Министерство образования и науки Российской Федерации

1. Санкт-Петербургский Политехнический Университет Петра Великого
2. –
3. Институт кибербезопасности и защиты информации

**КУРСОВАЯ РАБОТА**

«Реализация игры "Великий комбинатор" с применением динамических структур данных»

1. по дисциплине «Структуры данных»

Выполнили

студенты гр. 4851003/90002 Голуб В.В



1. Руководитель

ст. преподаватель

2. Санкт-Петербург
3. 2020

**Содержание**

|  |  |  |  |
| --- | --- | --- | --- |
| Введение......................................................................................................... | |  | 3 |
| 1 | Теоретические сведения о связном списке, описание игры "Великий комбинатор"...................................................................... |  | 5 |
| 1.1 | Правила игры "Великий комбинатор"…......................................... |  | 5 |
| 1.2 | Выбор стратегии игры...................................................................... |  | 6 |
| 1.3 | Худший и средний случай для выбранной комбинации .............. |  | 8 |
| 1.4 | Выбор структуры данных………………………………………… |  | 9 |
| 1.5 | Связный список.................................................................................. |  | 9 |
| 2 | Разработка и тестирование игры "Великий комбинатор" с применением связного списка......................................................... |  | 11 |
| 2.1 | Разработка игры................................................................................. |  | 11 |
| 2.2 | Блок-схемы алгоритмов.................................................................... |  | 14 |
| 2.3 | Тестирование игры............................................................................ |  | 16 |
| Заключение.................................................................................................... | |  | 23 |
| Список использованных источников.......................................................... | |  | 24 |
| Приложение................................................................................................... | |  | 25 |

|  |
| --- |
|  |

**Введение**

В середине 70-х годов широкую популярность в Англии получила игра “великий комбинатор”. Игра “великий комбинатор”- относится к логическим, комбинаторным играм.

При реализации данного приложения необходимо использовать структуру данных – программную единицу, позволяющую хранить и обрабатывать множество связанных данных.

Задачи, решаемые с использованием ПК, крайне редко отражаются в битовом языке. Как правило, данные выражаются в форме чисел, текстов, символов и более сложных структур (последовательностей, списков и деревьев). Поскольку предметом изучения информатики являются свойства и закономерности процессов работы с информацией, а структуры данных используются для хранения информации в упорядоченном виде, значимость структур данных очевидна. В зависимости от конкретных задач, данные нужно хранить в подходящем формате.

В современном мире существуют различные структуры данных. Массивы, множества, записи представляют собой статические структуры, так как их размеры остаются неизменными в течение всего времени работы программы. Однако нередко для решения определенных задач требуется, чтобы структуры данных меняли свои размеры в ходе работы программы. Для таких задач существуют структуры данных, которые называются динамическими, например, стеки, очереди, списки, деревья и другие. Описание динамических структур с помощью массивов, записей и файлов приводит к не экономному использованию памяти и увеличивает время решения задач. Каждая компонента любой динамической структуры представляет собой запись, содержащую по крайней мере два поля: одно поле типа указатель, а второе – для размещения данных.

Целью данной курсовой работы является изучение различных структур данных и реализация игры "Великий комбинатор", моделирующей жизнь колонии клеток, на языке программирования C с использованием динамических структур данных.

Задачи:

1. Исследовать существующие динамические структуры данных;
2. Выбрать структуру данных, подходящую для реализации игры "Великий комбинатор";
3. Разработать алгоритм приложения с использованием подходящей структуры данных;
4. Оценить эффективность работы приложения.

# Теоретические сведения о связном списке, описание игры

# "Великий комбинатор"

* 1. **Правила игры “Великий комбинатор”**

Правила игры крайне просты. Играют двое. Один из игроков загадывающий, записывает секретную комбинацию из любых четырех цифр от 1 до 6 (допускаются повторения), называемую кодом. Второй игрок, отгадывающий, пытается раскрыть код, высказывая разумные предположения, называемые пробами.

Каждая проба, как и код, представляет собой произвольную комбинацию из четырех цифр в диапазоне от 1 до 6. Отгадывающий игрок сообщает пробу загадывающему, и тот должен ответить, сколько цифр в пробе совпадает с цифрами кода как по положению, так и по величине и сколько из остальных цифр пробы входят в код, но стоят на другом месте. Цифры которые совпадают как по положения так и по величине называются черными, цифры которые входят в код, но стоят не на своем месте называются белыми.

Так, на пробу 1123 при коде 4221 будет получен ответ: «Одна цифра совпадает и стоит на том же месте, и еще одна совпадает, но стоит на другом месте».

Таблица 23.1. Великий комбинатор. Пример партии

Код: 4651

Проба 1: 2345. О точных попаданий. 2 совпадения по значению.

Проба 2: 4516. 1 точное попадание. 3 совпадения по значению.

Проба 3: 5461. 1 точное попадание. 3 совладения по значению.

Проба 4: 4165. 1 точное попадание. 3 совпадения по значению.

Проба 5: 4615. 2 точных попадания. 2 совпадения по значению.

Проба 6: 4651. 4 точных попадания. Игра окончена

Тур игры продолжается до тех пор, пока отгадывающий не назовет пробу, в точности совпадающую с кодом, т. е. пока не отгадает код. После этого игроки меняются ролями и проводят еще один тур. Победителем считается тот из игроков, кто определит код противника за меньшее число проб. Хотя здесь не последнюю роль играет везение, тем не менее игрок, систематически делающий правильные умозаключения из получаемой информации, должен иметь лучшие результаты по итогам нескольких партий.

Практически вы должны пытаться выводить из ответов на ваши пробы отрицательные следствия относительно того, какие коды невозможны.

* 1. **Выбор стратегии игры.**

Написать программу, имитирующую роль загадывающего, не составляет труда. Сложности возникают на моменте когда необходимо, чтобы компьютер мог быть в роли угадывающего.

Боб Кули из Lawrence Livermore Laboratory и Д. Кнут разработали довольно близкие стратегии, позволяющие ЭВМ достигнуть высокого класса игры. Центральное место в обеих стратегиях занимает идея **пространства решений**. Начальное пространство решений Р0 состоит из всех возможных кодов, после i-й пробы Gi пространство Pi состоит из всех тех членов пространства Pi?1, которые не опровергаются ответом Ri. Иными словами, пространство Pi — это множество всех комбинаций, которые все еще могут быть кодом; задача угадывающего — свести пространство к одному элементу.

Стратегия, предложенная Кули, несколько проще. Пробой Gi пусть будет любая случайно выбранная комбинация с одной повторяющейся цифрой, например 4311, 6552 или 1335. Выполните эту пробу и постройте пространство Pi на основе ответа Ri. Новая проба Gi+1 ищется по пространству Рi, i ? 1, путем поочередного сравнения всех комбинаций С из Pi с пробой Gi. В качестве следующей пробы выбирается наименее похожая на Gi комбинация С. Мерой сходства служит число точных совпадений, а в случае равенства — число цифр, совпадающих по значению, но расположенных по-другому. Так, среди трех комбинаций 2641, 2356 и 1345 наиболее похожей на 2345 будет 1345, а 2641 — наименее похожей. Если имеется несколько наименее похожих комбинаций, то можно выбрать любую кандидатуру случайным образом. Тур прекращается, когда будет получен ответ «четыре точных попадания», и, разумеется, в случае пространства из одного элемента в качестве следующей пробы всегда надо брать этот элемент. Как показывают эксперименты, размеры пространства решений сокращаются после каждой пробы примерно в 4 раза и никогда не требуется более шести проб.

Стратегия Д. Кнута показалась более эффективной, поэтому была выбрана она.

В основе алгоритма лежит наблюдение, что нам хотелось бы сделать пространство Pi как можно меньше. Следовательно, мы выбираем пробу Gi, минимизирующую |Pi| по всем возможным ответам Ri. Кандидатом в Gi будет любая комбинация С. Попробуйте применить все возможные комбинации С в качестве проб к пространству Pi?1; пусть Sc, <0,0> обозначает число членов Pi?1, дающих в ответе нулевое число точных совпадений и нулевое Число совпадений только по цвету[38] Sc, <0,1> — число членов, дающих соответственно нуль и одно совпадение и т. д. до Sc, <4,0> для наиболее удачной комбинации с четырьмя точными совпадениями. Введем обозначение



Теперь в качестве пробы Gi выберите комбинацию С, минимизирующую Sc (при наличии нескольких таких С выберите комбинацию из Pi?1, если это возможно; если же нет — делайте случайный выбор). Вы, вероятно, уже заметили, что этот алгоритм можно использовать для предварительного анализа *великого комбинатора*, так чтобы в процессе игры не был нужен никакой анализ комбинаций. Проделав такой анализ, Кнут показал, что оптимальной первой пробой при использовании его стратегии будет 1122.

* 1. **Худший и средний случаи для выбранной стратегии.**

#### Худший случай: алгоритм пяти предположений

В 1977 году [Дональд Кнут](https://ru.qwe.wiki/wiki/Donald_Knuth) продемонстрировал, что взломщик кода может решить шаблон за пять или меньше ходов, используя алгоритм, который постепенно уменьшает количество возможных шаблонов. Алгоритм работает следующим образом:

1. Создайте набор S из 1296 возможных кодов (1111, 1112 ... 6665, 6666)
2. Начните с начального предположения 1122 (Кнут приводит примеры, показывающие, что другие первые предположения, такие как 1123, 1234, не дают результата при пяти попытках для каждого кода)
3. Сыграйте в угадание, чтобы получить ответ из цветных и белых колышков.
4. Если ответ - четыре цветных колышка, игра выиграна, алгоритм завершается.
5. В противном случае удалите из S любой код, который не дал бы такой же ответ, если бы это (предположение) был код.
6. Примените технику [минимакса](https://ru.qwe.wiki/wiki/Minimax) , чтобы найти следующее предположение, следующим образом: для каждого возможного предположения, то есть для любого неиспользуемого кода 1296, а не только кода в S, вычислите, сколько возможностей в S будет исключено для каждого возможного показателя цветного / белого колышка. Оценка предположения - это *минимальное* количество возможностей, которые оно может исключить из S. Один проход через S для каждого неиспользованного кода 1296 даст счетчик совпадений для каждой найденной оценки цветных / белых колышков; оценка цветного / белого колышка с наибольшим количеством совпадений устранит наименьшее количество возможностей; рассчитать оценку предположения, используя «минимальное исключенное количество» = «количество элементов в S» - (минус) «максимальное количество совпадений». Из набора догадок с *максимальным* баллом выберите одно в качестве следующего предположения, выбирая по возможности член S. (Кнут следует соглашению о выборе предположения с наименьшим числовым значением, например, 2345 ниже, чем 3456. Кнут также приводит пример, показывающий, что в некоторых случаях ни один член S не будет среди предположений с наибольшим количеством очков, и, таким образом, предположение не может выиграть следующий ход, но еще будет необходимо обеспечить победу через пять.)
7. Повторите с шага 3.

#### Средний случай

Последующие математики находили различные алгоритмы, которые уменьшали среднее количество оборотов, необходимых для решения шаблона: в 1993 году Кенджи Кояма и [Тони В. Лай](https://ru.qwe.wiki/w/index.php?title=Tony_W._Lai&action=edit&redlink=1) провели исчерпывающий [поиск в глубину,](https://ru.qwe.wiki/wiki/Depth-first_search) показав, что оптимальный метод решения случайного кода может обеспечить в среднем 5625/1296 = 4,3403 хода для решения с наихудшим сценарием из шести ходов.

* 1. **Выбор структуры данных**

При реализации игры “Великий комбинатор”, возникает вопрос какую структуру данных использовать для хранения значений из пространства решений.

Для данных задач подходит статический массив, однако список предоставляет больше возможностей, так как является динамической структурой данных. Размер списка ограничивается только доступным объемом машинной памяти.

Помимо массива, для решения задачи с хранением могло справится и B-дерево, в котором операция удаления происходила бы быстрее, но т.к операция добавления занимает больше времени, был выбран линейный список.

* 1. **Связный список**

Линейный список представляет собой последовательность n >= 0 узлов X1, X2,…, Xn, важнейшей структурной особенностью которой является такое расположение элементов списка один относительно другого, как будто они находятся на одной линии. Иначе говоря, в такой структуре должно соблюдаться следующее условие: если n > 0 и X1 является первым узлом, а Xn – последним, то k-й узел Xk следует за Xk-1 и предшествует узлу Xk+1 для всех 1 < k < n. [1, с. 267]

Связный список является простейшим типом данных динамической структуры, состоящей из элементов (узлов). Каждый узел включает в себя в классическом варианте два поля:

- данные (в качестве данных может выступать переменная, объект класса или структуры и т. д.);

- указатель на следующий узел в списке.

Доступ к списку осуществляется через указатель, который содержит адрес первого элемента списка, называемый корнем списка. Основное назначение связного списка – предоставление механизма для хранения и доступа к произвольному количеству данных.

Связные списки бывают двух видов: односвязные и двусвязные. Связный список, содержащий только один указатель на следующий элемент, называется односвязным. Связный список, содержащий два поля указателя – на следующий элемент и на предыдущий, называется двусвязным.

По способу связи элементов различают линейные и циклические списки. Связный список, в котором, последний элемент указывает на NULL, называется линейным. Связный список, в котором последний элемент связан с первым, называется циклическим.

Таким образом, различают 4 основных вида списков:

**Односвязный линейный список (ОЛС)**. Каждый узел ОЛС содержит 1 поле указателя на следующий узел. Поле указателя последнего узла содержит нулевое значение (указывает на NULL).

**Односвязный циклический список (ОЦС)**. Каждый узел ОЦС содержит 1 поле указателя на следующий узел. Поле указателя последнего узла содержит адрес первого узла (корня списка).

**Двусвязный линейный список (ДЛС).** Каждый узел ДЛС содержит два поля указателей: на следующий и на предыдущий узел. Поле указателя на следующий узел последнего узла содержит нулевое значение (указывает на NULL). Поле указателя на предыдущий узел первого узла (корня списка) также содержит нулевое значение (указывает на NULL).

**Двусвязный циклический список (ДЦС).** Каждый узел ДЦС содержит два поля указателей: на следующий и на предыдущий узел. Поле указателя на следующий узел последнего узла содержит адрес первого узла (корня списка). Поле указателя на предыдущий узел первого узла (корня списка) содержит адрес последнего узла.

# Разработка и тестирование игры "Великий комбинатор" с применением связного списка

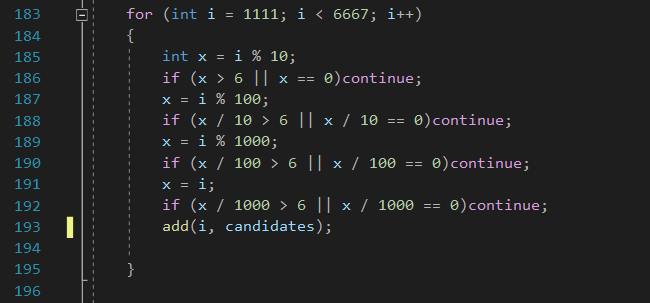
* 1. **Разработка игры**

Для реализации игры "Великий комбинатор", был создан связный список.

Каждый элемент списка содержит:

* значение data, в котором хранится число из поля возможных решение;
* ссылку на следующий элемент;

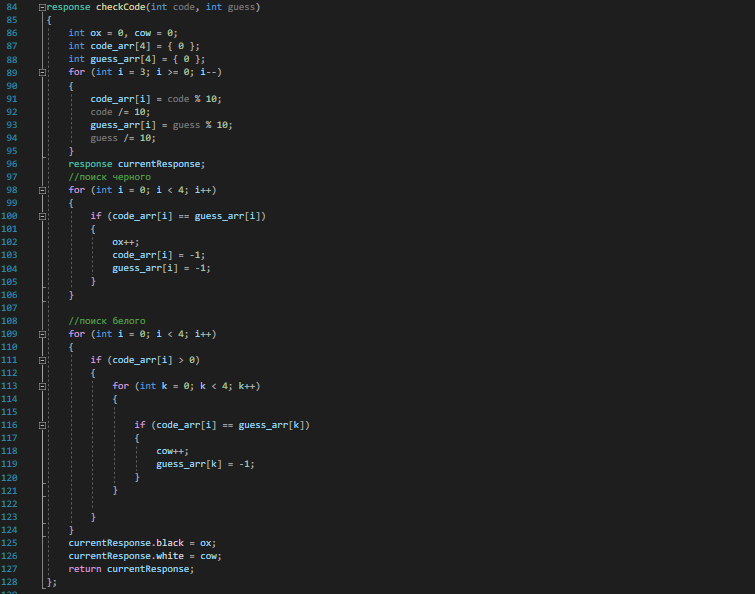
При запуске игры создается список. Который заполняется всеми комбинациями чисел от 1111 до 6666, из цифр от 1 до 6.



Заполнение происходит при помощи цикла for(от 1111 до 6666), для того чтобы убрать цифры от 7 до 9, применяется поразрядная проверка.

Также методом случайной генерации чисел создается код, который необходимо отгадать.

Далее в цикле пока не будет угадан код происходит сравнения пробы и кода, при помощи функции checkCode.



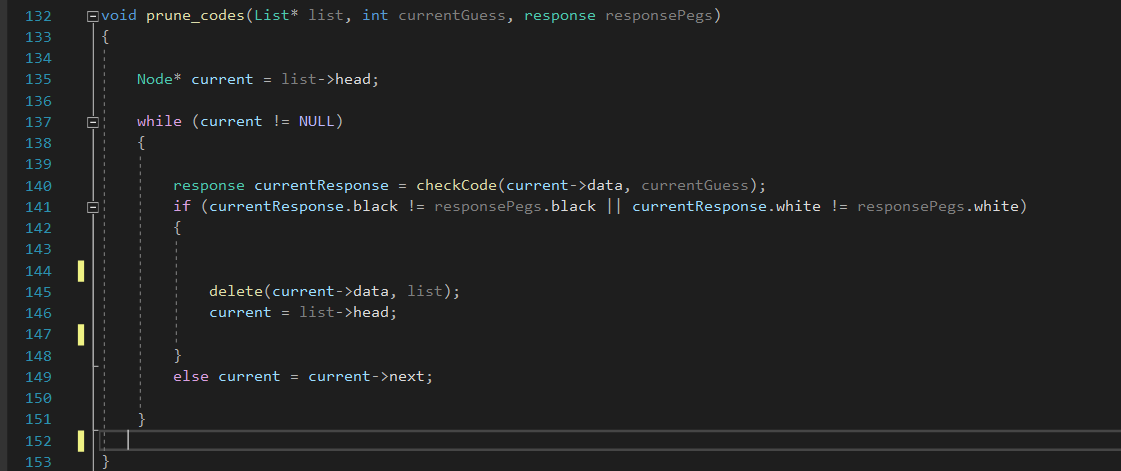
После получения чисел, функция разбивает числа на цифры. Помещая цифры в массив, где номер ячейки массив, является позицией цифры в числе.

Сама функция разбита на два блока. в первом блоке происходит проверка-является ли число чёрным, то есть совпадает по значению и по величине. В случается если цифра является черной то счетчик увеличивается на единицу, а ячейка в массиве становится минус единицей. Во втором блоке происходит поиск есть ли в пробе цифры, которые являются белыми, то есть теми которые входят в код, но стоят не на своем месте.

После завершения работы, функция возвращает количество черных и белых цифр в пробе.

После этого, если количество черных не равно четырем, то происходит удаления всех чисел, которые имеют отличное от пробы количество черных и белых цифр.

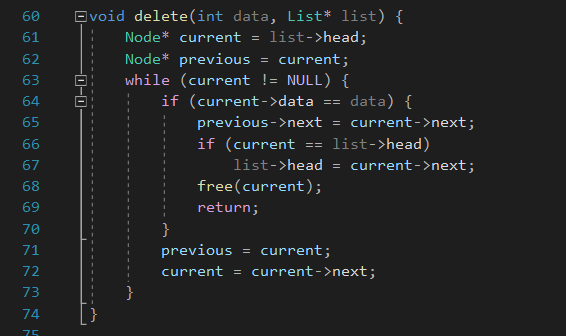
Реализовано это при помощи функции prune\_codes.



В функцию передаются: указатель на “голову” списка, текущее предположение, данные о количестве черных и белых цифр.

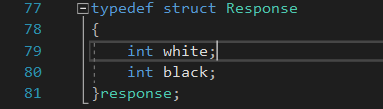
Для начала создается переменная, в которой хранится указатель на “голову” списка. Затем если “голова” не пустая, происходит сравнение чисел из поля всех решений с текущем предположением, в случае когда количество черных или белых отличается, происходит удаление этого числа из списка, при помощи функции delete, после чего указателю снова передается адрес на “голову”, во избежания потери ссылки на следующий элемент.

В функцию delete передается указатель и значение которое необходимо удалить.



В самой функции происходи проверка на то, что является ли узел пустым. Если узел не является пустым, то начинается поиск узла со значением которое было передано в функцию. После чего указатель от предыдущего этому элементу, переставляется на следующий, а сам узел удаляется.

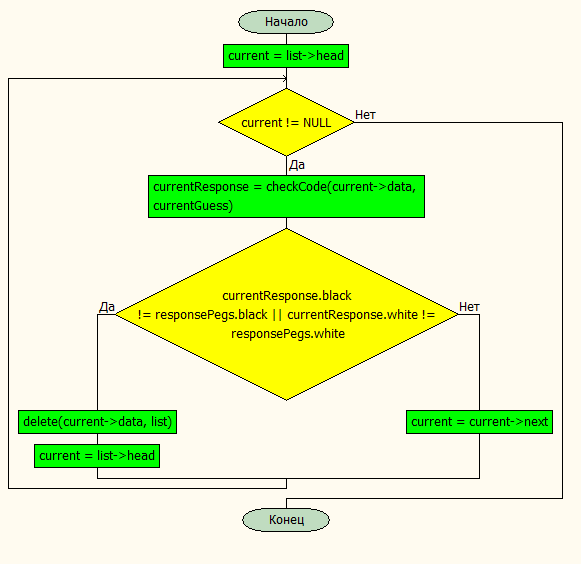
Для реализации функций checkCode и prune\_codes была создана структура данных response.

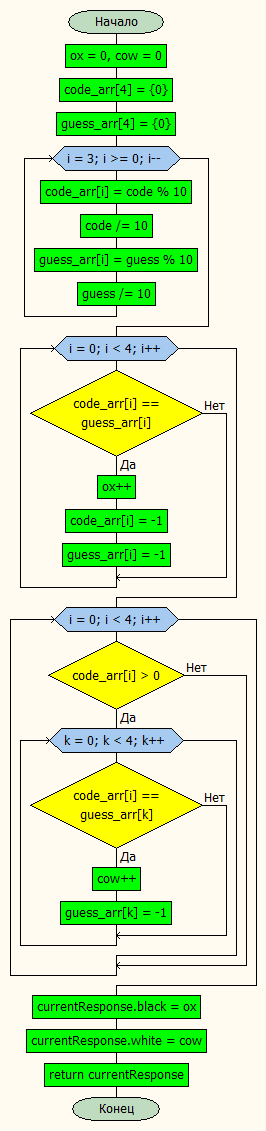


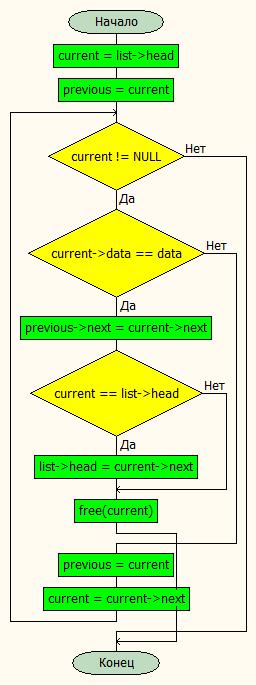
В которой хранятся:

* значение white, в котором хранится информация о количестве белых.
* значение black, в котором хранится информация о количестве черных.
  1. **Блок-схемы алгоритмов**

*Рис. Блок-схема функции prune\_codes*



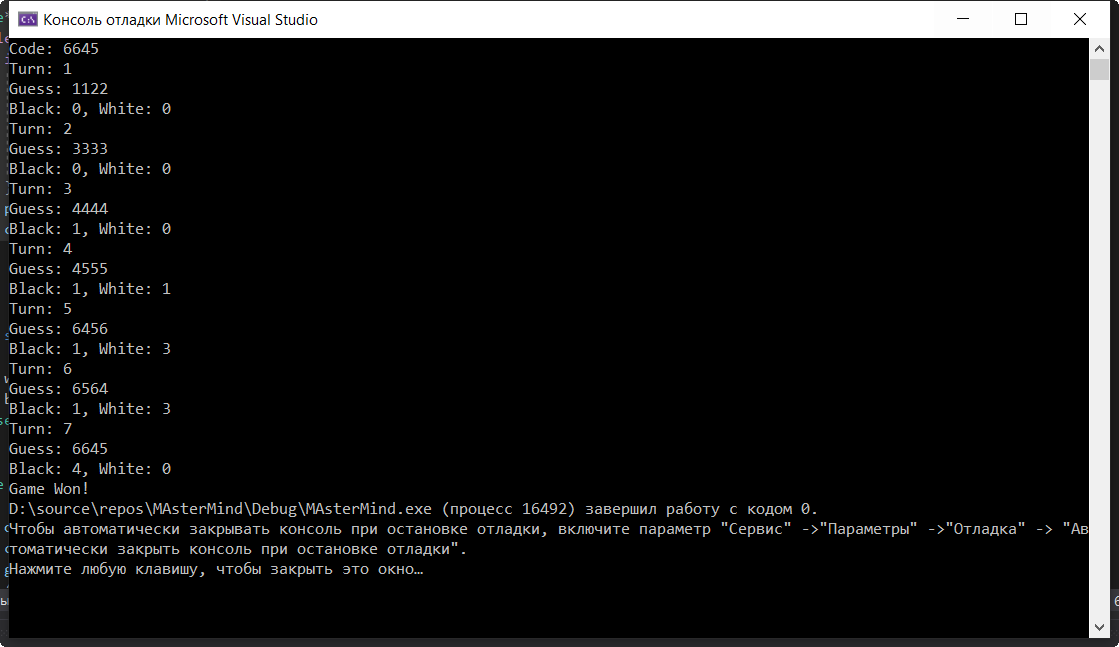
 *Рис. Блок-схема функции checkCode*



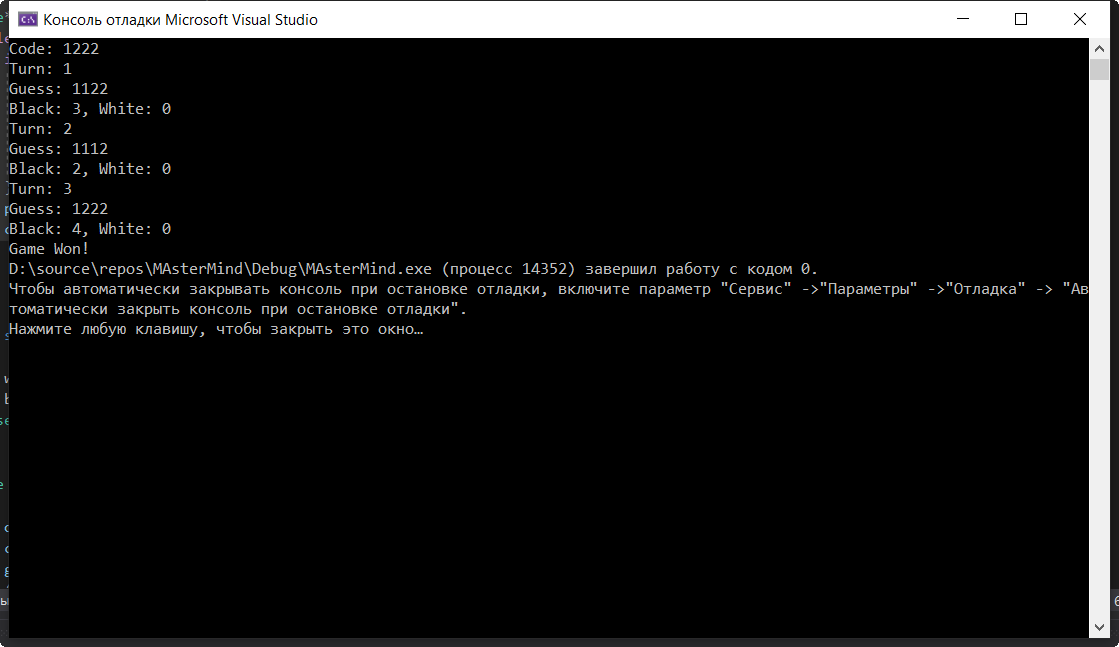
*Рис. Блок-схема функции delete*

* 1. **Тестирование игры**

Были проведены тесты основной целью которых было узнать, сколько в среднем требуется ходов для победы. Для большей чистоты эксперимента было принято тестировать на демонстрационной версии, то есть когда компьютер играет сам с собой. Нижу представлены результаты нескольких тестов



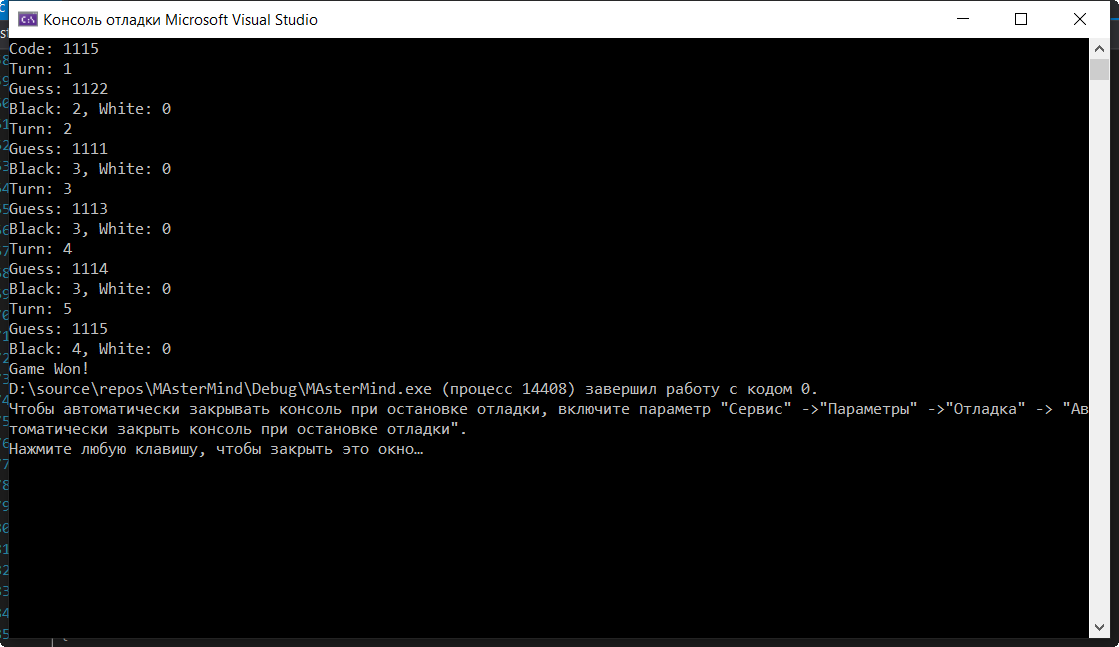
Случай является худшим так как, для победы потребовалось 7 ходов

* + 1. 

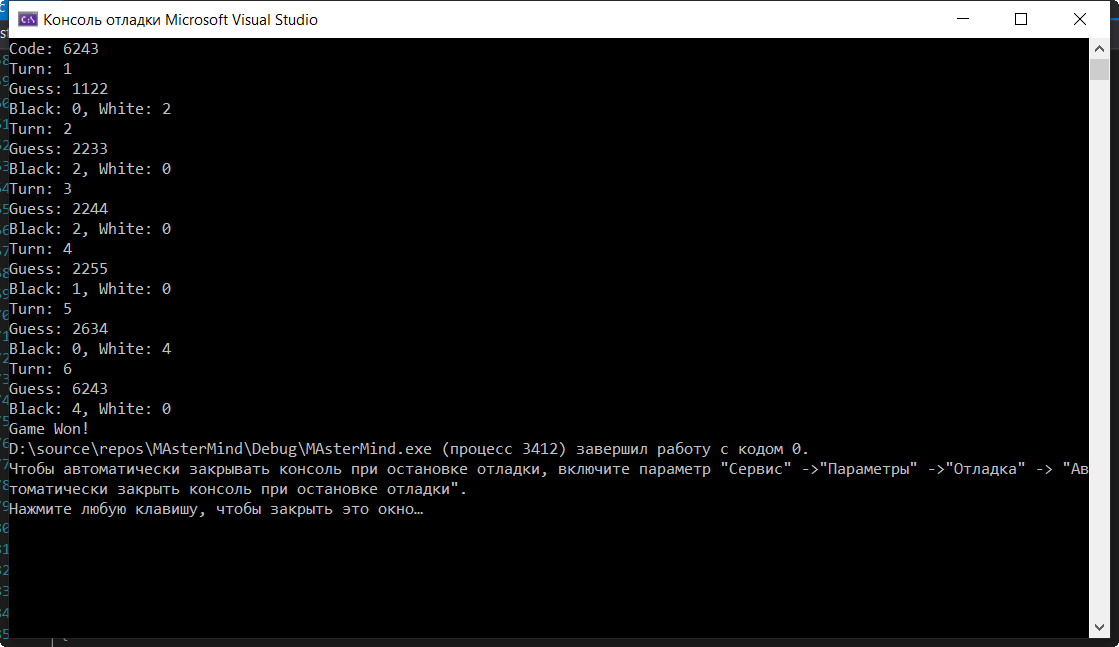
Случай близок к наилучшему, так потребовалось всего 3 хода.

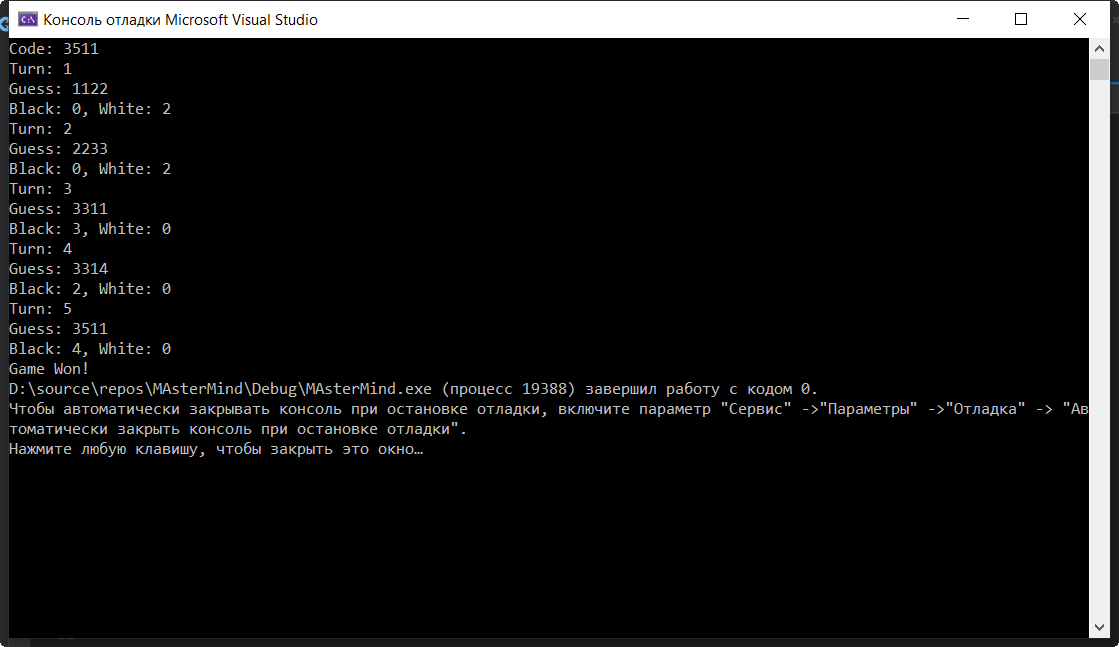
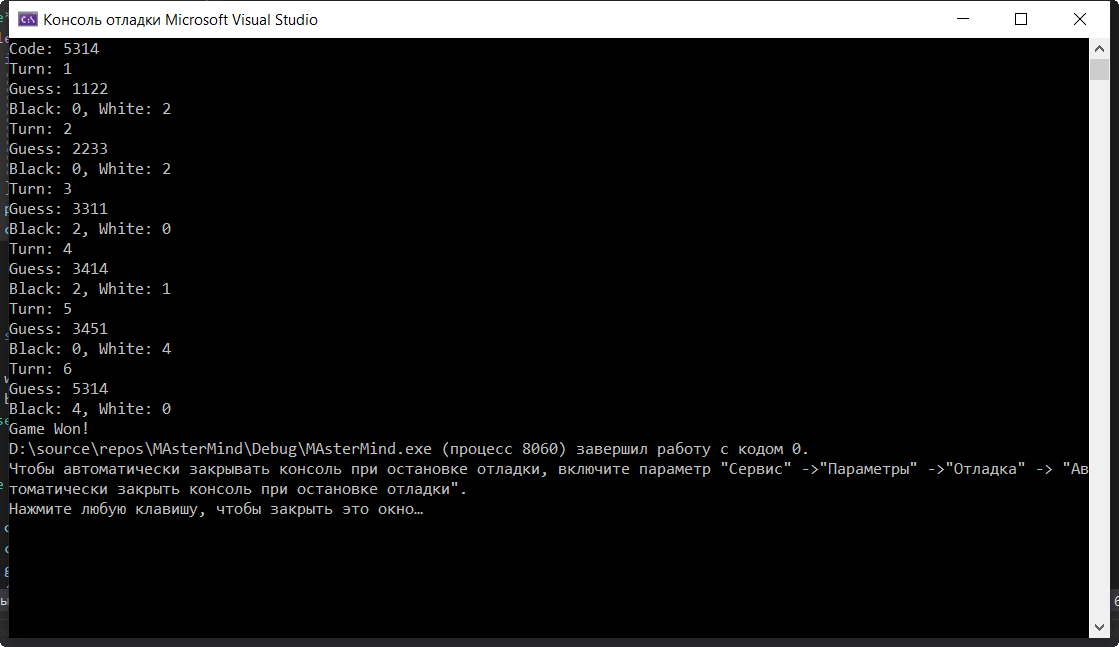
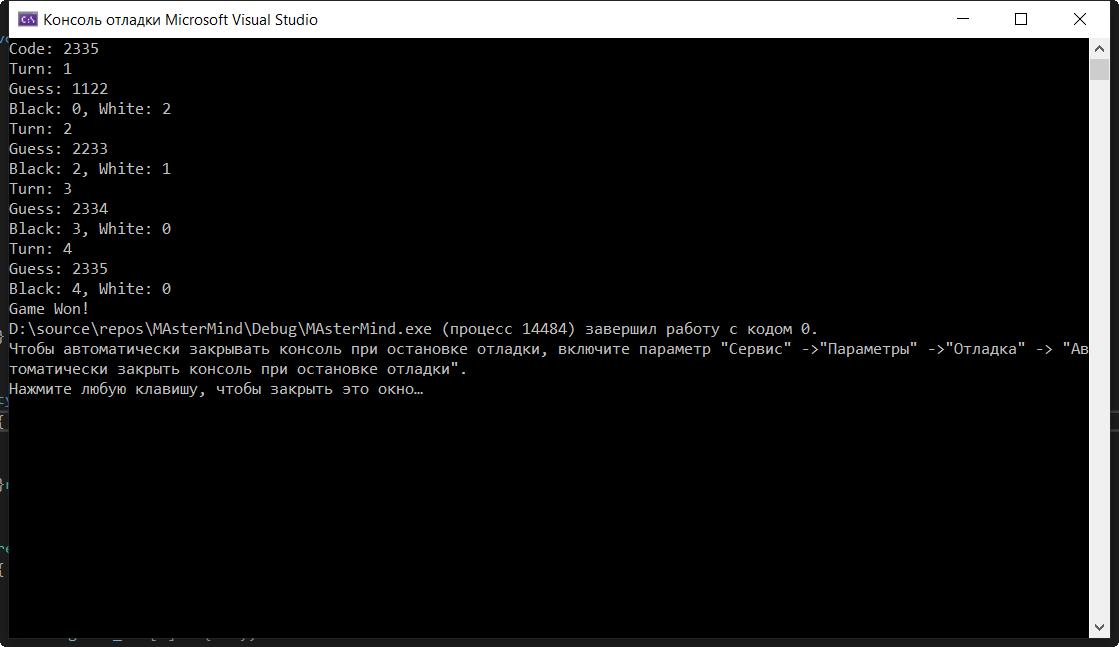
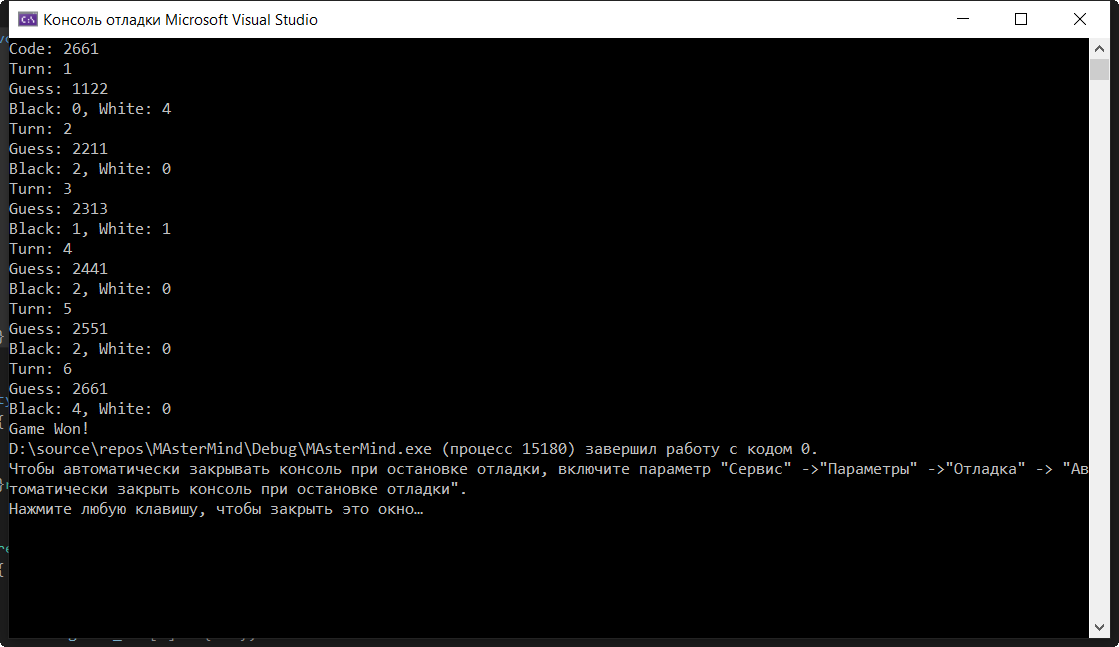
Лучшим является случая когда для отгадывания потребуется один ход.





* + 1. На 5 ходу компьютер находится максимально блико к решению, и реагируя на информацию о 4 белых, уменьшение поля возможных решений происходит до 24 вариантов.



* + 1. 
    2. 
    3. На примере этого теста видно что на количество попыток играет удача, так как начиная со второго хода, компьютер берет подряд пробы из оставшихся чисел, и ему удается за три хода найти код.
    4. 

После проведения тестов и вычислений было получено, что в среднем для победы требуется 5-6 ходов. Все зависит от того насколько близкая к коду проба была выбрана на втором ходу.

**Заключение**

В ходе работы были изучены динамические структуры данных и реализована игра "Жизнь" на основе связного списка. Была найдена теоретическая сложность разработанного алгоритма. Созданная игра была протестирована на правильность и быстродействие.

Разработка приложения показала, что сложность реализации и эффективность работы программы существенно зависит от выбора правильной структуры данных. Структуры данных выбираются из соображений оптимального выполнения ключевых задач посредством определённых алгоритмов, которые работают лучше всего со своим типом структур данных. Так, например, динамические структуры данных предоставляют ряд дополнительных возможностей, таких как увеличение объема обрабатываемых данных, возможность освобождения памяти и создания структуры данных переменного размера.

**Список использованных источников**

1. Дональд Э. Кнут Искусство программирования. Том 1. Основные алгоритмы / Перевод с английского под ред. С.Г. Тригуба, Ю.Г. Гордиенко, И.В. Красикова. – К.: Издательство «Диалектика», 2019. – 720 с.
2. Гарднер М. Математические досуги / Перевод с английского Ю.А. Данилова под ред. Я.А. Смородинского. – М., Издательство «Мир», 1972. – 496 с.
3. Уэзерелл Чарлз Этюды для программистов / Перевод с английского под ред. Ю.М. Баяковского. – М.: Издательство «Мир», 1982. – 288 с.
4. Елена Вставская Связные списки // Программирование C и C++ – СПб., 2014. – URL: https://prog-cpp.ru/data-list/ – (дата обращения: 31.05.2020).

**Приложение**

* Исходный код разработанного приложения

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

typedef struct node{

int data;

struct node\* next;

}Node;

typedef struct list {

Node\* head;

}List;

Node\* createnode(int data) {

Node\* newNode = malloc(sizeof(Node));

if (!newNode) {

return NULL;

}

newNode->data = data;

newNode->next = NULL;

return newNode;

}

List\* makelist() {

List\* list = malloc(sizeof(List));

if (!list) {

return NULL;

}

list->head = NULL;

return list;

}

void display(List\* list) {

Node\* current = list->head;

if (list->head == NULL)

return;

for (; current != NULL; current = current->next) {

printf("%d\n", current->data);

}

}

void add(int data, List\* list) {

Node\* current = NULL;

if (list->head == NULL) {

list->head = createnode(data);

}

else {

current = list->head;

while (current->next != NULL) {

current = current->next;

}

current->next = createnode(data);

}

}

void delete(int data, List\* list) {

Node\* current = list->head;

Node\* previous = current;

while (current != NULL) {

if (current->data == data) {

previous->next = current->next;

if (current == list->head)

list->head = current->next;

free(current);

return;

}

previous = current;

current = current->next;

}

}

typedef struct Response

{

int white;

int black;

}response;

response checkСode(int code, int guess)

{

int ox = 0, cow = 0;

int code\_arr[4] = { 0 };

int guess\_arr[4] = { 0 };

for (int i = 3; i >= 0; i--)

{

code\_arr[i] = code % 10;

code /= 10;

guess\_arr[i] = guess % 10;

guess /= 10;

}

response currentResponse;

//поиск черного

for (int i = 0; i < 4; i++)

{

if (code\_arr[i] == guess\_arr[i])

{

ox++;

code\_arr[i] = -1;

guess\_arr[i] = -1;

}

}

//поиск белого

for (int i = 0; i < 4; i++)

{

if (code\_arr[i] > 0)

{

for (int k = 0; k < 4; k++)

{

if (code\_arr[i] == guess\_arr[k])

{

cow++;

guess\_arr[k] = -1;

}

}

}

}

currentResponse.black = ox;

currentResponse.white = cow;

return currentResponse;

};

void prune\_codes(List\* list, int currentGuess, response responsePegs)

{

Node\* current = list->head;

while (current != NULL)

{

response currentResponse = checkСode(current->data, currentGuess);

if (currentResponse.black != responsePegs.black || currentResponse.white != responsePegs.white)

{

// printf("%d \n", current->data);

delete(current->data, list);

current = list->head;

//current = current->next;

// printf("%d \n", current->data);

/\*display(list);\*/

}

else current = current->next;

}

// display(list);

}

//\*--------------------------------------Функция для реализации интерфейса-------------------------------------------\*//

void eatline(void) {

char c;

while ((c = getchar()) != '\n');

}

//\*------------------------------------------------------------------------------------------------------------------\*//

int main() {

List\* candidates = makelist();

List\* combinations = makelist();

int code = 0;//загадываемое число

bool won = false;

srand(time(NULL));

for (int i = 0; i < 4; i++)

{

code \*= 10;

code += 1 + rand() % 6;

}

/\*code = 2222;\*/

printf("Code: %d \n", code);

for (int i = 1111; i < 6667; i++)

{

int x = i % 10;

if (x > 6 || x == 0)continue;

x = i % 100;

if (x / 10 > 6 || x / 10 == 0)continue;

x = i % 1000;

if (x / 100 > 6 || x / 100 == 0)continue;

x = i;

if (x / 1000 > 6 || x / 1000 == 0)continue;

add(i, combinations);

add(i, candidates);

}

int current\_guess = 1122;

response response\_pegs;

int turn = 0;

while(!won)

{

// display(candidates);

delete(current\_guess, candidates);

delete(current\_guess, combinations);

//display(candidates);

response\_pegs = checkСode(code, current\_guess);

turn++;

printf("Turn: %d \n", turn);

printf("Guess: %d \n", current\_guess);

printf("Black: %d, White: %d \n", response\_pegs.black, response\_pegs.white);

if (response\_pegs.black == 4)

{

won = true;

printf("Game Won!");

break;

}

//display(candidates);

prune\_codes(candidates, current\_guess, response\_pegs);

// display(candidates);

current\_guess = candidates->head->data;

}

}