**Санкт-Петербургский политехнический университет Петра Великого**

**Институт кибербезопасности и защиты информации**

**КУРСОВАЯ РАБОТА**

**Игра «Балда»**

по дисциплине «Структуры данных»

|  |  |  |
| --- | --- | --- |
| Выполнил студент  гр. 4851003/0001 | <*подпись*> | В. Тоцкий |
| Руководитель  Ассистент | <*подпись*> | И. Д. Панков |

|  |
| --- |
| «06» июня 2021 г. |

Санкт-Петербург

2021

**Содержание**

**Введение3**

**Структуры данных4**

**Алгоритм поиска7**

**Остальные алгоритмы11**

**Графика13**

**Заключение15**

# **Введение**

Одними из важнейших процедур обработки структурированной информации является поиск. Задача поиска привлекала большое внимание ученых (программистов) еще на заре компьютерной эры.

У каждого алгоритма есть свои преимущества и недостатки. Поэтому важно выбрать тот алгоритм, который лучше всего подходит для решения конкретной задачи. Задачу поиска можно сформулировать так: найти один или несколько элементов в множестве, причем искомые элементы должны обладать определенным свойством. Это свойство может быть абсолютным или относительным. Относительное свойство характеризует элемент по отношению к другим элементам: например, минимальный элемент в множестве чисел.

Таким образом, в задаче поиска имеются следующие шаги:

1) вычисление свойства элемента; часто это - просто получение «значения» элемента, ключа элемента и т. д.;

2) сравнение свойства элемента с эталонным свойством (для абсолютных свойств) или сравнение свойств двух элементов (для относительных свойств);

3) перебор элементов множества, т. е. прохождение по элементам множества.

Первые два шага относительно просты. Вся суть различных методов поиска сосредоточена в методах перебора, в стратегии поиска и здесь возникает ряд вопросов

В данной курсовой работе была предпринята попытка реализовать оптимальный поиск слов по заданным в правилах игры параметрам и дружественный интерфейс для пользователей с использованием

## **Структуры данных.**

Для того, чтобы поиск был максимально быстрым, требовалось выбрать оптимальную структуру данных для хранения словаря и алгоритм, который бы не зависел от количества уже выставленных слов и линейно зависел от размера поля (линейная зависимость в данном случае лучшая из возможных).

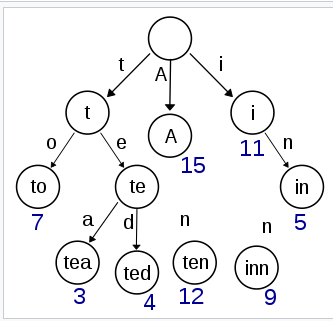
Для хранения данных (словарь из 11644 слов) была выбрана такая структура данных, как префиксное дерево, в котором поиск любого слова имеет сложность

Рисунок 1 – Представление префиксного дерева

Префиксное дерево - структура данных, позволяющая хранить ассоциативный массив, ключами которого 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. Некоторые узлы префиксного дерева выделены (на рисунке они подписаны цифрами) и считается, что префиксное дерево содержит данную строку-ключ тогда и только тогда, когда эту строку можно прочитать на пути из корня до некоторого (единственного для этой строки) выделенного узла. В некоторых приложениях удобно считать все узлы дерева выделенными.

Таким образом, в отличие от бинарных деревьев поиска, ключ, идентифицирующий конкретный узел дерева, не явно хранится в данном узле, а задаётся положением данного узла в дереве. Получить ключ можно выписыванием подряд символов, помечающих рёбра на пути от корня до узла. Ключ корня дерева — пустая строка. Часто в выделенных узлах хранят дополнительную информацию, связанную с ключом, и обычно выделенными являются только листья и, возможно, некоторые внутренние узлы.

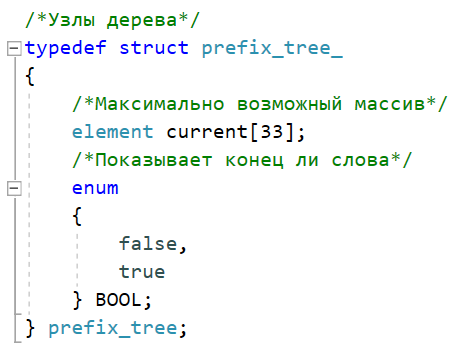
Выделяют три основные операции над префиксным деревом: проверка наличия ключа в дереве, удаление ключа из дерева и вставка нового ключа (возможно, с какой-то дополнительной связанной информацией). Каждая из этих операций реализуется с помощью спуска по дереву из корня, но эффективность такой операции напрямую зависит от организации навигации по узлам. В реализации игры использовалось хранение данных с помощью статического массива, состоящего из 33 элементов, что является не лучшим решением, как с точки зрения сложности по памяти, так и с точки зрения сложности по скорости, но данный способ самый простой в реализации. Для того, чтобы оптимизировать навигацию можно было бы использовать вместо статического массива динамический, а еще лучше сбалансированное дерево АВЛ или КЧ, это позволило бы добиться вычислительной сложности .

Рисунок 2 – Структура узла префиксного дерева

В узле хранится массив с буквами и указателями на следующий узел, которые составляют слова и перечислимая переменная, которая обозначает, является ли слово законченным на этом шаге, то есть, например, для слова «яблоко», элемент, который следует за ним будет иметь в переменной BOOL значение true, если предыдущее слово не закончено, то там хранится значение false.

Также помимо основного словаря в программе строится второе префиксное дерево, которое содержит все инверсированные префиксы для исходного словаря. Программное построение такого дерева на основе словаря достаточно затратная операция, которая требует большого количества времени, поэтому было решено построить его один раз и повторно строить уже из образа, сохраненного в отдельном текстовом файле, не содержащим повторений, все элементы являются инверсированными и отсортированными. Трудности могут возникать при добавлении пользователем слов. Для добавленных слов как раз происходит программная инверсия и разбиение на префиксы, но не обновляется образ исходного дерева, то есть каждый раз приходится отдельно добавлять каждый элемент, что при большом объеме данных, добавленных пользователем слов, программа может заметно медленнее выполняться.

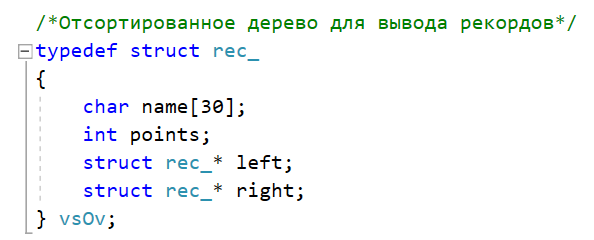
Помимо префиксных деревьев в программе строится бинарное дерево, которое используется для хранения рекордов. Это удобно, так как значения буду всегда отсортированы и сложность добавления будет всегда .

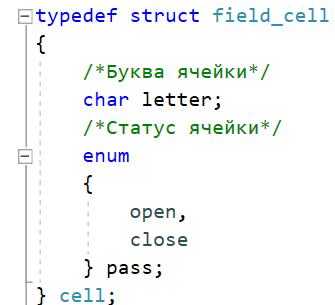
Рисунок 3 – Структура бинарного дерева

### **Алгоритм поиска ботом слов.**

По правилам балды для хода нужно выбрать пустую клетку, и поставить туда любую букву. Затем нужно составить слово, используя поставленную букву. Данная буква может быть как первой буквой в слове, так и не первой буквой (хоть и капитанство, но это важно). Если бы в правилах было сказано, что поставленная буква должна быть точно первой, то просто используем словарное дерево, двигаясь по полю от поставленной буквы одновременно спускаясь по дереву. Но так как поставленная буква может и не быть первой, словарное дерево использовать нельзя, потому что не известно с какого именно узла начинать поиск. Но у нас ещё есть инвертированное дерево.

Идея такая: из клетки с поставленной буквой движемся по полю, спускаясь по инвертированному дереву. Если находим узел, у которого BOOL имеет значение true, значит это валидный инвертированный префикс слова. Имея префикс слова, можно в словарном дереве найти узел, соответствующий поставленной букве, и далее просто найти оставшийся кусок слова уже в словарном дереве, начиная с этого узла.

Для того чтобы алгоритм не заходил в клетки, которые ранее для данного слова уже прошёл, структура, которая хранит ячейки поля имеет переменную перечислимого типа pass, которая в свою очередь имеет 2 состояния – closed (то есть ячейка закрыта) и open (то есть ячейка открыта).

Рисунок 4 – Структура, хранящее поле для игры

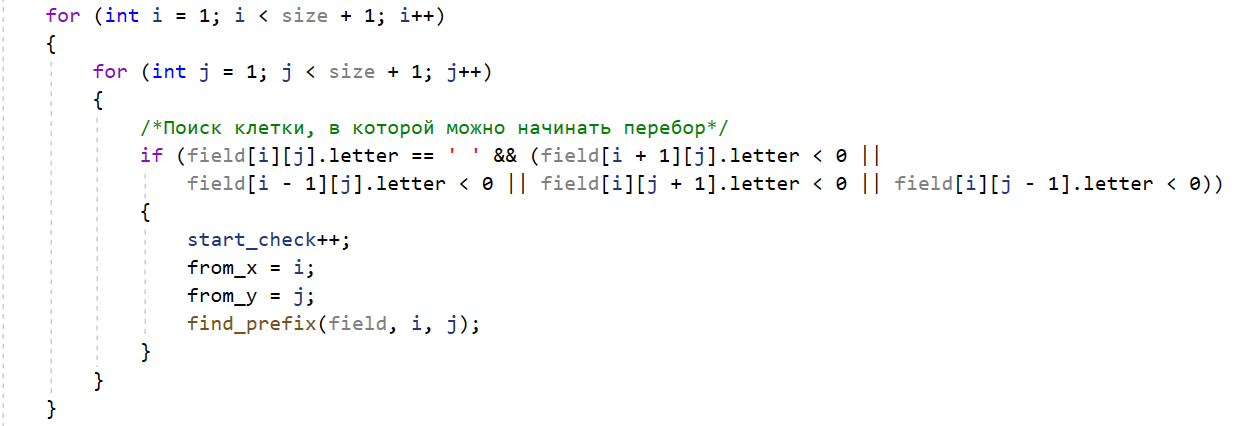
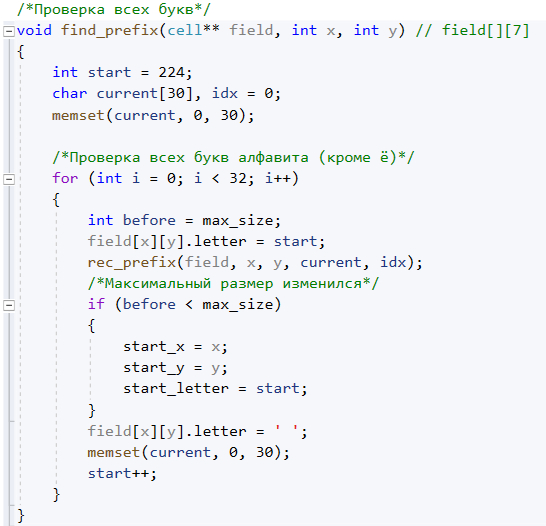
Алгоритм поиска начинается с того, что циклом проходится матрица размером , где N – размер стороны игрового поля. Каждая ячейка проверяется на смежность с другими, то есть проверяется возможно ли в данную ячейку поставить букву так, чтобы она стала частью нового слова по правилам игры.

Рисунок 5 – Функция проверки ячеек на смежность.

Далее в каждой подходящей ячейке проверяются все буквы русского алфавита, то есть перебираются всевозможные вставки.

Рисунок 6 – Функция подстановок всех букв.

Потом относительно каждой буквы (в каждой ячейке) сначала ищутся варианты в префиксном дереве, то есть ищутся всевозможные варианты, потом, если найдено, то рекурсивно достраиваются до итоговых слов в основном словаре. Поскольку максимальное слово в словаре имеет размер 12, то максимально возможная глубина рекурсии пропорциональна максимальному размеру.

В итоге выбирается максимальное возможное слово (первое найденное). Это является слабым местом работы бота. Для улучшения можно добавить эвристический алгоритм, который, например, проверяет возможно ли поставить новое слово, которое целиком включает текущее.

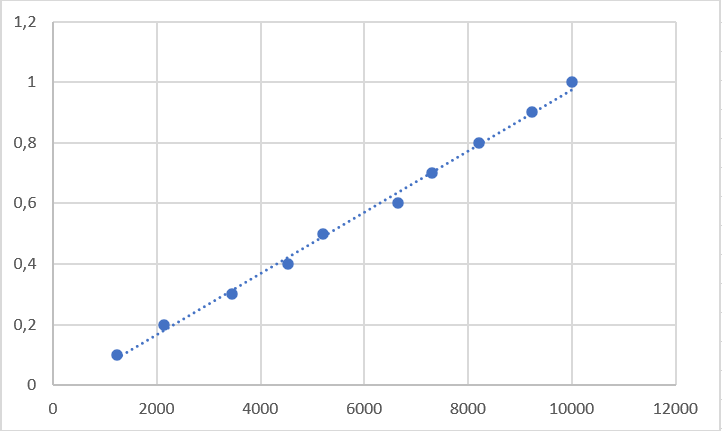
Реализованный бот имеет линейную зависимость от размера поля (чем больше клеток, тем больше рекурсивных вызовов). В целом среднее значение для поиска самого оптимального слова занимает 1 секунду только на поле, имеющим размер 10000 на 10000.

Рисунок 7 – Зависимость времени поиска от размера

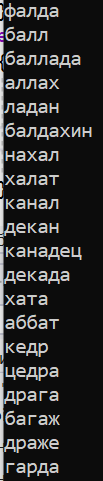
Как видно из графика, зависимость времени поиска от размера является линейной. Также, на больших полях, словарных слов начинает не хватать примерно на 3000 ходе. Полная игра двух ботов самих с собой на поле 5 на 5 занимает меньше секунды.

Рисунок 8 – Набор слов, выбранных в игре двух ботов

#### **Остальные алгоритмы**

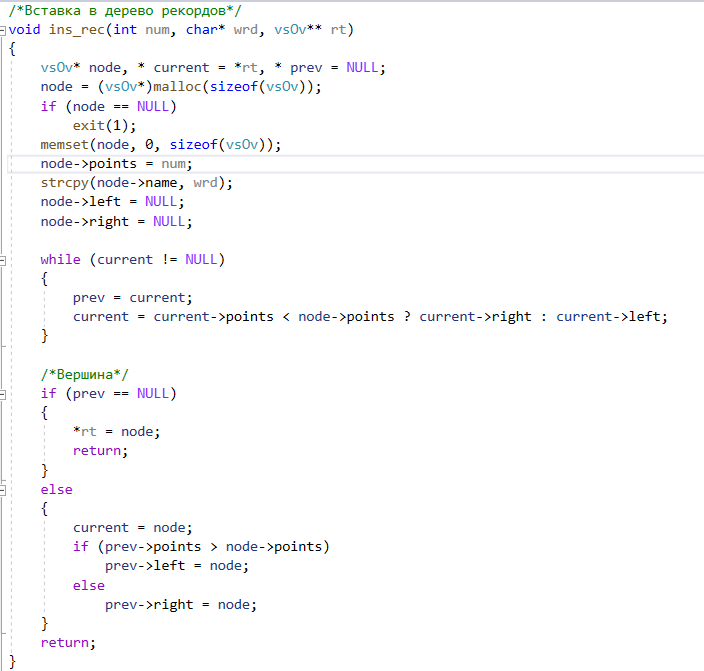
Помимо поиска, в программе реализованы также другие алгоритмы. Такие, как занесение и вывод рекордов на экран. Сделано это с помощью бинарного дерева.

Рисунок 9 – Вставка в бинарное дерево

Бинарное дерево выбрано для того, чтобы было удобно вставлять и выводить новые рекорды. Аналогичным способом могла бы быть реализация таблицы рекордов через линейный список, но операция добавления узла там значительно хуже, чем добавление узла в бинарное дерево в среднем случае, конечно, бинарное дерево может вырождаться, но, на самом деле, происходит это крайне редко, так что в общем случае, бинарное дерево является более оптимальным вариантом, нежели линейный список.

Для хранения использованных в игре слов использовался линейный список, хотя тут тоже лучшим решением была бы реализация дерева.

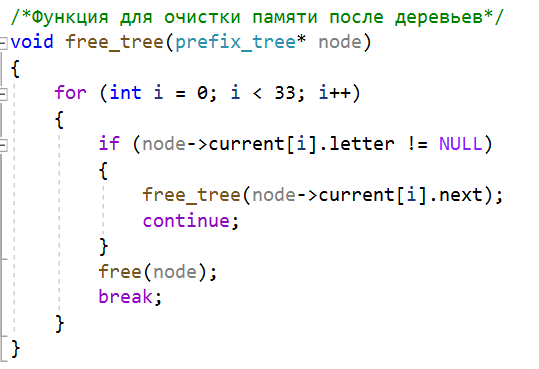
Для очистки памяти после выполнения программы, реализован рекурсивный алгоритм обхода префиксных деревьев

Рисунок 10 – Рекурсивная очистка префиксных деревьев

##### **Графика**

Для реализации графической составляющей игры была использована библиотека “wincon.h”. С помощью Wincon можно создавать различные программы и даже игры в консоли. В целом все сводится к банальной прорисовке элементов на экране. Основными будут функции перемещения по консоли к заданной точке (gotoxy()), функция блокирования/разблокирования экрана(con\_draw\_lock(), con\_drow\_release()). Консоль поддерживает 16 цветов, чего, в принципе, для простой в графическом плане реализации игры балды достаточно.

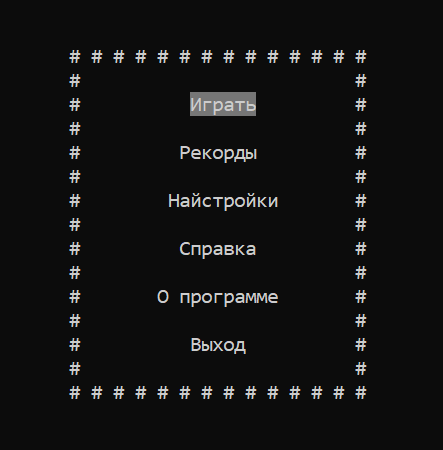
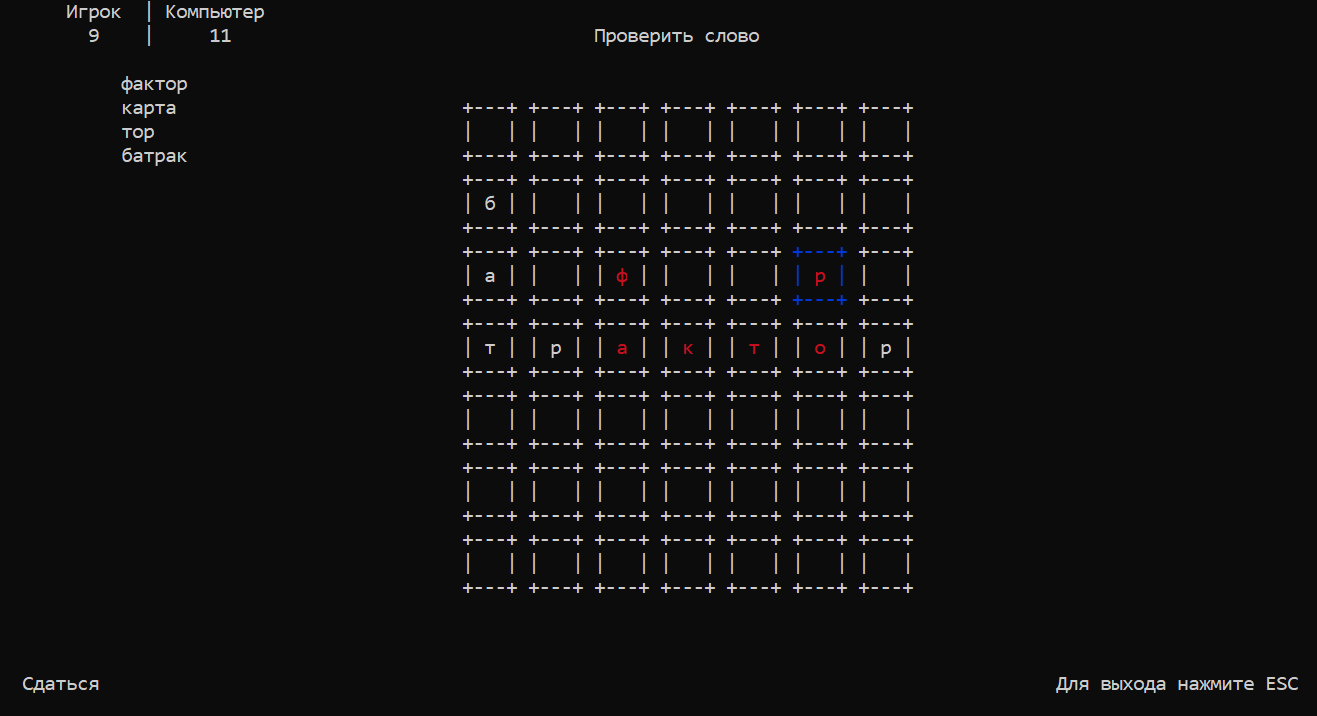
Для того, чтобы каждый раз не перерисовывать с нуля все диалоговые окна, было сделано так, что перерисовываются только две кнопки, которые участвуют в перемещении.

Рисунок 11 – Простое главное меню

Рисунок 12 – Реализация игрового поля.

###### **Заключение**

Реализация игры была является весьма продуктивным способом поднять практический уровень умения программировать. Были использованы и реализованы различные структуры данных: разные виды деревьев, линейные списки, структуры. Также были использованы функции из большого количества библиотек. Получены базовые навыки работы с простой графической библиотекой.