МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МОЭВМ

КУРСОВАЯ РАБОТА

по дисциплине «Алгоритмы и структуры данных»

Тема: Анализ и реализация структуры данных, подходящей под полученное техническое задание

Студент гр. 3383 _____ Матвеев Н.С.

Преподаватель _____ Шестопалов Р.П.

Санкт-Петербург

2024

ЗАДАНИЕ
НА КУРСОВУЮ РАБОТУ

Студент Матвеев Н.С.

Группа 3383

Тема работы: Анализ и реализация структуры данных, подходящей под полученное техническое задание.

Вариант 5
Вы работаете в трейдерской компании, которая работает нехитрым образом: каждый день обновляется информация о рекомендуемой цене акций на день. Когда цена акции изменяется, ищется её рекомендуемая стоимость, и если текущая стоимость меньше, то акция покупается, иначе продается.
Но цена акций меняется очень часто, поэтому время запроса к цене может сильно сказаться на итоговой прибыли.
Компания стала нести убытки, и владелец считает, что это не из-за неэффективной стратегии трейдинга, а из-за скорости поиска рекомендуемой цены акции. Он думает, что в этом виноваты редкие ситуации, когда для поиска приходится перебирать почти все акции, которых немало.
Решение никто не придумал, поэтому владелец собирается сократить штат, в том числе и Вас, а зарплату терять не хочется, поэтому придумайте структуру данных, которая спасет компанию.
Примечание: акция представляет собой тикер (буквенное название длиной до 4 заглавных латинских символов).
Придумайте и реализуйте подходящую структуру данных.

Содержание пояснительной записки:
«Аннотация», «Содержание», «Введение», «Заключение», «Список использованных источников»

Предполагаемый объем пояснительной записки:
Не менее 15 страниц.

Дата выдачи задания: 06.11.2024

Дата сдачи реферата: 14.12.2024

Дата защиты реферата: 16.12.2024

Студент гр. 3383 _____ Матвеев Н.С.

Преподаватель _____ Шестопалов Р.П.

АННОТАЦИЯ

Данная курсовая работа посвящена разработке консольного приложения для работы с акциями и для анализа их цен — сравнивается текущая цена с рекомендуемой. Программа реализована на языке C++ и использует дерево поиска, а именно префиксное дерево для хранения тикеров акций и их цены.

Приложение позволяет добавлять акции и их рекомендуемою цену, удалять акции, осуществлять поиск тикера и сравнивать текущую цену с рекомендуемой, которая хранится в дереве, а также выводить список всех тикеров и их рекомендуемой цены в алфавитном порядке. Разработанная структура данных обеспечивает высокую производительность работы с большими объемами информации.

Информация о использовании программы выводится на экран.

SUMMARY

This term paper is devoted to the development of a console application for analyzing stock prices - the current price is compared with the recommended price. The program is implemented in C++ and uses a search tree, namely a prefix tree for storing stock tickers and their price.

The application allows you to add stocks and their recommended price, delete stocks, search for a ticker and compare the current price with the recommended price stored in the tree, as well as display a list of all tickers and their recommended price in alphabetical order. The developed data structure provides high performance of working with large amounts of information.

Information about the program usage is displayed on the screen.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Цель работы.

Цель работы заключается в разработке и реализации эффективной структуры данных для управления информацией о рекомендованных ценах акций. Эта структура данных должна обеспечивать быстрый доступ к ценам акций по тикерам, минимизируя время поиска, а также корректно хранить все тикеры и цены.

1. ВЫБОР СТРУКТУРЫ ДАННЫХ

1.1. Анализ возможных структур данных

Для реализации программы необходимо выбрать структуру данных, которая обеспечит эффективное выполнение операций добавления, удаления, вывода данных и обеспечит лучшую скорость поиска. Рассмотрим несколько возможных вариантов и оценим их преимущества и недостатки.

Хэш-таблица.

Хеш-табли́ца — структура данных, реализующая интерфейс ассоциативного массива, а именно, она позволяет хранить пары (ключ, значение) и выполнять три операции: операцию добавления новой пары, операцию удаления и операцию поиска пары по ключу.

Применение:

Хэш-таблица может использоваться для хранения тикера и рекомендуемой цены акции.

Преимущества:

- Вставка, удаление и поиск выполняются в среднем за O(1), но из-за коллизий на поиск элемента в хеш-таблице в худшем случае, может потребоваться столько же времени, как и в списке, а именно O(n).
- Простое управление большими объемами данных.

Недостатки:

- Коллизии, которые усложняют реализацию и увеличивают время вставки и поиска.
- Избежать коллизий не получится, а для их уменьшения нужно тщательнее подбирать хэш-функции.
- Потребление памяти может быть неэффективным при низкой заполненности таблицы и хранении пустых ячеек.

Двоичные деревья поиска.

Двоичное дерево поиска - это дерево, в котором у каждого из его узлов не более двух дочерних узлов. При этом каждый дочерний узел тоже представляет собой бинарное дерево. На основе их также строят ассоциативные массивы. Двоичное дерево поиска поддерживают отсортированные данные и обеспечивают O(logn) для вставки, удаления и поиска при условии что дерево сбалансировано.

Применение:

Для хранения тикеров в отсортированном виде и для стабильной работы поиска цены без худших случаев.

Преимущества:

- Поддержка упорядоченности данных.
- Стабильное выполнение вставки, удаления, поиска элемента.
- Эффективное использование памяти, так как элементы хранятся в узлах дерева без необходимости резервирования пространства для пустых ячеек, как например в хэш-таблицах.

Недостатки:

- Более сложная реализация по сравнению с хэш-таблицами.
- Скорость операций в среднем ниже, чем у хэш-таблиц при больших объемах данных.
- Необходима балансировка для поддержания эффективности.

Дерево поиска: префиксное дерево/нагруженное дерево.

Префиксное дерево поиска - древовидная структура данных для хранения множества строк или в качестве ассоциативного массива. Каждая строка представлена в виде цепочки символов, начинающейся в корне. Если у двух

строк есть общий префикс, то у них будет общий корень и некоторое количество общих вершин.

Применение:

Использование префиксного дерева в качестве ассоциативного массива для хранения тикеров(строк) и их цены.

Преимущества:

- Скорость вставки, удаления, поиска — O(k), где k — длина слова. Для коротких слов это сопоставимо константе O(1).
- Стабильное выполнение вставки, удаления, поиска элемента.
- Эффективное использование для поиска по префиксу и сортировки всех слов/ключей.

Недостатки:

- Использование памяти зависит от длины алфавита, занимает больше места чем бинарные деревья поиска.

Связный список.

Односвязный или двусвязный список можно использовать для последовательного хранения информации.

Применение:

Хранение тикеров и их цены с поддержкой последовательного перебора.

Преимущества:

- Простая структура и ее реализация.
- Удобство при добавлении и удалении элементов.
- Эффективное использование памяти.

Недостатки:

- Операции над произвольными элементами медленнее, чем с массивами, так как к произвольному элементу списка можно обратиться, только пройдя все предшествующие ему элементы.
- Линейная сложность поиска O(n) — что не подходит для данной задачи.
- Очень слабая эффективность при большом объеме данных.

Массив.

Массив - структура данных, которая хранит набор однородных данных, доступ к которым осуществляется по индексам.

Применение:

Хранение акций в массиве как пара ключ-значение.

Преимущества:

- Простая структура и ее реализация.
- Поддержка сортировки.
- Доступ к элементу по индексу.
- Эффективное использование памяти.

Недостатки:

- Вставка и удаление элементов требуют значительных затрат при больших объемах данных, что неэффективно при частом изменении данных.
- Линейная сложность поиска O(n) — что не подходит для данной задачи.

1.2. Обоснованность выбора префиксного дерева.

Префиксное дерево — это дерево поиска, позволяющее хранить ассоциативный массив, ключами которого чаще всего являются строки. Представляет собой корневое дерево, каждое ребро которого помечено каким-то символом так, что для любого узла все рёбра, соединяющие этот узел с его сыновьями, помечены разными символами.

После сравнения всех вариантов для реализации выбрано именно префиксное дерево по следующим причинам:

По условию задачи требуется реализовать такую стуктуру данных, которая гарантирует очень быстрый поиск и частый поиск и избежит редких ситуаций с перебором большого количества акций. Следовательно список и массив сразу не подходят, так как там поиск линейный и при большом наборе данных придется перебирать огромное число акций. Бинарные деревья не подойдут из-за постоянной скорости поиска O(logn), так как есть альтернативные структуры с более быстрым поиском — например хэш-таблицы, где в среднем O(1). Однако хэш-таблицы не гарантирует константное время, так как существуют коллизии, от которых не избавиться даже при самом грамотном подборе хэш-функции. Чтобы гарантировать константное время в хэш-таблицах можно использовать для борьбы с коллизиями метод хэширования кукушки, но для ее стабильной работы коэффициент заполненности должен быть меньше или равен 50%, что приводит к выделению большей памяти и к огромному количеству пустых ячеек, особенно это играет роль при огромном наборе данных — нужно будет в 2 раза больше памяти. Также есть проблема при вставке элемента при хэшировании кукушки — нужно тщательно подбирать две хэш-функции, чтобы избежать зацикленности как можно дольше, а также проводить перехэширование, а это довольно затратная операция на большом наборе данных — это также минус к варианту хэш-таблицы. Как альтернативный вариант хэширования для данной задачи — это задать таблицу фиксированного размера (в нашей задаче слова длиной от 1 до 4х символов — что задает фиксированный набор ключей), следовательно можно подобрать хэш-функцию, которая каждому ключу сопоставляет уникальное значение и избежать коллизий, однако наш набор ключей составляет 475254 — при малом наборе ключей, такое количество памяти будет крайне излишним. Для данной задачи идеально подойдет префиксное дерево, так как в этой структуре скорость поиска зависит от длины подаваемого слова O(k) k - длина, в нашем случае максимальная длина 4, следовательно это сопоставимо константе O(1). И скорость никак не зависит от количества данных в дереве — она стабильно эффективная. Тоже самое можно сказать и

про скорость вставки и удаления они также зависят от длины слова и в нашем случае сопоставимы константе. Таким образом, префиксным деревом мы гарантируем поиск сопоставимый O(1) и здесь не может быть наихудшего случая с перебором большого набора акций.

За стабильную и быструю скорость придется платить памятью, но в нашей ситуации алфавит всего 26 заглавных латинских букв, что уменьшает объем затрат. Также по памяти данная структура не сильно проигрывает бинарным деревьям, а в каких-то случаях, (когда много ключей с общим префиксом) префиксное дерево будет даже эффективнее. Если брать рассмотренные выше решения с хэш-таблицами, то в методе с кукушкой хэш-таблица с большим набором данных будет сильно уступать по памяти и половина ячеек при этом будут пустые, а в решении с фиксированным размером - при меньших объемах мы также оставим кучу пустых ячеек и проиграем по памяти.

Также стоит отметить отсутствие балансировок и перехэширования, что упрощает логику программы и в случае с перехэшированием — временные затраты.

2. СТРУКТУРА TRIENODE

Для корректной и удобной работы с префиксном деревом реализована структура узла (вершины) данного дерева – TrieNode. В своих полях структура хранит вектор указателей на узлы, которые являются детьми для данного (std::vector<TrieNode*> children), число с плавающей точкой для хранения цены (float price = 0.0), булевая переменная для обозначения что данный узел является концом слова (bool is_word = false), положительное число для хранения количества существующих детей (size_t not_nullptr = 0).

Структура узла содержит метод-конструктор, в котором вектор с указателями заполняется nullptr и задается размером 26. Также есть метод для очистки всей структуры — Clear(). В методе очистки очищается циклом массив указателей и все поля устанавливаются по умолчанию.

Объявление структуры узла расположено в файле TrieStock.h.

3. КЛАСС ПРЕФИКСНОЕ ДЕРЕВО STOCKS

3.1. Приватные поля и конструктор

Класс Stocks представляет собой основную структуру данных для управления акциями. Он объединяет данные и методы для выполнения всех операций связанных с вставкой, удалением, поиском, выводом акций.

Поля класса: указатель типа TrieNode на вершину дерева (TrieNode* head;) — сама вершина не является никакой буквой а только хранит изначальные 26 указателей. Положительное число показывающее количество ключей в дереве (size_t nums_stocks = 0).

Конструктор: Stocks() - на вход ничего не принимает, внутри создает новый экземпляр TrieNode, который будет служить корнем префиксного дерева.

3.2. Метод добавления тикера AddStock

Метод Stocks::AddStock отвечает за добавление нового тикера и его рекомендуемой цены в дерево. Метод принимает строку — название тикера и число с плавающей точкой — цену. Указатель ptr инициализируется корнем дерева, при помощи него происходит перемещение по дереву. Далее при помощи цикла for происходит проход по символам названия и для каждого символа вычисляется индекс id, который соответствует позиции символа в алфавите. Если данного дочернего узла нету, то создается новый узел и для узла родителя увеличивается счетчик ненулевых дочерних узлов. Указатель ptr переходит к следующему узлу. После завершения обхода всех символов указатель ptr указывает на конец слова. Флаг, отвечающий за конец слова становится равным true, цена становится равной переданной. Также обработана ситуация, если передавали тикер, который уже хранится в дереве — тогда проверяется был ли в данном узле флаг равным true, если нет, то счетчик количества всего ключей в дереве увеличивается.

3.3. Метод поиска ключа в дереве Stocks::FindStock

Метод Stocks::FindStock отвечает за поиск ключа в префиксном дереве и возвращает указатель на узел. Метод принимает строку — название тикера.

Указатель ptr инициализируется корнем дерева, при помощи него происходит перемещение по дереву. Далее при помощи цикла for происходит проход по символам названия и для каждого символа вычисляется индекс id, который соответствует позиции символа в алфавите. Если данного дочернего узла нету, то метод возвращает nullptr (так как просто нету нашего ключа или его префикса), иначе - указатель ptr переходит к следующему узлу. После завершения обхода всех символов указатель ptr указывает на конец слова. Если в данном узле флаг конца слова равен false, то ключа нет в дереве и возвращается nullptr, иначе возвращается указатель на узел.

3.4. Метод сравнения текущей цены и рекомендуемой Stocks::CompareStock

Метод Stocks::CompareStock отвечает за сравнение текущей цены, которую подают программе и цены, которая хранится в префиксном дереве. Метод принимает строку — название тикера и число с плавающей точкой — текущую цену. Внутри вызывается метод поиска ключа в дереве и если полученный указатель nullptr, то данного ключа нет в дереве и об этом сообщается пользователю. Иначе в соответствии с условием — если текущая цена ниже чем в дереве, то пользователю сообщается что акция должна покупаться, иначе продаваться.

3.5. Метод удаления ключа Stocks::DeleteStock

Метод Stocks::DeleteStock отвечает за удаление ключа из префиксного дерева. На вход принимает строку — название тикера. Переменная parent_symbol: индекс символа в названии тикера, после которого символы можно удалять. ptr: указатель на текущий узел, изначально указывающий на корень (head). prefix_parent: указатель на узел текущего префикса, после которого узлы можно будет удалять. counter_letter: счётчик для отслеживания позиции буквы в тикере. Цикл проходит по каждому символу в тикере, для каждого символа вычисляется индекс дочернего узла. Если дочерний узел по этому индексу равен nullptr, это означает, что тикера не существует, и метод об этом сообщает и завершает работу. Иначе необходимо запомнить последний

префиксный узел, после которого мы сможем удалять дочерние узлы. Такой узел будет храниться в prefix_parent, а индекс буквы данного узла в parent_symbol. Есть 2 варианта определить такой узел: либо у него больше чем один дочерний узел, либо у него всего один дочерний узел, но он сам является ключом/тикером. В parent_symbol будет записано текущее состояние counter_letter, которое меняется во время прохода по буквам в тикере. Указатель ptr переходит к следующему узлу. Если после цикла в данном узле флаг конца слова равен false, то значит данного тикера нет в префиксном дереве, об этом сообщается пользователю. Иначе как минимум для удаления: флаг конца слова становится равным false и цена равна 0. Если у данного узла 0 дочерних узлов, значит можно удалить узлы данного тикера после узла prefix_parent. Указатель ptr становится равным prefix_parent, находится индекс следующий буквы после parent_symbol, создается указатель previous, который будет использоваться для удаления узлов, считается индекс дочернего узла в который будет происходить переход. Пока дочерний узел существует происходит в него переход, удаляется предыдущий узел, предыдущий становится равным текущему узлу, увеличивается индекс буквы в слове, если индекс за пределами длины тикера, цикл завершается, иначе считается индекс следующего дочернего узла. Удаляется последний узел, и соответствующий дочерний указатель в родительском узле устанавливается в nullptr. Уменьшается счетчик ненулевых дочерних узлов родительского узла. В конце количество ключей уменьшается на один.

3.6. Метод очистки дерева с заданного узла Stocks::ClearTrie

Метод Stocks::ClearTrie очищает все узлы с заданного. На вход принимает указатель узла, с которого нужно очистить дерево. Если указатель nullptr, то метод просто возвращается. Это базовый случай рекурсии, который предотвращает попытки доступа к несуществующим узлам. Далее происходит обход дочерних узлов и для каждого рекурсивно вызывается данный метод. После того как все дочерние узлы были обработаны, вызывается метод Clear() для текущего узла.

3.7. Метод очистки всего дерева Stocks::ClearAllTrie

Метод Stocks::ClearAllTrie предназначен для очистки всего дерева. На вход метод ничего не принимает. Внутри вызывается метод очистки дерева с заданного узла, где заданный узел является корень дерева.

3.8. Метод вывода всех тикеров и их цен в алфавитном порядке Stocks::PrintLikeTable

Метод Stocks::PrintLikeTable рекурсивно обходит дерево и выводит на экран все тикеры, которые хранятся в узлах, вместе с их ценами. На вход принимает указатель на узел, с которого начинается обход и строку, представляющая текущий префикс слова, которая формируется во время обхода дерева. Если указатель nullptr, то метод просто возвращается. Это базовый случай рекурсии, который предотвращает попытки доступа к несуществующим узлам. Если текущий узел является концом слова, то выводится текущий префикс и цена, которая хранится в данном узле. Далее происходит обход дочерних узлов и для каждого в текущий префикс добавляется соответствующая буква и рекурсивно вызывается данный метод. После завершения рекурсивного вызова последняя добавленная буква удаляется из префикса. Это позволяет вернуть префикс к предыдущему состоянию, чтобы правильно формировать слова для следующих дочерних узлов.

Данный метод вызывается из публичного метода Stocks::PrintStocks, где создается пустая строка-префикс, соответствующая вершине дерева и передается указатель на вершину в метод Stocks::PrintLikeTable.

3.9. Метод обхода в глубину структуры Stocks::DFS

Метод Stocks::DFS предназначен для обхода в глубину. На вход принимает указатель на узел, с которого начинается обход и строку, представляющая текущий префикс слова, которая формируется во время обхода дерева. Если указатель nullptr, то метод просто возвращается. Это базовый случай рекурсии, который предотвращает попытки доступа к несуществующим узлам. Далее в текущий префикс добавляется

соответствующая буква и выводится данный префикс. Далее происходит обход дочерних узлов и рекурсивно вызывается данный метод для каждого префикса. После завершения рекурсивного вызова последняя добавленная буква удаляется из префикса. Это позволяет вернуть префикс к предыдущему состоянию, чтобы правильно формировать слова для следующих дочерних узлов.

Данный метод может быть вызван из публичного метода Stocks::PrintAllPrefix, в котором создается пустая строка-префикс, соответствующая вершине дерева и передается указатель на вершину.

4. ПРИЛОЖЕНИЕ – CLI

В данном коде реализованы функции для взаимодействия с созданной структурой в соответствии с заданием.

Реализована функция ввода данных с консоли InputData для ввода рекомендуемой цены, то есть добавление тикеров и их цен в дерево, а также для сравнения текущей цены с рекомендуемой, которая хранится в дереве. Для определения цели ввода, функция принимает флаг из класса перечисления TypeInput{RecommendedInput, CompareInput}. Также во время ввода происходит проверка на корректность строки и числа при помощи функций CheckName — для проверки названия тикера и CheckPrice — для проверки цены тикера. При ошибке ввода пользователю об этом сообщается и дают новую попытку ввода для данного тикера. Обработка происходит при помощи цикла while и блока try-catch. Также ввести рекомендуемые цены, то есть добавить их в дерево можно при помощи функции InputFromFile, которая считывает все тикеры и их цены из файла и заносит их в реализованную структуру. Если происходил ввод рекомендуемых цен, то вызывается функция AfterRecommended, если ввод для анализа, то AfterAnalyze.

Для дальнейшей работы с программой реализованы функции AfterRecommended, AfterAnalyze, AfterLook. Которые предоставляют выбор действия после ввода рекомендуемых цен / ввода для анализа / просмотра всех тикеров и их цен. В которых происходит обработка на корректность ввода команд, а также вызываются дальнейшие взаимодействия с программой.

Для закрытия программы реализована функция CloseProgram. В котором удаляется текущий объект класса и программа завершает свою работу.

5. MAIN

В данном коде реализовано первое взаимодействие с программой, пользователю дается выбор для дальнейшего действия: закрыть программу, ввести рекомендуемые цены и тикеры из файла или с консоли. Команды обрабатываются на корректность ввода и в зависимости от выбора вызывают соответствующий функции из реализованного блока CLI.

6. АНАЛИЗ ВРЕМЕНИ РАБОТЫ

Для реализованного класса префиксное дерево было замерено время выполнения работы операций вставки, поиска и удаления для разного количества данных (100, 1000, 10000, 100000, 475254). Результаты представлены в таблице 1.

Таблица 1: Результаты замеров времени выполнения базовых операций

	Поиск и сравнение текущей цены и рекомендуемой	Добавление тикера	Удаление тикера
100 элементов	0.0000006711	0.00000147	0.00000115
1000 элементов	0.0000007530	0.00000116	0.0000009020
10000 элементов	0.0000006170	0.00000114	0.00000093
100000 элементов	0.0000005100	0.00000118	0.00000110
475254 элементов	0.0000008010	0.00000076	0.00000107

Представленные данные показывают, что от изменения количества элементов в изначальных данных время работы операций никаким образом не изменяется, что говорит о том, что время выполнения как и предполагалось сопоставимо константе O(1).

Графики представлены на рисунках 1-3.

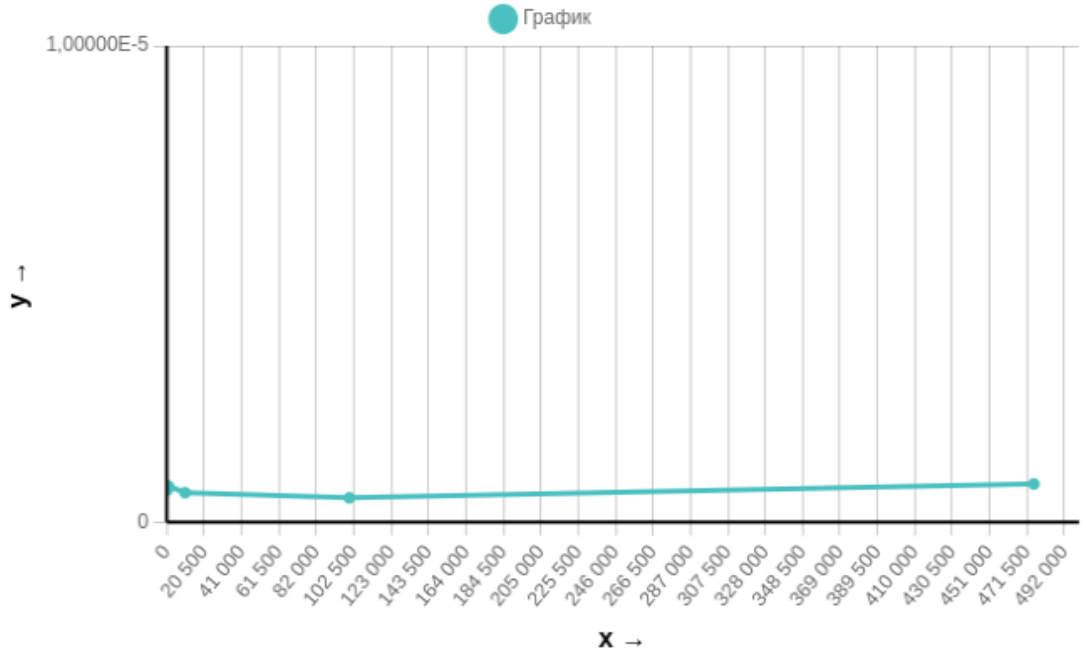

Рисунок 1 Поиск и сравнение цены

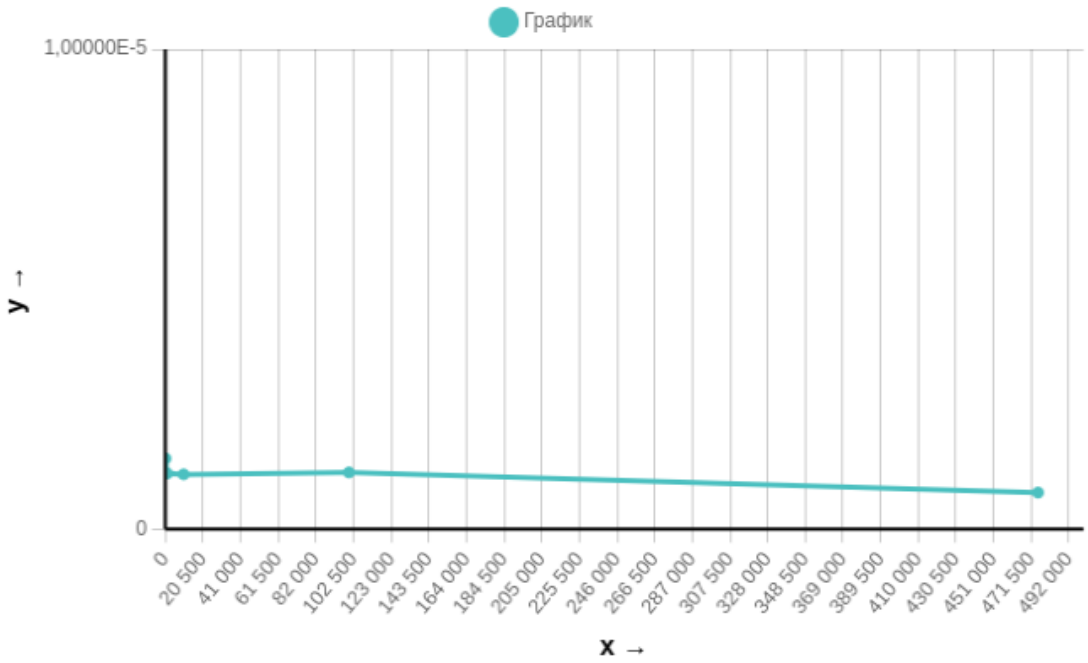

Рисунок 2 Добавление тикера

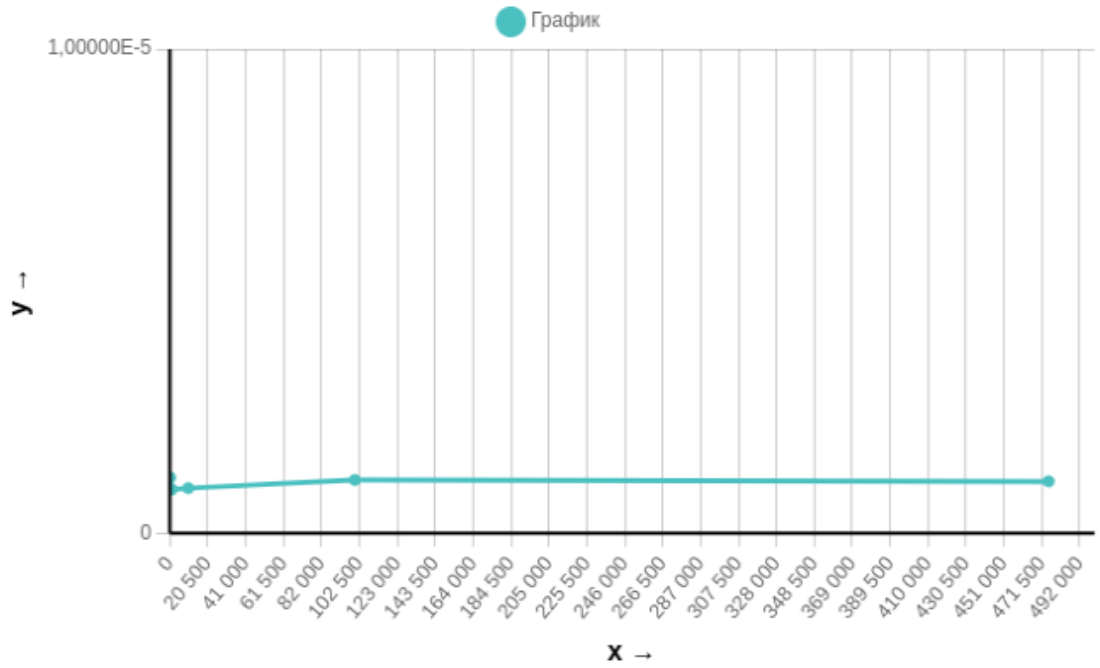

Рисунок 3 Удаление тикера

ЗАКЛЮЧЕНИЕ

В результате разработана программа, которая позволяет эффективно работать с тикерами акций.. Выбранная структура данных в виде префиксного дерева успешно решает поставленную задачу управления информацией о рекомендованных ценах акций. Благодаря своей эффективности, она обеспечивает быстрый доступ к данным по тикерам и минимизирует время поиска, что является критически важным в данной задаче. Корректное хранение всех тикеров и цен дополнительно подтверждает надежность предложенного решения. Таким образом, реализованная структура данных не только соответствует заявленным требованиям, но и демонстрирует высокую производительность, что делает ее оптимальным инструментом для работы с данными в реальном времени.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Лекционные материалы // google drive. URL: https://drive.google.com/drive/folders/1xlbCBdI41hcw-kBPH-ZsmLlA-3in5scV (дата обращения: 04.12.2024).

2. Хэширование кукушки // ИТМО вики. URL: https://neerc.ifmo.ru/wiki/index.php?title=Хеширование_кукушки (дата обращения 05.12.2024).

3. Префиксное дерево // Википедия. URL: https://ru.wikipedia.org/wiki/Префиксное_дерево (дата обращения 05.12.2024).

ПРИЛОЖЕНИЕ А

ПРИМЕРЫ РАБОТЫ ПРОГРАММЫ

1. Ввод тикеров с консоли и их вывод в алфавитном порядке

```
Entry recommended prices
Enter "NAME price"
If you want to finish entering prices - enter "Stop"
XXXX 345
ER 23.678
WER 123.45
A 123456
Stop
Now you can look at all stocks or start analyzing!
If you want to see stocks - enter "Look"
If you want to close program - enter "Analyze"
Look
A -- 123456
ER -- 23.678
WER -- 123.45
XXXX -- 345
```

Рисунок 4 Тестирование программы №1

2. Ввод тикеров из файла и вывод их в алфавитном порядке

```
File
Now you can look at all stocks or start analyzing!
If you want to see stocks - enter "Look"
If you want to close program - enter "Analyze"
Look
A -- 18.55
EBQU -- 47.26
JFB -- 19.96
MD -- 83.96
NAKH -- 46.29
OZ -- 41.1
QNEQ -- 49.98
T -- 74.32
V -- 19.75
WXKS -- 13.49
```

Рисунок 5 Тестирование программы №2

3. Сравнение текущей цены с рекомендуемой на тикерах из примера выше

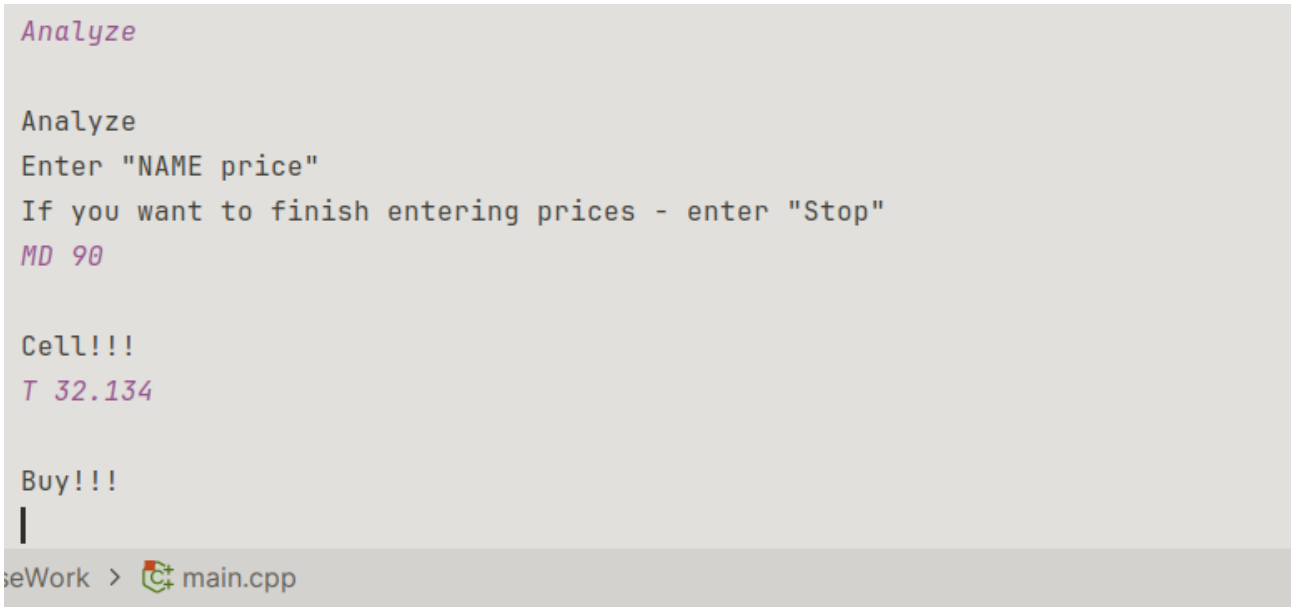

Рисунок 6 Тестирование программы №3

Если пытаются анализировать акцию, которой нету

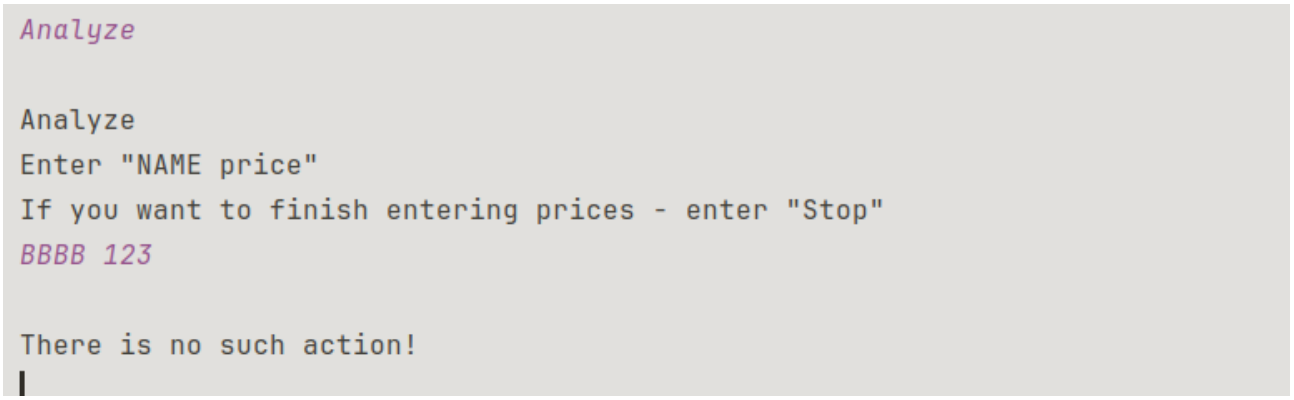

Рисунок 7 Тестирование программы №4

ПРИЛОЖЕНИЕ Б

ИСХОДНЫЙ КОД ПРОГРАММЫ

файл main.cpp:

```
#include "CLI.h"
int main() {
   std::cout << "Now Stock Market is closed! List of stocks empty.\n"
             "If you want to add Recommended prices from terminal - enter \"Terminal\" \n"
             "If you want to add Recommended prices from file - enter \"File\" \n"
             "If you want to close program - enter \"Close\" \n";
   std::string input;
   std::getline(std::cin, input);
   while (input != "Terminal" &&  input != "Close" && input != "File"){
      std::cout << "You entered an unknown command: ";
      std::cout << input  << "\n";
      std::cout << "Try again!\n";
      std::cin.clear();
      std::cin.sync();
      std::getline(std::cin, input);
   }
   auto* stocks_data = new Stocks;
   if (input == "Terminal"){
      InputData(stocks_data, TypeInput::RecommendedInput);
   }
   else if (input == "File"){
      InputFromFile(stocks_data);
      AfterRecommended(stocks_data);
   }
   else CloseProgram(stocks_data);
   return 0;
}
```

файл CLI.h:

```
#ifndef CPPCOURSEWORK_CLI_H
#define CPPCOURSEWORK_CLI_H
#include "TrieStock.h"
enum class TypeInput{
   RecommendedInput,
   CompareInput
};
bool CheckName(std::string& str);
bool CheckPrice(std::string& str, float& our_price);
void HandleInputError();
void CloseProgram(Stocks* StockMarket);
void InputData(Stocks* stocks_data, TypeInput flag);
void AfterAnalyze(Stocks* stocks_data);
void AfterRecommended(Stocks* stocks_data);
void AfterLook(Stocks* stocks_data);
void InputFromFile(Stocks* stocks_data);
#endif //CPPCOURSEWORK_CLI_H
```

файл CLI.cpp:

```
#include "CLI.h"
#include "TrieStock.h"

bool CheckName(std::string& str){
    if (str.size() > 4) return false;
    bool flag = true;
    for (char i : str){
        if (!std::isupper(i)){
            flag =  false;
            break;
        }
    }
    return flag;
}
bool CheckPrice(std::string& str, float& our_price){
    float price = std::stof(str);
    if (price < 0){
        std::cout << "Error: Price can`t be < 0!\n";
        return false;
    }
    our_price = price;
    return true;
}
void HandleInputError() {
    std::cout << "Enter again: ";
    std::cin.clear();
    std::cin.sync();
}
void CloseProgram(Stocks* StockMarket){
    std::cout << "\nProgram close....\n";
    delete StockMarket;
}

void AfterLook(Stocks* stocks_data){
    std::string input;
    std::cout << "Now you can analyze or change recommended prices or close all program!\n"
                 "If you want to change recommended prices - enter \"Update\" \n"
                 "If you want to close program - enter \"Analyze\" \n"
                 "If you want to close program - enter \"Close\" \n";
    std::getline(std::cin, input);
    while (input != "Update" &&  input != "Analyze" && input != "Close"){
        std::cout << "You entered an unknown command: ";
        std::cout << input  << "\n";
        std::cout << "Try again!\n";
        std::cin.clear();
        std::cin.sync();
        std::getline(std::cin, input);
    }
    if (input == "Close"){
        CloseProgram(stocks_data);
    }
    else if (input=="Update"){
        InputData(stocks_data, TypeInput::RecommendedInput);
    }
    else{
```

```
        InputData(stocks_data, TypeInput::CompareInput);
    }
}
void AfterRecommended(Stocks* stocks_data){
    std::string input;
    std::cout << "Now you can look at all stocks or start analyzing!\n"
                 "If you want to see stocks - enter \"Look\" \n"
                 "If you want to close program - enter \"Analyze\" \n";
    std::getline(std::cin, input);
    while (input != "Look" &&  input != "Analyze"){
        std::cout << "You entered an unknown command: ";
        std::cout << input  << "\n";
        std::cout << "Try again!\n";
        std::cin.clear();
        std::cin.sync();
        std::getline(std::cin, input);
    }
    if (input == "Look"){
        stocks_data->PrintStocks();
        AfterLook(stocks_data);
    }
    else InputData(stocks_data, TypeInput::CompareInput);
}
void AfterAnalyze(Stocks* stocks_data){
    std::string input;
    std::cout << "Now you can look at all stocks or change recommended prices or close all
program!\n"
                 "If you want to see stocks - enter \"Look\" \n"
                 "If you want to change recommended prices from terminal - enter \"Update1\" \n"
                 "If you want to change recommended prices from file - enter \"Update2\" \n"
                 "If you want to close program - enter \"Close\" \n";
    std::getline(std::cin, input);
    while (input != "Look" &&  input != "Close" && input != "Update1" && input != "Update2"){
        std::cout << "You entered an unknown command: ";
        std::cout << input  << "\n";
        std::cout << "Try again!\n";
        std::cin.clear();
        std::cin.sync();
        std::getline(std::cin, input);
    }
    if (input == "Look"){
        stocks_data->PrintStocks();
        AfterLook(stocks_data);
    }
    else if (input == "Close"){
        CloseProgram(stocks_data);
    }
    else if (input == "Update1"){
        InputData(stocks_data, TypeInput::RecommendedInput);
    }
    else{
        InputFromFile(stocks_data);
        AfterRecommended(stocks_data);
    }
}
void InputData(Stocks* stocks_data, TypeInput flag){
    if (flag == TypeInput::RecommendedInput){
```

```
        std::cout << "\nEntry recommended prices\n";
    }
    else std::cout << "\nAnalyze\n";
    std::cout << "Enter \"NAME price\" \n";
    std::cout << "If you want to finish entering prices - enter \"Stop\" \n";
    std::string input;
    std::getline(std::cin, input);
    std::string word;
    float price;
    while (input != "Stop"){
        while (true){
            try{
                size_t id = input.find(' ');
                if (id != std::string::npos){
                    word = input.substr(0, id);
                    if (!CheckName(word)){
                        std::cout << "Error: wrong ticker!\n";
                        throw std::invalid_argument("");
                    }
                    std::string price_str = input.substr(id+1);
                    if (!CheckPrice(price_str, price)){
                        throw std::invalid_argument("");
                    }
                    if (flag == TypeInput::RecommendedInput){
                        stocks_data->AddStock(word, price);
                    }
                    else if (flag == TypeInput::CompareInput){
                        stocks_data->CompareStock(word, price);
                    }
                    break;
                }
                else{
                    std::cout << "Error: wrong string!\n";
                    throw std::invalid_argument("");
                }
            }
            catch (const std::invalid_argument& e ) {
                HandleInputError();
                std::getline(std::cin, input);
            }
            catch (const std::out_of_range& e){
                HandleInputError();
                std::getline(std::cin, input);
            }
        }
        std::cin.clear();
        std::cin.sync();
        std::getline(std::cin, input);
    }
    if (flag == TypeInput::RecommendedInput){
        AfterRecommended(stocks_data);
    }
    else{ // compare
        AfterAnalyze(stocks_data);
    }
}
```

```
void InputFromFile(Stocks* stocks_data){
    std::string file_name = "/home/nikita/stocks.txt";
    std::ifstream file(file_name);
    std::string line;
    std::string word;
    float price;
    while (std::getline(file, line)) {
        size_t id = line.find(' ');
        word = line.substr(0, id);
        price = std::stof(line.substr(id));
        stocks_data->AddStock(word, price);
    }
    file.close();
}
```

файл TrieStock.h:

```
#ifndef CPPCOURSEWORK_TRIESTOCK_H
#define CPPCOURSEWORK_TRIESTOCK_H
#include <iostream>
#include <vector>
#include <chrono>
#include <iomanip>
#include <fstream>
struct TrieNode{
    std::vector<TrieNode*> children;
    float price = 0.0;
    bool is_word = false;
    size_t not_nullptr = 0;
    TrieNode(){
        children.assign(26, nullptr);
    }
    void Clear(){
        for (auto& i : children){
            delete i;
        }
        children.clear();
        price = 0.0;
        is_word = false;
        not_nullptr = 0;
    }
};
class Stocks{
private:
    TrieNode* head;
    size_t nums_stocks = 0;
    void PrintLikeTable(TrieNode* node, std::string& prefix);
    TrieNode* FindStock(std::string& ticker);
    void ClearTrie(TrieNode* node);
    void DFS(TrieNode* node, std::string& prefix);
public:
    Stocks();
    void AddStock(std::string& ticker, float price);
    void DeleteStock(std::string& ticker);
    void CompareStock(std::string & ticker, float current_price);
    void PrintStocks();
    void PrintAllPrefix();
```

```
    void ClearAllTrie();
    size_t GetNumsStocks();
    bool IsEmpty();
    ~Stocks();
};
#endif //CPPCOURSEWORK_TRIESTOCK_H
```

файл TrieStock.cpp:

```
#include "TrieStock.h"

void Stocks::PrintLikeTable(TrieNode *node, std::string& prefix) {
    if (node == nullptr) return;
    if (node->is_word){
        std::cout << prefix << " -- " << node->price << "\n";
    }
    for (int i =0; i<26; ++i){
        prefix.push_back('A'+i);
        PrintLikeTable(node->children[i], prefix);
        prefix.pop_back();
    }
}

TrieNode *Stocks::FindStock(std::string &ticker) {
    TrieNode* ptr = head;
    size_t id;
    for (char i : ticker) {
        id = i - 'A';
        if (ptr->children[id] == nullptr){
            return nullptr;
        }
        ptr = ptr->children[id];
    }
    if (ptr->is_word) return ptr;
    else return nullptr;
}

void Stocks::ClearTrie(TrieNode *node) {
    if (node == nullptr) return;
    else{
        for (size_t i =0; i<26; ++i){
            ClearTrie(node->children[i]);
        }
        node->Clear();
    }
}

void Stocks::DFS(TrieNode *node, std::string& prefix) {
    if (node == nullptr) return;
    for (int i =0; i<26; ++i){
        if (node->children[i] != nullptr){
            prefix.push_back('A'+i);
            std::cout << prefix << std::endl;
            DFS(node->children[i], prefix);
            prefix.pop_back();
        }
    }
```

```
}

Stocks::Stocks() {
    head = new TrieNode();
}

void Stocks::AddStock(std::string &ticker, float price) {
    TrieNode* ptr = head;
    size_t id;
    for (char i : ticker){
        id = i - 'A';
        if (ptr->children[id] == nullptr){
            ptr->children[id] = new TrieNode();
            ptr->not_nullptr+=1;
        }
        ptr = ptr->children[id];
    }
    if (!ptr->is_word){
        nums_stocks++;
    }
    ptr->is_word = true;
    ptr->price = price;
}

void Stocks::DeleteStock(std::string &ticker) {
    int parent_symbol = -1;
    TrieNode* ptr = head;
    TrieNode* prefix_parent = head;
    size_t id;
    size_t counter_letter = 0;
    for (char i : ticker) {
        id = i - 'A';
        if (ptr->children[id] == nullptr) {
            std::cout << "\nThere is no such action!\n";
            return;
        }
        else{
            if (ptr->children[id]->not_nullptr > 1 ){
                prefix_parent = ptr->children[id];
                parent_symbol = counter_letter;
            }
            else if (ptr->children[id]->not_nullptr == 1 && ptr->children[id]->is_word){
                prefix_parent = ptr->children[id];
                parent_symbol = counter_letter;
            }
            ptr = ptr->children[id];
        }
        counter_letter++;
    }

    if (!ptr->is_word){
        std::cout << "\nThere is no such action!\n";
        return;
    }

    ptr->is_word = false;
    ptr->price =0.0;
```

```
    if (ptr->not_nullptr == 0){
        ptr = prefix_parent;
        TrieNode* previous = nullptr;
        size_t index_for_letter = parent_symbol+1;
        id = ticker[index_for_letter] - 'A';
        size_t post_prefix_id = id;
        while (ptr->children[id] != nullptr){
            ptr = ptr->children[id];
            delete previous;
            previous = ptr;
            index_for_letter++;
            if (index_for_letter < ticker.size()) id = ticker[index_for_letter] - 'A';
            else break;
        }
        delete previous;
        prefix_parent->children[post_prefix_id] = nullptr;
        prefix_parent->not_nullptr -=1;
    }
    nums_stocks--;
}

void Stocks::CompareStock(std::string &ticker, float current_price) {
    TrieNode* ptr = FindStock(ticker);
    if (ptr == nullptr){
        std::cout << "\nThere is no such action!\n";
        return;
    }
    if (current_price < ptr->price){
        std::cout << "\nBuy!!!\n";
    }
    else{
        std::cout << "\nCell!!!\n";
    }
}

void Stocks::PrintStocks() {
    std::string prefix;
    PrintLikeTable(head, prefix);
}

void Stocks::PrintAllPrefix() {
    std::string prefix;
    DFS(head, prefix);
}

void Stocks::ClearAllTrie() {
    ClearTrie(head);
}

size_t Stocks::GetNumsStocks() {
    return nums_stocks;
}

bool Stocks::IsEmpty() {
    return head->not_nullptr == 0;
}
```

```
Stocks::~Stocks() {
   ClearTrie(head);
}
```

файл Tests.cpp:

```
#include "TrieStock.h"
#include "gtest/gtest.h"
TEST(TrieTests, Test1){
   Stocks stock_market = Stocks();
   std::string ticker;
   ticker = "ABCD";
   EXPECT_TRUE(stock_market.IsEmpty());
   stock_market.AddStock(ticker, 123.44);
   EXPECT_EQ(stock_market.GetNumsStocks(), 1);
   ticker = "AB";
   stock_market.AddStock(ticker, 56.445);
   ticker = "FBDC";
   stock_market.AddStock(ticker, 12);
   ticker = "Z";
   stock_market.AddStock(ticker, 1234);
   EXPECT_EQ(stock_market.GetNumsStocks(), 4);
   TrieNode* ptr = stock_market.FindStock(ticker);
   EXPECT_EQ(ptr->price, 1234);
   ticker = "FBDC";
   ptr = stock_market.FindStock(ticker);
   EXPECT_EQ(ptr->price, 12);
   ticker = "AB";
   ptr = stock_market.FindStock(ticker);
   EXPECT_NEAR(ptr->price, 56.445, 0.001);
   EXPECT_EQ(ptr->not_nullptr, 1);
   ticker = "ABCD";
   ptr = stock_market.FindStock(ticker);
   EXPECT_NEAR(ptr->price, 123.44, 0.001);
   EXPECT_EQ(ptr->not_nullptr, 0);
   EXPECT_FALSE(stock_market.IsEmpty());
}
TEST(TrieTests, Test2){
   Stocks stock_market = Stocks();
   std::string ticker;
   ticker = "ABCD";
   stock_market.AddStock(ticker, 123.44);
   ticker = "AB";
   stock_market.AddStock(ticker, 56.445);
   ticker = "A";
   stock_market.AddStock(ticker, 12);
   ticker = "BCD";
   stock_market.AddStock(ticker, 1234);
   ticker = "BC";
   stock_market.AddStock(ticker, 11);
   ticker = "ZZZZ";
   stock_market.AddStock(ticker, 999);
   EXPECT_NE(stock_market.FindStock(ticker), nullptr);
   stock_market.DeleteStock(ticker);
   EXPECT_EQ(stock_market.GetNumsStocks(), 5);
   EXPECT_EQ(stock_market.FindStock(ticker), nullptr);
```

```
    ticker = "BC";
    EXPECT_NE(stock_market.FindStock(ticker), nullptr);
    stock_market.DeleteStock(ticker);
    EXPECT_EQ(stock_market.GetNumsStocks(), 4);
    EXPECT_EQ(stock_market.FindStock(ticker), nullptr);
    ticker = "BCD";
    EXPECT_NE(stock_market.FindStock(ticker), nullptr);
    stock_market.DeleteStock(ticker);
    EXPECT_EQ(stock_market.GetNumsStocks(), 3);
    EXPECT_EQ(stock_market.FindStock(ticker), nullptr);
    ticker = "A";
    EXPECT_NE(stock_market.FindStock(ticker), nullptr);
    stock_market.DeleteStock(ticker);
    EXPECT_EQ(stock_market.GetNumsStocks(), 2);
    EXPECT_EQ(stock_market.FindStock(ticker), nullptr);
    ticker = "AB";
    TrieNode* ptr = stock_market.FindStock(ticker);
    EXPECT_EQ(ptr->not_nullptr, 1);
    ticker = "ABCD";
    EXPECT_NE(stock_market.FindStock(ticker), nullptr);
    stock_market.DeleteStock(ticker);
    EXPECT_EQ(stock_market.GetNumsStocks(), 1);
    EXPECT_EQ(stock_market.FindStock(ticker), nullptr);
    EXPECT_EQ(ptr->not_nullptr, 0);
    ticker = "AB";
    EXPECT_NE(stock_market.FindStock(ticker), nullptr);
    stock_market.DeleteStock(ticker);
    EXPECT_EQ(stock_market.GetNumsStocks(), 0);
    EXPECT_EQ(stock_market.FindStock(ticker), nullptr);
    EXPECT_TRUE(stock_market.IsEmpty());
}

TEST(TrieTest, Test3){
    Stocks stock_market = Stocks();
    std::string ticker;
    ticker = "A";
    stock_market.AddStock(ticker, 123.44);
    ticker = "AB";
    stock_market.AddStock(ticker, 56.445);
    ticker = "ABC";
    stock_market.AddStock(ticker, 12);
    ticker = "B";
    stock_market.AddStock(ticker, 1234);
    ticker = "BC";
    stock_market.AddStock(ticker, 11);
    ticker = "BCD";
    stock_market.AddStock(ticker, 11);
    ticker = "ZZZZ";
    stock_market.AddStock(ticker, 999);
    ticker = "QE";
    stock_market.AddStock(ticker, 999);
    ticker = "FTR";
    stock_market.AddStock(ticker, 999);
    ticker = "F";
    stock_market.AddStock(ticker, 999);
    ticker = "LZ";
```

```
    stock_market.AddStock(ticker, 999);

    ticker = "A";
    stock_market.DeleteStock(ticker);
    ticker = "ABC";
    stock_market.DeleteStock(ticker);
    ticker = "AB";
    stock_market.DeleteStock(ticker);
    ticker = "F";
    stock_market.DeleteStock(ticker);
    ticker = "FTR";
    stock_market.DeleteStock(ticker);
    ticker = "ZZZZ";
    stock_market.DeleteStock(ticker);
    ticker = "BCD";
    stock_market.DeleteStock(ticker);
    ticker = "LZ";
    stock_market.DeleteStock(ticker);
    EXPECT_EQ(stock_market.GetNumsStocks(), 3);
    ticker = "B";
    EXPECT_EQ(stock_market.FindStock(ticker)->not_nullptr, 1);
    EXPECT_NE(stock_market.FindStock(ticker)->children[2], nullptr);
    stock_market.DeleteStock(ticker);
    EXPECT_EQ(stock_market.GetNumsStocks(), 2);
}
int main(int argc, char **argv){
    testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

файл Makefile:

```
all: main Tests clean
main: TrieStock.o CLI.o main.o
	g++ TrieStock.o CLI.o main.o -o main
Tests: TrieStock.o Tests.o
	g++ TrieStock.o Tests.o -lgtest -o Tests
TrieStock.o: TrieStock.h TrieStock.cpp
	g++ -c TrieStock.cpp
CLI.o: CLI.h TrieStock.h CLI.cpp
	g++ -c CLI.cpp
main.o: main.cpp CLI.h
	g++ -c main.cpp
Tests.o: Tests.cpp TrieStock.h
	g++ -c Tests.cpp
clean:
	rm -rf *.o
```