобы протестировать обработку ошибочной ситуации. Вывод программы представлен на рисунке 15.



Рисунок 15 – Вывод программы контрольного примера

Результаты подтверждают полную корректность работы программы. После вставки элементов структура была правильно сформирована: максимальный элемент 30 занял корневую позицию, а свойство max-heap было соблюдено для всех узлов.

Операции peekMax и extractMax работали ожидаемо. Каждое извлечение приводило к корректной перестройке кучи. Программа успешно обработала попытку извлечения из пустой структуры выводом сообщения об ошибке. Таким образом, контрольный пример доказывает, что программа полностью соответствует заявленным требованиям, корректно реализуя все операции.

## **ЗАДАЧА 2**

## **1. Постановка задачи**

В работе необходимо выполнить следующие задачи:

* Создать программу телефонный справочник, который загружает пары "имя -номер телефона" из текстового файла. Программа ожидает ввод имени с клавиатуры и выводит соответствующий номер телефона, если имя найдено, либо сообщение "Имя не найдено", если такой записи нет.
* Изучить теорию по хеш-таблицам: как работают хеш-функции, какие есть способы борьбы с коллизиями (в этой работе используем метод цепочек).
* Написать на C++ класс для хеш таблицы(.

## **2. Описание входной и выходной информации**

Входные данные:

* Текстовый файл phonebook.txt в той же директории, что и программа. Каждая строка файла должна содержать имя и номер телефона, разделенные пробелом (имена и номера из одного слова).
* Имя человека, номер телефона которого, необходимо найти в справочнике.
* Команда exit для завершения работы программы

Выходные данные:

При запуске выводится текст об успешной или нет загрузке файла phonebook.txt, и если да, количество считанных записей. После выводится приглашение к вводу имени человека “> “. Если программа найдёт в файле номер телефона человека с этим именем, то выведен его иначе покажет ошибку "Имя не найдено". Для закрытия программы вводится exit.

## **3. Алгоритм решения задачи**

Решение задачи включает реализацию класса хеш-таблицы и основной программы (main), которая реализует функционал телефонного справочника.

Хеш-таблица реализуется с использованием метода цепочек. В качестве основы используется массив (std::vector), элементами которого являются списки (std::list), хранящие пары "ключ-значение" (обе строки). Для преобразования ключа (имени) в индекс используется стандартная функция std::hash<std::string>. Индекс вычисляется как остаток от деления хеш-кода на размер таблицы. При коллизиях новая пара добавляется в список, соответствующий вычисленному индексу.

Основные методы класса:

Метод insert: добавляет пару (имя, номер). Вычисляет индекс по имени. Проверяет наличие имени в соответствующей цепочке. Если имя найдено, обновляет номер телефона. Если нет — добавляет новую запись в список. Для загрузки из файла вывод сообщений в этом методе отключен.

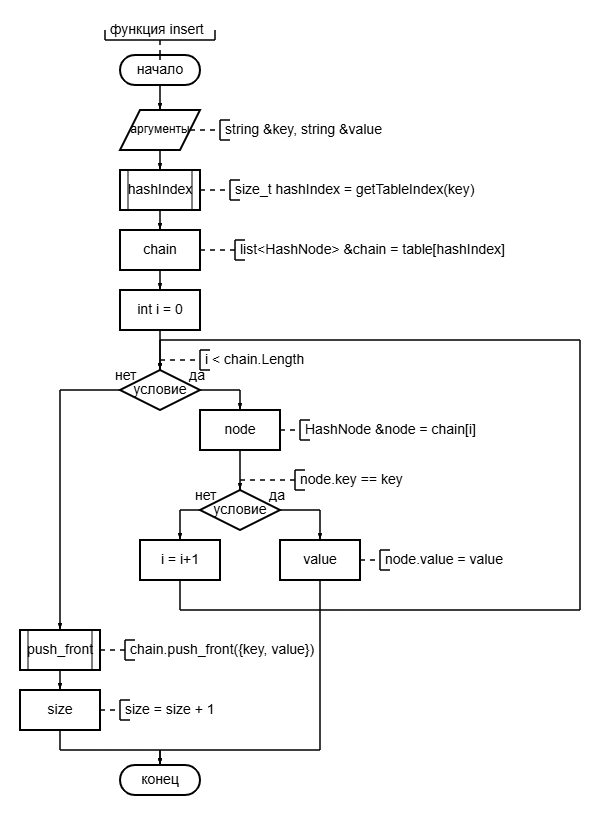


Рисунок 16 – Блок-схема функции insert для хеш-таблицы

Метод search: ищет номер по имени. Вычисляет индекс. Перебирает элементы цепочки. При нахождении совпадения по имени возвращает true и соответствующий номер. Если имя не найдено, возвращает false.

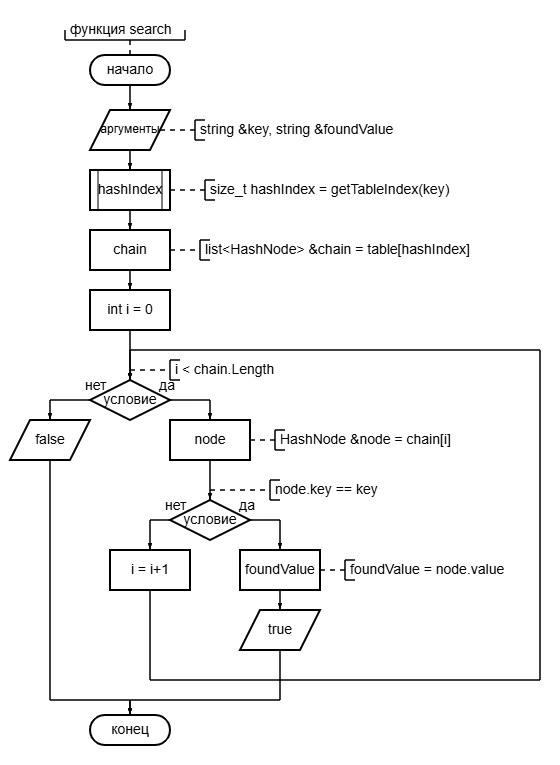


Рисунок 17 – Блок-схема функции search для хеш-таблицы

Функция main создаёт телефонную книгу как HashTable, по средствам функции loadFromFile загружает в неё данные из файла phonebook.txt. Далее в main описана логика взаимодействия с пользователем. Он может узнать номер телефона человека введя его имя или выйти введя exit.

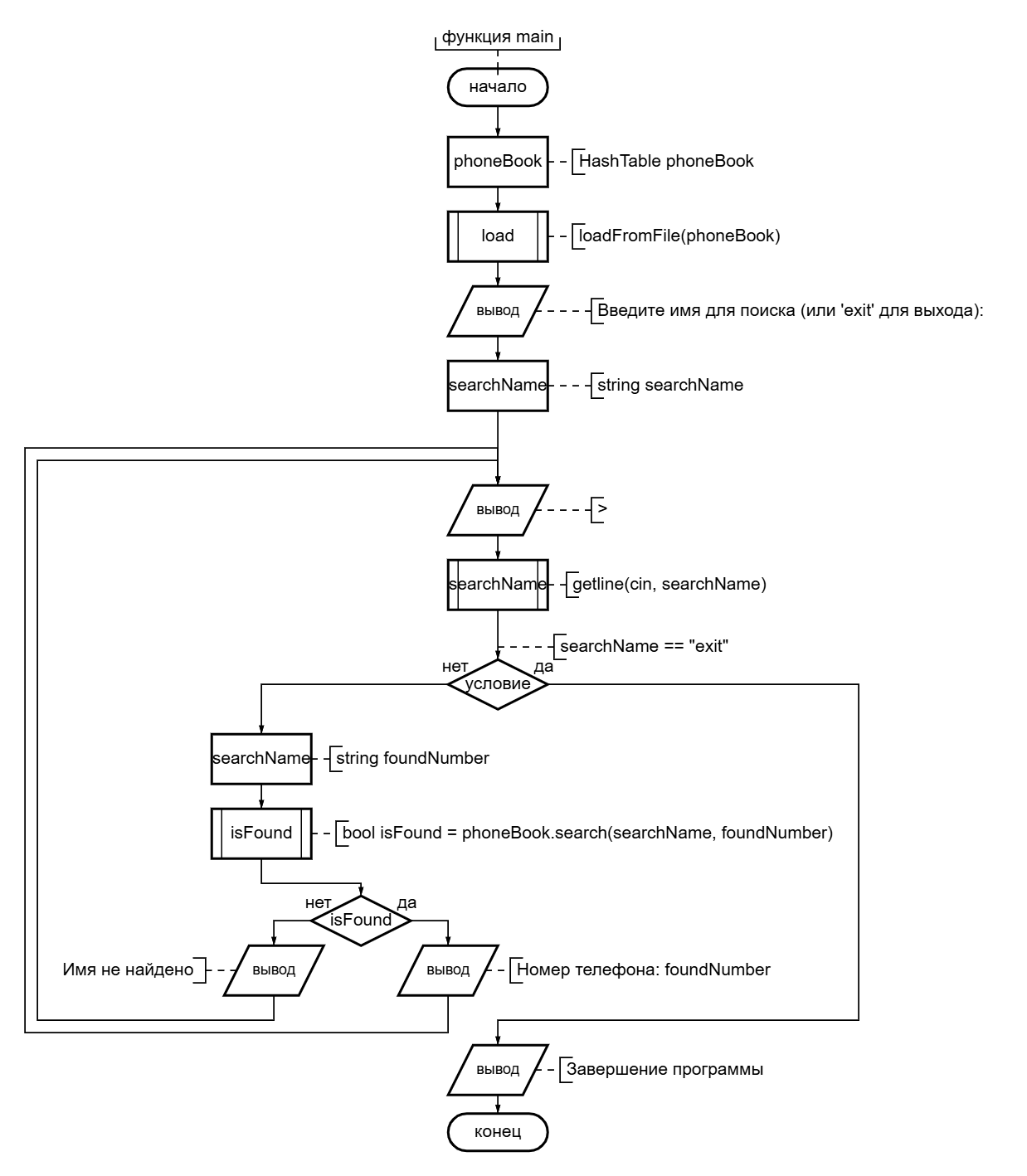


Рисунок 18 – Блок-схема функции main для задания 2

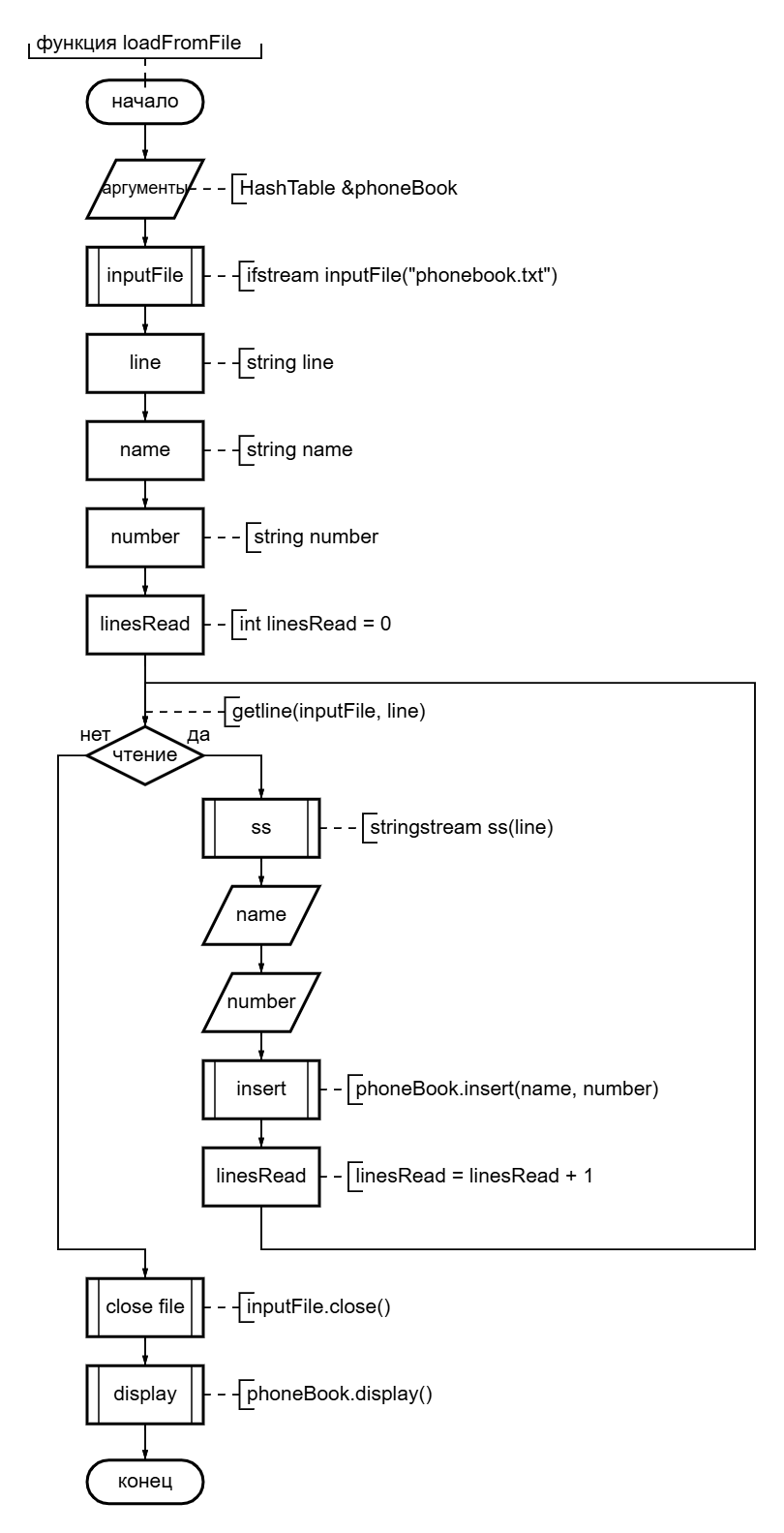


Рисунок 19 – Блок-схема функции loadFromFile

Блок схемы других, не менее важных функций:

Изображение выглядит как текст, снимок экрана, диаграмма, Шрифт

Содержимое, созданное искусственным интеллектом, может быть неверным.

Рисунок 20 – Блок-схема функции getTableIndex для хеш-таблицы

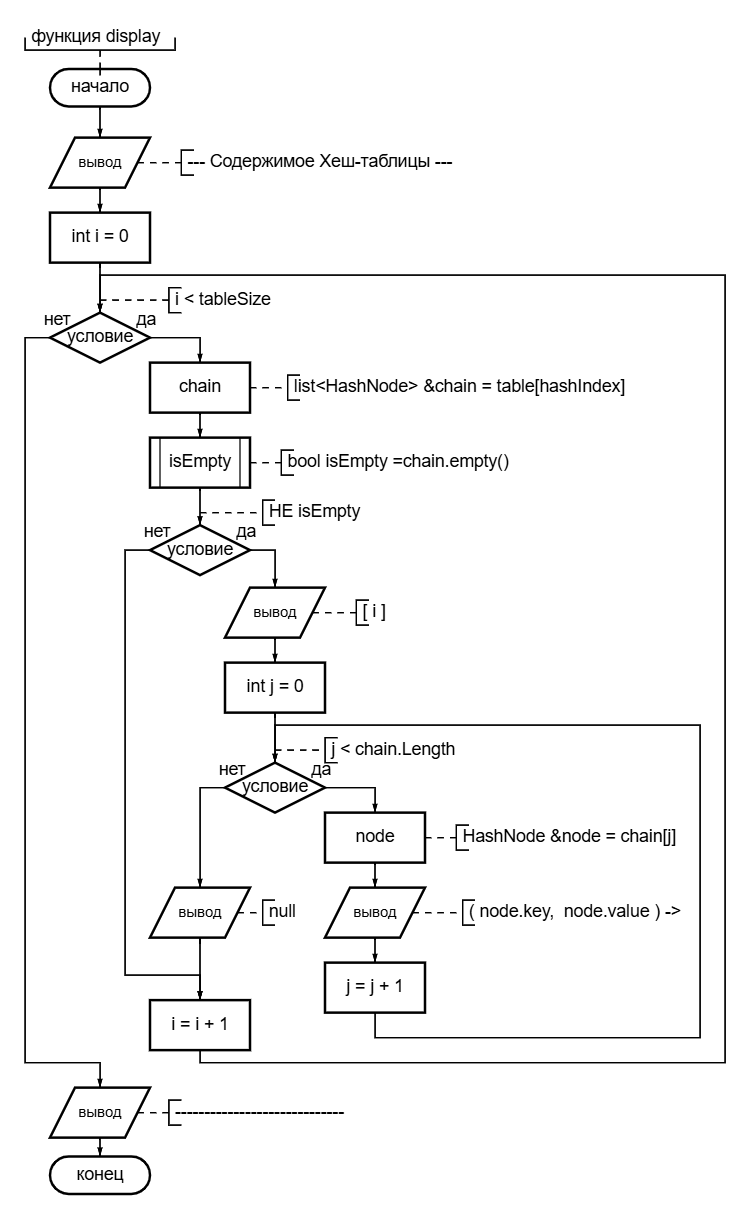


Рисунок 21 – Блок-схема функции display для хеш-таблицы

## **4. Общие требования к программе**

Программа телефонного справочника должна соответствовать следующим требованиям:

* Использовать хеш-таблицу с методом цепочек для хранения данных справочника.
* Успешно загружать данные из файла phonebook.txt.
* Осуществлять поиск номера телефона по имени, введенному пользователем.
* Выводить результат поиска или сообщение об отсутствии записи.
* Обрабатывать возможные ошибки (проблемы с файлом, некорректный формат данных)

Язык программирования C++ был выбран для данной курсовой работы по ряду причин.

Скорость. C++ позволяет писать быстрые программы. Это важно, когда мы работаем со структурами данных и хотим оценить их производительность. Он дает контроль над памятью и близок к "железу".

Гибкость. На C++ можно легко расширять программу. Мы сделали структуры данных в виде классов – это значит, что их можно будет потом доработать, добавить новые методы, если понадобится.

Кроссплатформенность. Код на C++ можно собрать и запустить на разных системах (Windows, Linux, macOS) почти без изменений. Это удобно.

Готовые инструменты (STL). В C++ есть стандартная библиотека с кучей полезных вещей: готовые контейнеры (vector, list), алгоритмы (swap), хеш-функция (std::hash). Это сильно упрощает разработку.

Популярность и поддержка. C++ – известный язык, по нему много документации, примеров и готовых решений в интернете. Если что-то непонятно, легко найти помощь.

Это значит, что легко найти информацию, примеры и помощь, если что-то пойдет не так при изучении или разработке.

## **5. Описание структуры программы для решения задачи**

Из чего состоит программа:

1. Заголовочные файлы и пространства имен:
   * #include: Включение необходимых библиотек C++:
     + <iostream>: для ввода/вывода данных в консоль.
     + <vector>: для реализации основы кучи и массива списков в хеш-таблице.
     + <list>: для реализации цепочек в хеш-таблице.
     + <string>: для работы со строковыми ключами/значениями.
     + <fstream>: для чтения файла.
     + < sstream>: для комфортного чтения строк из файла.
   * using namespace std;: упрощает код, позволяя использовать элементы пространства имен std без необходимости писать std:: перед каждым из них.
2. Класс HashTable:

Реализует хеш-таблицу (ключ - std::string, значение - std::string) с методом цепочек.

Содержит внутреннюю, вектор списков table, переменные для хранения размеров, объект std::hash<std::string>.

Приватные члены:

* структура HashNode – структура ключ - значение.
* vector<list<HashNode> > table: хранит данные в хеш-таблице
* getTableIndex(key): получение индекса для данного ключа.

Публичные методы:

* Конструктор: Создает пустую хеш-таблицу размером 101.
* insert(key, value): добавляет новый элемент.
* search(key, foundValue): поиск элемента по ключу в хеш-таблице. Возвращает true, если найдено иначе ложь.
* display(): Выводит текущее содержимое в консоль.
* getSize(): Возвращает текущее количество элементов.

1. Функция main:

Точка старта программы. Создает объект HashTable. Запускает основной цикл программы.

1. Функция loadFromFile

Служит для загрузки в телефонный справочник данных из файла phonebook.txt.

## **6. Инструкция по эксплуатации программы**

1. Запустите программу.
2. Создайте файл phonebook.txt. Он должен находится в той же папке, где и исполняемый файл программы. В файле каждая строка должна содержать имя и номер, разделенные пробелом (например: **Petr 987-65-43**)
3. Программа сообщит о результате загрузки данных из файла.
4. Введите имя абонента, чей номер вы хотите найти, и нажмите Enter.
5. Программа выведет либо номер телефона, либо сообщение "Имя не найдено".
6. Для завершения работы введите exit и нажмите Enter.

## **7. Описание контрольного примера**

Для проверки корректности работы программы-телефонного справочника необходимо провести контрольный запуск. Этот тест продемонстрирует ключевые аспекты работы программы: успешную загрузку данных из файла, правильное построение хеш-таблицы, а также корректную обработку запросов пользователя на поиск.

Сначала мы создадим текстовый файл phonebook.txt со списком контактов. Этот файл будет содержать пять записей, которые программа должна будет загрузить и разместить в хеш-таблице.

Содержимое файла phonebook.txt на рисунке 22.

Изображение выглядит как текст, снимок экрана, Шрифт, число

Содержимое, созданное искусственным интеллектом, может быть неверным.

Рисунок 22 – Содержимое файла phonebook.txt

После запуска программа должна прочитать этот файл, загрузить все пять записей в хеш-таблицу и вывести её содержимое для наглядности. Затем мы выполним несколько поисковых запросов: найдём два существующих контакта, попробуем найти один несуществующий и, наконец, завершим работу программы командой exit.

Вывод программы представлен на рисунке 23.

Изображение выглядит как текст, снимок экрана, Шрифт, число

Содержимое, созданное искусственным интеллектом, может быть неверным.

Рисунок 23 – Содержимое файла phonebook.txt

Результаты контрольного примера подтверждают полную корректность работы программы. Программа успешно загрузила 5 записей из файла, о чём свидетельствует стартовое сообщение. Вывод содержимого хеш-таблицы наглядно показывает, что все записи были распределены по разным ячейкам в соответствии с их хеш-кодами. В данном наборе данных коллизий не произошло, и каждая запись сформировала отдельную цепочку, что является одним из возможных сценариев работы хеш-таблицы.

Поисковые запросы были обработаны верно. Программа успешно нашла и вывела номера для существующих имён "Petr" и "Anna". Попытка найти несуществующее имя "Nikolay" привела к правильному результату — выводу сообщения "Имя не найдено", что демонстрирует корректную работу алгоритма поиска. Команда exit завершила выполнение программы как было задумано.

Таким образом, контрольный тест доказывает, что программа надёжно выполняет все поставленные задачи: читает данные, строит хеш-таблицу и эффективно осуществляет поиск.

# **ЗАКЛЮЧЕНИЕ**

В этой курсовой были написаны на C++ две программы: хеш-таблица (где коллизии решаются методом цепочек) и двоичную кучу (а именно max-heap). Обе эти структуры – полезные инструменты, которые помогают эффективно хранить и обрабатывать данные в разных ситуациях. Понять, как они работают, важно, чтобы правильно выбирать структуру под свою задачу.

Главное достоинство хеш-таблицы – скорость доступа по ключу. В среднем, если все хорошо поиск, вставка и удаление работают почти мгновенно – за O(1)O(1). Но если не повезет и все ключи попадут в одну ячейку, придется пройтись по длинному списку, и скорость упадет до O(n)O(n). Хеш-таблицы – идеальный вариант для словарей, кешей и везде, где нужен быстрый доступ по уникальному ключу. Порядок элементов они не хранят.

Двоичная куча max-heap. Это структура, похожая на дерево, где наверху всегда самый большой элемент. Вставка и извлечение этого максимального элемента требуют немного работы (просеивания вверх или вниз), но все равно делаются быстро – за O(logn). Просто посмотреть на максимум можно мгновенно – за O(1). А вот искать какой-то произвольный элемент в куче долго – придется перебирать все O(n). Эта структура данных отлично подходит для задач с приоритетами и для сортировки HeapSort.

Выбор зависит от целей: нужен быстрый поиск по ключу –хеш-таблица. Нужен быстрый доступ к максимальному элементу – используем двоичную кучу.к

## **Библиографический список**

Книги:

1. "Введение в алгоритмы", Томас Х. Кормен, Чарльз Э. Лейзерсон, Рональд Л. Ривест, Клиффорд Штайн.

2. "Алгоритмы: Построение и анализ", С. С. Скворцов.

3. "Алгоритмы на C++" Роберт Седжвик

Статьи:

1. GeeksforGeeks: "Hashing | Set 2 (Separate Chaining)". Статья, описывающая метод цепочек для разрешения коллизий в хеш-таблицах. Доступно по ссылке: https://www.geeksforgeeks.org/hashing-set-2-separate-chaining/

Онлайн-ресурсы:

1. Википедия: "Хеш-таблица". Статья на Википедии, описывающая структуру данных, хеш-функции и методы разрешения коллизий. Доступно по ссылке: <https://ru.wikipedia.org/wiki/Хеш-таблица>

2. Википедия: "Двоичная куча". Статья на Википедии, описывающая свойства, операции (включая Max-Heap) и применение двоичных куч. Доступно по ссылке: <https://ru.wikipedia.org/wiki/Двоичная_куча>

3. cppreference.com: Справочник по языку C++ и стандартной библиотеке. Полезен для уточнения деталей реализации контейнеров и функций STL. Доступно по ссылке: <https://en.cppreference.com/w/>

## **Приложения**

Листинг программы для задания 1:

#include <iostream>

#include <vector>

#include <limits>

using namespace std;

class BinaryHeap {

private:

vector<int> heap{};

static int parent(int i) { return (i - 1) / 2; }

static int leftChild(int i) { return 2 \* i + 1; }

static int rightChild(int i) { return 2 \* i + 2; }

// Просеивание вверх

void siftUp(int i) {

// Пока не дошли до корня и текущий элемент БОЛЬШЕ родителя

while (i > 0) {

int el = heap[i];

int p = heap[parent(i)];

if (el > p) {

swap(heap[i], heap[parent(i)]);

i = parent(i); // Переходим на позицию родителя}

} else break;

}

}

// Просеивание вниз

void siftDown(int i) {

int maxIndex = i; // Индекс наибольшего элемента (сначала текущий)

int leftIndex = leftChild(i);

int size = heap.size();

// Если левый потомок существует и он БОЛЬШЕ текущего максимума

if (leftIndex < size && heap[leftIndex] > heap[maxIndex]) maxIndex = leftIndex;

int rightIndex = rightChild(i);

// Если правый потомок существует и он БОЛЬШЕ текущего максимума

if (rightIndex < size && heap[rightIndex] > heap[maxIndex]) maxIndex = rightIndex;

// Если наибольший элемент не текущий, меняем и продолжаем просеивание

if (i != maxIndex) {

swap(heap[i], heap[maxIndex]);

siftDown(maxIndex); // Рекурсивно для новой позиции

}

}

public:

BinaryHeap() = default;

int getSize() {

int size = heap.size();

return size;

}

bool isEmpty() {

bool isEmpty = heap.empty();

return isEmpty;

}

void insert(int value) {

heap.push\_back(value); // Добавляем в конец

siftUp(heap.size() - 1); // Восстанавливаем свойство кучи просеиванием вверх

cout << "Элемент " << value << " добавлен. " << endl;

}

// Просмотр максимального элемента (без удаления)

int peekMax() {

bool flag = isEmpty();

if (flag) {

cout << "Куча пуста (peekMax)" << endl;

return -1;

}

int result = heap[0];

return result; // Максимальный элемент всегда в корне

}

// Извлечение максимального элемента

int extractMax() {

bool flag = isEmpty();

if (flag) {

cout << "Куча пуста (extractMax)" << endl;

return -1;

}

int result = heap[0]; // Запоминаем максимальный элемент

// Перемещаем последний элемент в корень

int back = heap.back();

heap[0] = back;

heap.pop\_back(); // Удаляем последний элемент

// Если куча не пуста, восстанавливаем свойство просеиванием вниз

flag = isEmpty();

if (!flag) siftDown(0);

return result;

}

void displayState() {

cout << "Куча: [";

bool flag = isEmpty();

if (flag) cout << "(пусто)";

else {

int size = getSize();

for (int i = 0; i < size; ++i) {

int el = heap[i];

cout << el << " ";

}

}

cout << "]" << endl;

}

};

void clearInputBuffer() {

long long max = numeric\_limits<streamsize>::max();

cin.ignore(max, '\n');

}

// Интерфейс двоичной кучи

void runBinaryHeap() {

BinaryHeap binary\_heap{};

cout << "\n--- Меню Двоичной кучи (Max-Heap) ---" << endl;

cout << "Команды: insert <value>, extractMax, peekMax, display, size, back" << endl;

while (true) {

cout << "MaxHeap> ";

string command;

cin >> command;

if (command == "insert") {

int value;

cin >> value;

binary\_heap.insert(value);

} else if (command == "extractMax") {

int result = binary\_heap.extractMax();

if (result != -1)

cout << "Извлечен максимальный элемент: " << result << ". " << endl;

} else if (command == "peekMax") {

int result = binary\_heap.peekMax();

if (result != -1)

cout << "Максимальный элемент: " << result << endl;

} else if (command == "display") binary\_heap.displayState();

else if (command == "size") {

int size = binary\_heap.getSize();

cout << "Текущий размер кучи: " << size << endl;

} else if (command == "back") break; // Выход из меню

else {

cout << "Неизвестная команда." << endl;

clearInputBuffer();

}

}

}

int main() {

setlocale(LC\_ALL, "ru\_RU.UTF-8");

#ifdef WIN32

system("chcp 65001"); // Для UTF-8 в консоли Windows

system("cls");

#endif

runBinaryHeap();

return 0;

}

Листинг программы для задания 2:

#include <iostream>

#include <vector>

#include <list>

#include <string>

#include <fstream>

#include <sstream>

using namespace std;

class HashTable {

private:

struct HashNode {

string key;

string value;

};

vector<list<HashNode> > table;

int tableSize;

hash<string> hashFunction;

size\_t getTableIndex(string &key) {

size\_t hash = hashFunction(key);

hash = hash % tableSize;

return hash;

}

public:

int size;

explicit HashTable(int initialSize = 101) : tableSize(initialSize), size(0) {

if (tableSize == 0) tableSize = 101;

table.resize(tableSize);

}

void insert(string &key, string &value) {

size\_t hashIndex = getTableIndex(key);

list<HashNode> &chain = table[hashIndex];

for (HashNode &node: chain) {

if (node.key == key) {

node.value = value;

return;

}

}

chain.push\_front({key, value});

size++;

}

bool search(string &key, string &foundValue) {

size\_t hashIndex = getTableIndex(key);

list<HashNode> &chain = table[hashIndex];

for (HashNode &node: chain) {

if (node.key == key) {

foundValue = node.value;

return true;

}

}

return false;

}

void display() {

cout << "--- Содержимое Хеш-таблицы ---" << endl;

for (int i = 0; i < tableSize; ++i) {

list<HashNode> &chain = table[i];

bool isEmpty = chain.empty();

if (isEmpty == false) {

cout << "[" << i << "]: ";

for (HashNode &node: chain) {

cout << "(" << node.key << ", " << node.value << ") -> ";

}

cout << "null";

cout << endl;

}

}

cout << "-----------------------------" << endl;

}

};

void loadFromFile(HashTable &phoneBook) {

ifstream inputFile("phonebook.txt");

string line, name, number;

int linesRead = 0;

while (getline(inputFile, line)) {

stringstream ss(line);

ss >> name >> number;

phoneBook.insert(name, number);

linesRead++;

}

inputFile.close();

phoneBook.display();

}

int main() {

setlocale(LC\_ALL, "ru\_RU.UTF-8");

#ifdef WIN32

system("chcp 65001"); // Для UTF-8 в консоли Windows

system("cls");

#endif

HashTable phoneBook;

loadFromFile(phoneBook);

cout << "Введите имя для поиска (или 'exit' для выхода):" << endl;

string searchName;

while (true) {

cout << "> ";

getline(cin, searchName);

if (searchName == "exit") break;

string foundNumber;

bool isFound = phoneBook.search(searchName, foundNumber);

if (isFound)

cout << "Номер телефона: " << foundNumber << endl;

else cout << "Имя не найдено" << endl;

}

cout << "Завершение программы" << endl;

return 0;

}